Introduction to R Programming

DSCOE Team

2017-04-11

Contents

These five lessons are designed to provide a person with the fundamental understanding of the R Programming language. It will cover data structures, data manipulation, as well as basic data visualization. The recommended learning approach is to install R and RStudio on a computer or cloud node, and follow along with the provided material and videos.

Chapter 1

Basic Data Structures

1.1 The R Programming Language

R is an extremely powerful statistical scripting language. It is open-source and used broadly across academia, research organizations, and businesses. It is often the tool of choice for data scientists, data analysts, quantitative financial analysts, and a myriad of other professions. It is used for research at the vast majority of graduate schools. It is currently used by companies like Facebook, Google, the NY Times and Wallstreet financial organizations. Microsoft has invested heavily in integrating R into its desktop and cloud data science tools. Google has written the R Style Guide that is widely used. Facebook data scientists use R to analyze and understand the vast Facebook social network.

Figure 1.1: Figure 1: Facebook created this image with R to show how Facebook connects the world

1.1.1 R or Python?

The world is quickly moving toward leveraging open source data science tools rather than proprietary software. Over the last 5 years R and Python have risen as the two primary open source tools used by data professionals. While there is significant overlap in the capabilities of both languages, in general the R Programming language is better at data analysis and visualization, and Python is better at data acquisition and producing code for production environments. We decided to teach R in this class since it generally has better visualizations, allowing our analysts to tell the data narrative. Additionally, R is generally more accessible across the Department of Defense.

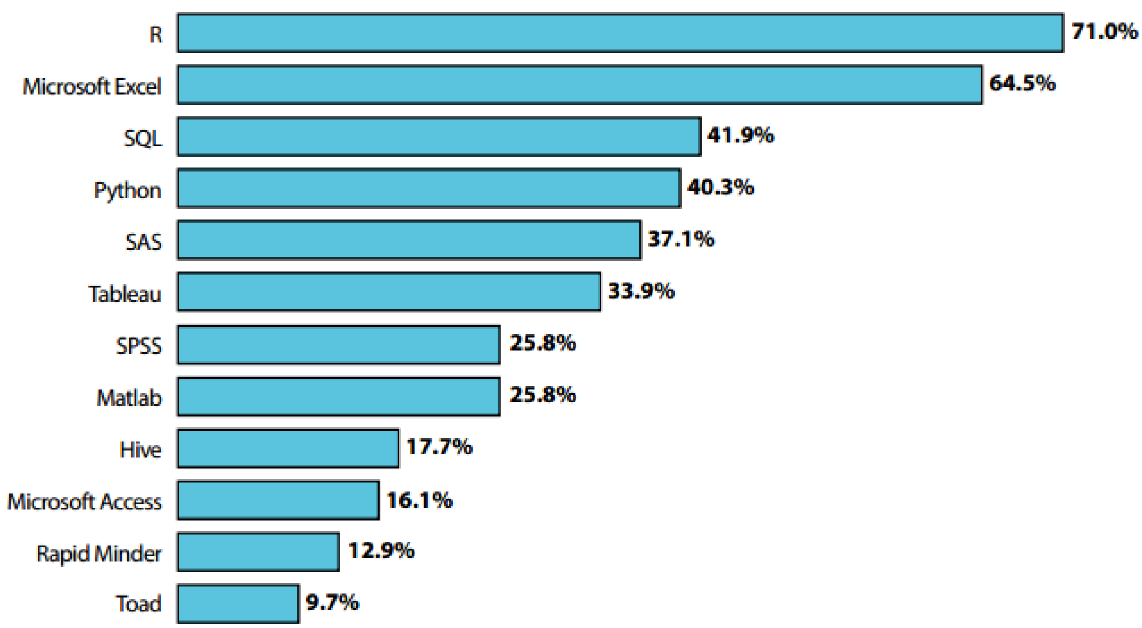


Figure 1.2: Figure 2: LAVASTORM 2014 Survey of Industry: Primary tools used by data scientists

R is open-source and is freely available to download. You can use base R as-is to write and run R scripts. That being said, RStudio has provided a very useful Integrated Development Environment (IDE) or "front-end" for R that is generally easier to use (R is still the "engine"; you can't run RStudio without R). We will primarily use RStudio in this course.

Note that you can also run R from a server in the "cloud". The Army Data Science Center of Education (DSCOE) provides several tutorials that explain how to do this.

1.2 Installation

1. Install Base R by going to http://cran.r-project.org/bin/windows/base/
2. Install RStudio by going to http://www.rstudio.com/products/rstudio/download/

1.3 R Environment and Workspace

Introductory Video:

R is always pointing to a specific directory (or folder) on your computer. This is called your *working directory*. R will always directly read files and write files to this directory. You can see your working directory by typing

```
getwd()
```

```
## [1] "/home/dmbeskow/Dropbox/bookdown-demo-master"
```

If you want to change where your working directory is, you can do this three ways. If you are using RStudio, you can go to *Session -> Set Working Directory*. You can also use the *Files* tab to navigate to your desired working directory, and then click on *More -> Set as Working Directory*. If you want to change your *working directory* using a command (especially if you're using base R), then you can type

```
setwd("C:/Users/beskow/Documents")  ###Make sure you use Forward Slashes in Windows
```

If you want to see the names of files in your working directory without opening Windows Explorer, you can use the command

```
dir()
```

```
##  [1] "01-fundamentals.Rmd"      "02-munging.Rmd"
##  [3] "03-visualization.Rmd"     "04-control.Rmd"
##  [5] "05-dates.Rmd"             "06-exam.Rmd"
##  [7] "07-references.Rmd"        "_book"
##  [9] "book.bib"                 "bookdown-demo_files"
## [11] "bookdown-demo.Rmd"        "bookdown-demo.Rproj"
## [13] "_bookdown_files"          "_bookdown.yml"
## [15] "_build.sh"                "dataframe.PNG"
## [17] "dataWrangling.jpg"        "_deploy.sh"
## [19] "DESCRIPTION"              "dplyr.png"
## [21] "environment.PNG"          "facebook.png"
## [23] "filterColumn.PNG"         "filterRow.PNG"
## [25] "index.Rmd"                "KoreanConflict.csv"
## [27] "LICENSE"                  "list.PNG"
## [29] "matrix.PNG"               "_output.yml"
## [31] "packages.bib"             "preamble.tex"
## [33] "prerequisite-book.Rproj" "rating2.csv"
## [35] "README.md"                "screen1.png"
## [37] "style.css"                "summer.csv"
## [39] "tidyr.png"                "toc.css"
## [41] "vector.PNG"               "whyR2.PNG"
## [43] "whyR.PNG"
```

Note that this gives the names of the files in your working directory, which saves you the time of opening up Windows Explorer to remind yourself what you named your data file.

1.4 Types and Shape of Data

Before we get into data, I first want to show you that your command line can operate like a calculator

```
5 + 4 + 7 * 7
```

```
## [1] 58
```

or

```
pi * 7.2^2
```

```
## [1] 162.8602
```

Note that in both of these examples, the answer is printed to the screen, but not stored in memory. In other words, I cannot access that answer without redoing the calculation. If I want to store it in memory, then I assign the answer to a name. We use the symbol <- to mean "assign". In other words, the result of the computation on the right of the symbol is assigned to the name on the left of the symbol. For example:

```
x <- 4*4
```

I have now assigned the result of my computation to the name x. If I want to see this value of x in the future, I can just type it in the console.

```
x
```

```
## [1] 16
```

Note that in *RStudio* you can also see your variable in the *Environment* window.

I can also use it in future computations:

```
y<-x/2
```

x is now stored in your *Global Environment*. Think of this as your "workbench" that contains all of the data and values that you are working on. In RStudio, you can usually see what is in your *Global Environment* in the top right part of the RStudio window.

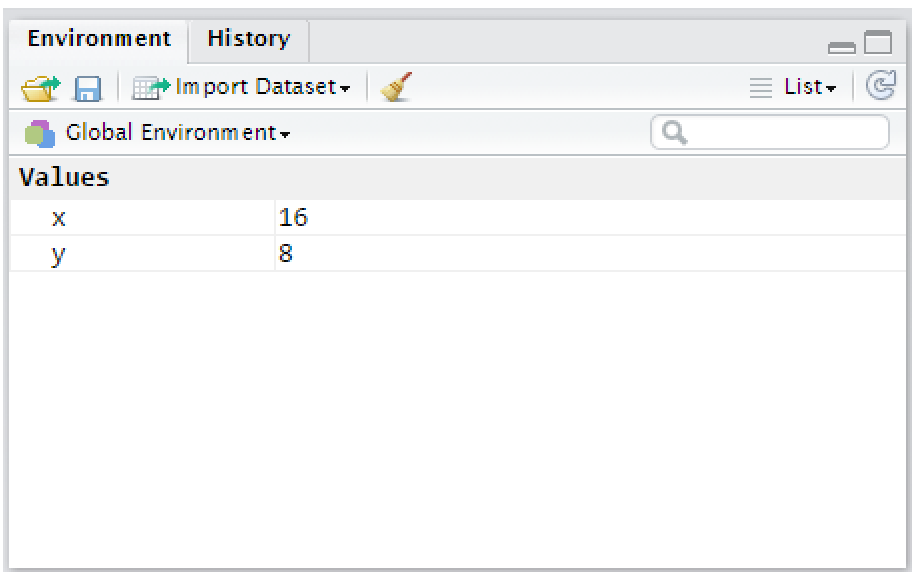


Figure 1.3: Figure 3: The "Environment" window shows the name and type of data held in memory

If you're using base R, you can list the variables that are in your *Global Environment* by typing

```
ls()
```

```
## [1] "x" "y"
```

When you close either RStudio or base R, it will ask you if you want to save your *work space*. It is essentially asking you if you want to save what is on your workbench. If you choose "yes", then it will save an *.RData file of everything that is in your workspace in your working directory. If you restart R from this working directory, it will load all of these items into your workspace. Generally it is not a good idea to save your workspace as long as you have all of the code it would take to quickly recreate all of the items in your workspace. However, if you have some code that takes along time to run, then it is best to save these items in a workspace so that you don't have to wait hours/days a second time to recreate them. For example, I created some R code to "clean" operational combat data. It took approximately 11 days to clean the data.

In this case, I would want to save my results so I don't have to wait 11 days again for this to run. In general, however, R takes seconds to run, and it is best to not save your workspace as long as you have clean and easy to run code.

1.5 Data Types

Now that we have R and RStudio installed, let's look at different classes of data. The basic building blocks are *integer*, *numeric*, *character*, *date*, *boolean* (logical) or *factor* classes of data. The first four should be self explanatory, and examples of all four are below:

```
x<-4                       #integer
x<-4.56                    #numeric
x<-TRUE                    #boolean
x<-"Rangers Lead the Way!"  #character
```

Use the class command to find out what type of data you have. Note that because we were using *x* for all three, that we were writing over the value of x. At the end of running these four lines of code, *x* would equal the last line of code: the character string "Rangers Lead the Way!"

```
class(x)
```

```
## [1] "character"
```

R does not automatically recognize *date* data. When you read *date* data into R, it is initially converted to *character* data. If you want R to recognize it as a *date*, you need to explicity change it (we will go over this in more detail later):

```
x<-"2014-01-01"
x<-as.Date(x)
class(x)
```

```
## [1] "Date"
```

There is also a type of data called *factor* data. This is categorical data (often a character string) that has a numeric value tied to it for certain types of models. Character data is often coerced to the *factor* class when you have nominal data (for example, a *gender* field that contained the strings "male" and "female"). If I change this into a factor, it will still be represented as "male" and "female", but it will also be represented numerically (as a 1 and 2). You need to be very careful when using factors, since many of the functions in R can't handle factor data. You can see the use of factor data below:

```
y<-c("male","male","female","male","female")
```

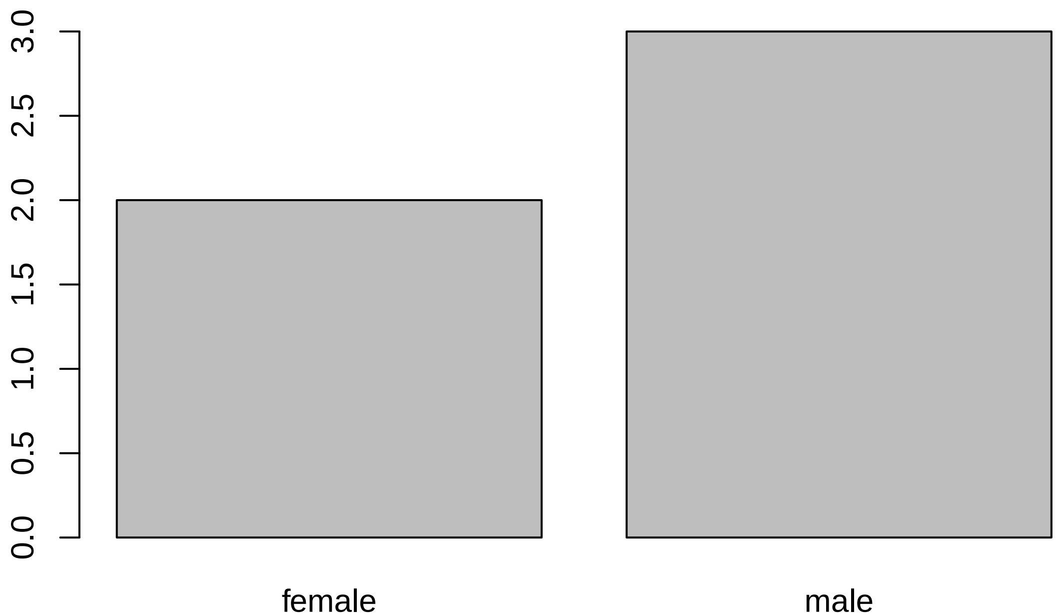
This is character data. If I tried to plot y right now, R would show an error, since you can't print *character* data. Lets convert this to a factor now:

```
y<-as.factor(y)
y
```

```
## [1] male   male   female male   female
## Levels: female male
```

Now watch when I try to plot this:

```
plot(y)
```



It plots a barchart because R recognizes this as a factor and has a numeric value associated with both of the "levels" in the factor

1.6 Data Structures

The data that we showed above is trivial (and very small) data. To work with data, we'd prefer to have it organized into a usable data structure. In this section we will introduce you to the four primary data structures that we will use:

Data Structure	Definition
Vector	Data in one dimension
Data Frame	Two dimensional data (most commonly used data structure)
List	A one dimensional data structure that can contain any class of data (objects could be other data structures)
Matrix	Multi-dimensional data of the same class

There are also different dimensions of data. So far we've been using *scalars*, in which our variable x is a single value. Data can have 1, 2, or many dimensions, however.

1.6.1 Vector Data Structure

One dimensional data that is of the same class is often organized into a *vector*. All objects in a vector must be of the same class (or will be coerced to the same class). A picture of a vector is given below

An example of a vector in R is given below:

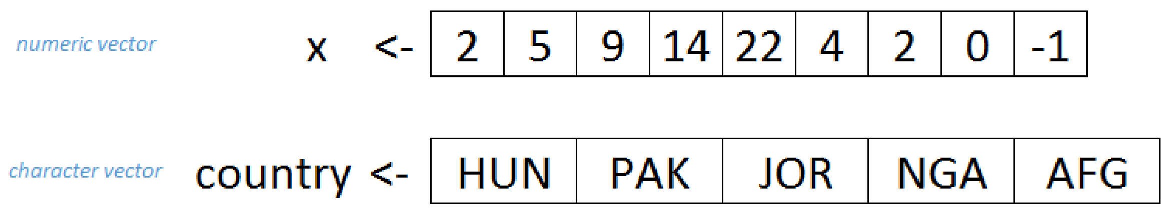


Figure 1.4: Figure 4: Understanding Vector Data in R

```
x<-c(1,6,3,9,8,2)    ## "c" means combine the values into a vector
```

If you need to create a vector of sequential integers, you can use a colon:

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

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

If you need to create a vector of the same number, you can use the repeat command:

```
rep(1,10)   # Repeat 1 ten times
```

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

1.6.2 Data Frame Data Structure

Anyone who has used Microsoft Excel is used to seeing data in the traditional two dimensional table. The data frame structures data in this way. A picture of a data frame is provided below:

Date	FirstName	LastName	Gender	PU	SU	Run
20170120	John	Smith	M	70	90	14:28
20170120	Laura	Brown	F	52	85	13:30
20170120	Jim	Wilson	M	49	60	12:36
20170120	Matthew	White	M	78	55	15:04
20170120	Heather	Farmer	F	72	76	14:01

Figure 1.5: Figure 5: Understanding Data Frame Structure in R

Each column of a data frame is a vector, and must have the same class of data. A data frame is a list of vectors where each each vector has the same length. A data frame is usually created when you read data from an external file (usually a CSV file), but you can create one manually, as seen below:

```
##Create a data frame
apft <- data.frame(Name = c("John","Laura","Jim"),
                   Gender = c("M","F","M"),
                   PU = c(70, 52, 49),
                   SU = c(90, 85, 60),
                   Run = c("14:28","13:30","12:36"))

##Print object
apft
```

```
##    Name Gender PU SU   Run
## 1  John      M 70 90 14:28
## 2 Laura      F 52 85 13:30
## 3   Jim      M 49 60 12:36
```

1.6.3 List Data Structure

A list is a linear container for objects of any class or data structure. Each object in list is separate and distinct.

A list is helpful in several situations. For example, there are many time you have vectors that do not all have the same length. For example, lets say we extracted hash-tags from Tweets at the Rio Olympics. The number of hash-tags per tweet can range from zero to seven or eight (see Figure 6 below). You can't store these vectors in a data frame because they aren't the same length. A list is the appropriate object to store these vectors in.

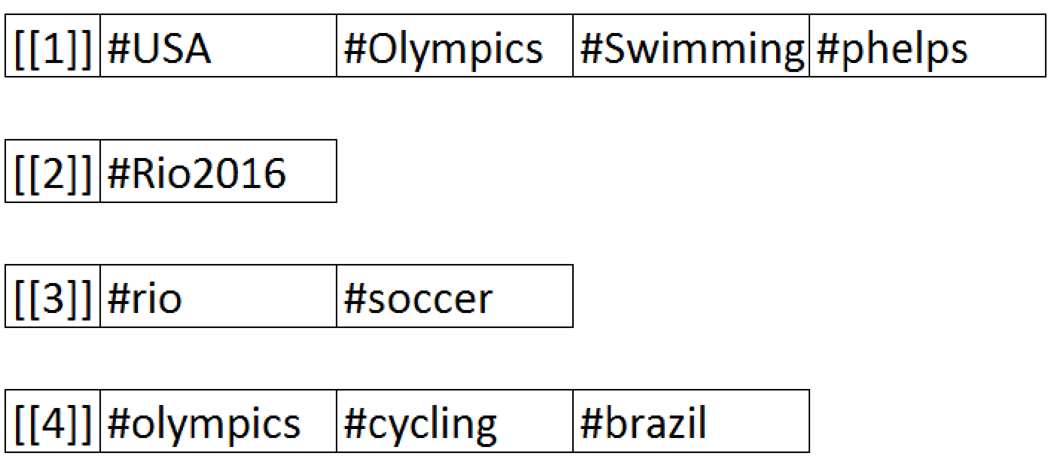


Figure 1.6: Figure 6: Understanding List Data Structure in R

A list is also helpful for storing different types of data in a single object. For example, we can store a scalar, a data frame, and a vector in a single list:

```
##Store a scalar, vector, and data frame in a list
myList <- list(y, x, apft)
```

```
##Print object
myList
```

```
## [[1]]
## [1] male   male   female male   female
## Levels: female male
##
## [[2]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[3]]
##    Name Gender PU SU   Run
## 1  John      M 70 90 14:28
## 2 Laura      F 52 85 13:30
## 3   Jim      M 49 60 12:36
```

Lists also create a great container for reading multiple data files into R and combining them into a single data frame. We will teach this technique later.

1.6.4 Matrix Data Structure

While an important data structure in R, we will not use the matrix structure often in this course. A matrix is a multi-dimensional array of numeric, boolean, or integer data (NOT character, date, or factor data).

12	17	9	18	20	14
6	3	5	19	18	2
1	18	15	5	7	7
6	9	2	3	9	3
9	6	10	13	17	12
12	8	6	16	19	17

Figure 1.7: Figure 7: Understanding Matrix Data Structure in R

Below is an example of creating a matrix object in R:

```
## Example of setting row and column names
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE)

##Print object
mdat
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
```

As mentioned above, we will not use matrices much in this course.

1.7 Input/Output Data

Now that we have all of that done, let's learn how to read and write data. To do this with some fun data, let's read in some data on movie ratings. This data contains users that rated movies in 2015. Each record (or row) represents a single user rating a single movie. Movies can have more than one rating, and users can rate more than one movie. Make sure you download the data at https://s3.amazonaws.com/dscoe-data/rating2.csv and follow along with this tutorial.

Note: if you're using a cloud environment, you can download the data by running the following command:

```
download.file("https://s3.amazonaws.com/dscoe-data/rating2.csv", destfile = "rating2.csv")
```

We use the command *read.csv* to read in data. We also make sure to assign this to an object name (in this case, the object name is `rating`)

```
rating <- read.csv("rating2.csv", as.is = TRUE)
```

The `as.is = TRUE` parameter ensures that any character data is formatted into a *character* vector rather than a *factor* vector. As a personal preference, I always explicitly convert to the *factor* data type when necessary so that I don't have any undesired consequences.

Now that we've read the file in, we'll explore this data object a little bit. Below is the top commands that I use to explore a data object.

One of the most powerful commands to explore any object is the structure command `str`. This command gives the overall size of the object (in this case it has 283,886 rows and 7 columns), as well as the class of each column vector and the first few observations from each column vector.

```
##The structure command prints the structure of the data object
str(rating)
```

```
## 'data.frame':    283886 obs. of  7 variables:
##  $ userId   : int  31 31 31 31 31 31 31 31 31 31 ...
##  $ movieId  : int  1 110 260 364 527 588 594 616 1196 1197 ...
##  $ rating   : num  3 5 5 3 0.5 3 2.5 4 5 3 ...
##  $ timestamp: chr  "2015-02-23 23:18:07" "2015-02-23 23:17:53" "2015-02-23 23:17:13" "2015-02-25 06:13:27
##  $ year     : int  2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 ...
##  $ title    : chr  "Toy Story (1995)" "Braveheart (1995)" "Star Wars: Episode IV - A New Hope (1977)" "Lion K
##  $ genres   : chr  "Adventure|Animation|Children|Comedy|Fantasy" "Action|Drama|War" "Action|Adventure|Sc
```

Related to the `str` command is the `summary` command. This command is especially helpful if you have numeric data in the object and you want to view some of the basic statistics regarding this data.

```
##The summary command prints summary statistics about an object in memory
summary(rating)
```

```
##      userId          movieId           rating      timestamp
##  Min.   :    31   Min.   :     1   Min.   :0.5   Length:283886
##  1st Qu.: 34847   1st Qu.:  2712   1st Qu.:3.0   Class :character
##  Median : 69852   Median :  8644   Median :3.5   Mode  :character
##  Mean   : 69325   Mean   : 39896   Mean   :3.5
##  3rd Qu.:104000   3rd Qu.: 79132   3rd Qu.:4.0
```

```
##  Max.   :138414   Max.   :131262   Max.   :5.0
##       year         title               genres
##  Min.   :2015   Length:283886      Length:283886
##  1st Qu.:2015   Class :character   Class :character
##  Median :2015   Mode  :character   Mode  :character
##  Mean   :2015
##  3rd Qu.:2015
##  Max.   :2015
```

I usually also use the command **head** to print the first 5 rows. This gives titles of the variables (columns) as well as a feel for the data:

```
##The head command prints the first five rows of the data set
head(rating)
```

```
##   userId movieId rating           timestamp year
## 1     31       1    3.0 2015-02-23 23:18:07 2015
## 2     31     110    5.0 2015-02-23 23:17:53 2015
## 3     31     260    5.0 2015-02-23 23:17:13 2015
## 4     31     364    3.0 2015-02-25 06:13:27 2015
## 5     31     527    0.5 2015-02-23 23:19:58 2015
## 6     31     588    3.0 2015-02-25 05:41:09 2015
##                                      title
## 1                         Toy Story (1995)
## 2                        Braveheart (1995)
## 3 Star Wars: Episode IV - A New Hope (1977)
## 4                    Lion King, The (1994)
## 5                  Schindler's List (1993)
## 6                           Aladdin (1992)
##                                           genres
## 1     Adventure|Animation|Children|Comedy|Fantasy
## 2                                Action|Drama|War
## 3                         Action|Adventure|Sci-Fi
## 4 Adventure|Animation|Children|Drama|Musical|IMAX
## 5                                       Drama|War
## 6     Adventure|Animation|Children|Comedy|Musical
```

If you only want to print the names of the columns, use the **names** command:

```
##The names command just prints the column names of a data frame
names(rating)
```

```
## [1] "userId"    "movieId"   "rating"    "timestamp" "year"      "title"
## [7] "genres"
```

Finally, if we only want the dimensions of the data, we can use **dim** to get all of the dimensions, ***nrow** to access the number of rows, and **ncol** to access the number of columns:

```
##The dim command prints the dimensions of the object
dim(rating)
```

```
## [1] 283886      7
```

```
##The nrow command prints the number of rows of a data frame
nrow(rating)
```

```
## [1] 283886
```

```
##The ncol command prints the number of columns of a data frame
ncol(rating)
```

```
## [1] 7
```

1.8 Getting Help

There's several ways to get help in R. The **help** function and the **?** function can access the documentation for packages and functions that you have loaded into R. **help.search** and the **??** function both search within documentation for loaded packages. Additionally, you can use the **args** function to print out the arguments for a function.

```
##Getting help for the str function
help(str)

##or
?str

##Searching within documentation for "subset"
help.search('subset')

##or
??subset
```

1.9 Practice Problem

Download the Korean War Casualty Data by downloading the Comma Separated Value (CSV) file here:

https://s3.amazonaws.com/dscoe-data/KoreanConflict.csv

If you're using a cloud environment, you can download the data by running the following command:

```
download.file("https://s3.amazonaws.com/dscoe-data/KoreanConflict.csv", destfile = "KoreanConflict.csv")
```

Read this into your R environment. Explore the data given the commands that we leaned this lessons. We will use this data in future lessons.

Chapter 2

Basic Data Manipulation

The most time intensive task in data science endeavors is pre-processing data. Real world data is often complex and messy. Data processing (sometimes called "munging" or "data wrangling") cleans and manipulates data so that it is in a form that is useful for models and visualizations. The R programming language is one of the best tools for manipulating data. This lesson will discuss the basics of data structure as well as ways to subset, extract and otherwise manipulate basic data.

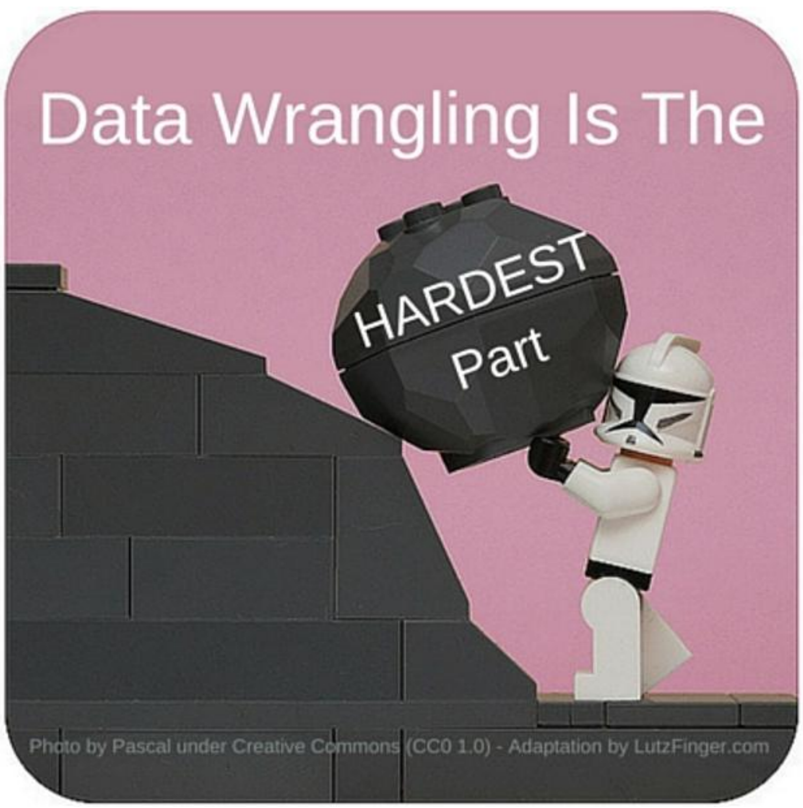


Figure 2.1: Figure 1: "Data Wrangling"" is often the most difficult part of data science

2.1 Data

For this lesson we will use casualty data from the Korean War. This data is available at Kaggle. You should have downloaded this data for the practice problem in Lesson 1. First, let's read the data into R:

```
kor <- read.csv("KoreanConflict.csv", as.is = TRUE)
```

Now let's explore the data with some of the tools we learned in Lesson 1. First, let's look at the structure of the data:

```
str(kor)  ## Print the structure of the Korean Casualty Data
```

```
## 'data.frame':    36574 obs. of  25 variables:
##  $ SERVICE_TYPE        : chr  "V" "R" "R" "V" ...
##  $ SERVICE_CODE        : chr  "L" "K" "K" "L" ...
## $ ENROLLMENT       : chr  "ACTIVE - GUARD/RESERVE" "ACTIVE - REGULAR" "ACTIVE - REGULAR" "ACTIVE - GUARD/
##  $ BRANCH              : chr  "AIR FORCE" "ARMY" "ARMY" "ARMY" ...
##  $ RANK                : chr  "CAPT" "PVT" "PFC" "2LT" ...
##  $ PAY_GRADE           : chr  "O03" "E02" "E03" "O01" ...
## $ POSITION         : chr  "" "FOOD SERVICE APPRENTICE" "HEAVY WEAPONS INFANTRYMAN" "INFANTRY UNIT COMMAND
##  $ BIRTH_YEAR          : chr  "1917" "1927" "1932" "1929" ...
##  $ SEX                 : chr  "M" "M" "M" "M" ...
##  $ HOME_CITY           : chr  "NEW YORK" "UNKNOWN" "UNKNOWN" "UNKNOWN" ...
##  $ HOME_COUNTY         : chr  "NEW YORK" "OCONEE" "BIBB" "COAHOMA" ...
##  $ NATIONALITY         : chr  "US" "US" "US" "US" ...
##  $ STATE_CODE          : chr  "NY" "GA" "GA" "MS" ...
##  $ HOME_STATE          : chr  "NEW YORK" "GEORGIA" "GEORGIA" "MISSISSIPPI" ...
##  $ MARITAL_STATUS      : chr  "MARRIED" "UNKNOWN" "UNKNOWN" "UNKNOWN" ...
##  $ ETHNICITY           : chr  "WHITE" "WHITE" "WHITE" "WHITE" ...
## $ ETHNICITY_1      : chr  "NOT SPECIFIED" "NOT SPECIFIED" "NOT SPECIFIED" "NOT SPECIFIED" ...
##  $ ETHNICITY_2         : chr  "WHITE" "WHITE" "WHITE" "WHITE" ...
## $ DIVISION         : chr  "93 BOMB SQ 19 BOMB GP" "29 RGT CMBT TEAM" "5 RGT 1 CAV DIV" "32 INF 7 DIV" ...
##  $ INCIDENT_DATE       : chr  "19510412" "19500727" "19510316" "19530122" ...
##  $ FATALITY_YEAR       : chr  "1951" "1950" "1951" "1953" ...
##  $ FATALITY_DATE       : chr  "20010402" "19500727" "19510316" "19530122" ...
##  $ HOSTILITY_CONDITIONS: chr  "H" "H" "H" "H" ...
## $ FATALITY         : chr  "DECLARED DEAD" "KILLED IN ACTION" "KILLED IN ACTION" "KILLED IN ACTION" ...
##  $ BURIAL_STATUS       : chr  "Y" "Y" "Y" "Y" ...
```

We see that this data has 36,574 rows and 25 columns. It appears that each row of the data represents an individual service member who died in the Korean War. Note that every single column is a *character* vector. This includes the rows like `BIRTH_YEAR` and `INCIDENT_DATE` that appear like they should be numeric (the fact that they are character means that at least one entry in this column has *alphabetic letters* rather than *numbers*).

2.2 Cell level data access

This data set contains two dimensions (rows and columns). To access specific rows and columns in R, we use *[row,column]* format. For example, to access the data in the first row and first column of Korea data, we would use

```
kor[1,1]  ##First row, first column
```

```
## [1] "V"
```

If we want to access the first 5 entries from the first column, we would use

```
kor[1:5,1] ##First five entries from the first column
```

```
## [1] "V" "R" "R" "V" "R"
```

Now if we want to access the first three rows from the 1st, 3rd, and 8th column, we use the following format

```
kor[1:3,c(1,3,8)]
```

```
##   SERVICE_TYPE             ENROLLMENT BIRTH_YEAR
## 1            V ACTIVE - GUARD/RESERVE       1917
## 2            R       ACTIVE - REGULAR       1927
## 3            R       ACTIVE - REGULAR       1932
```

You can also use column names (or headers) to extract data from specific columns. This is especially helpful if you can't remember respective column numbers, or if you think the column order will ever change. To extract the first three rows of data from `BRANCH`, `RANK`, and `HOME_STATE`, we can use the code below.

```
kor[1:3,c("RANK","BRANCH","HOME_STATE")]
```

```
##   RANK    BRANCH HOME_STATE
## 1 CAPT AIR FORCE   NEW YORK
## 2  PVT      ARMY    GEORGIA
## 3  PFC      ARMY    GEORGIA
```

Remember that each column represents a vector. In addition to the method we just showed, you can access data from each column vector with the following script:

```
kor$RANK[1:5]  ##Prints first five entries in RANK vector
```

```
## [1] "CAPT" "PVT"  "PFC"  "2LT"  "CPL"
```

The script above essentially says select the RANK column from the `kor` data frame, and then print to the screen the first five entries of this column.

2.3 `Table` Command (and an example of data "cleaning")

Let's explore the data a bit more. The `table` command provides a great way to see all of the possible entries in categorical data. The table command has similar functionality to *Pivot Tables* in Excel, but is much easier to use. To illustrate this command, we will table the `BIRTH_YEAR`

```
table(kor$BIRTH_YEAR) ##Table BIRTH_YEAR
```

```
## 
##      1889 1894 1895 1896 1900 1902 1903 1904 1905 1906 1907 1908 1909 1910
## 2271    1    1    1    1    5    7    2   15   14   25   22   26   48   61
## 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925
##   76  104  116  143  183  224  300  424  421  506  624  657  781  888 1107
## 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935   A2   A3   A4  ANT  ART
## 1278 1988 3621 4358 5479 5077 3630 1296  328   61    8   16    3    1   31
##  AUT  CHI  CLA  COA  COM  CON  COR  CRY  ENG  FIE  FIR  FIX  FUE  GEN  GUN
##   13   41    1    1    8    2    4    1    1    6    2    3    1   71    1
##  HEA  HIG  INT  LAN  LAU  LIG  LOW  MAJ  MAR  MIL  MIN  MOT  NON  OPE  RAD
##   37    4   17    2    1    2   25    1    1    1   20    3    5    6    1
##  RAI  SAX  SIG  SNA  STA  TAC   TE  TOP  TRA  TUB  WAR
##    2    2    2    2    2    1    2    4   31    1   14
```

The table command provides the number of records for each category. Here we learn that our data is a bit messy. Notice that although most of the entries are numerical, that there are numerous entries that don't look like a year. We can see this again if we table data by gender:

```
table(kor$SEX)  ##Table by gender
```

```
##
##                19040000                19060000
##                       2                       1
##                19070000                19080000
##                       3                       1
##                19081017                19090000
##                       1                       1
##                19100000                19110000
##                       4                       6
##                19120000                19130000
##                       1                       2
##                19130816                19140000
##                       1                       3
##                19150000                19150810
##                       7                       1
##                19160000                19170000
##                       6                       2
##                19180000                19190000
##                      11                      14
##                19190222                19200000
##                       1                       7
##                19210000                19220000
##                      13                      11
##                19230000                19240000
##                       6                      16
##                19240905                19250000
##                       1                      15
##                19250511                19250909
##                       1                       1
##                19260000                19270000
##                      20                      19
##                19280000                19280527
##                      36                       1
##                19281122                19290000
##                       1                      47
##                19290821                19291105
##                       1                       1
##                19300000                19300526
##                      41                       1
##                19300624                19310000
##                       1                      36
##                19311003                19320000
##                       1                      15
##                19320525                       F
##                       1                       2
##                       M                  MANUAL
##                   36169                       4
##                     S2)                     S3)
##                       8                      16
```

```
##                         S4)  TRACK VEHICLE (3D ECHELON)
##                           3                           1
##     WHEEL VEHICLE GASOLINE)  WHEEL YEHICLE (3D ECHELON)
##                           1                           9
```

Note that this doesn't give just *male* and *female*. For our purposes we're going to try to remove this *messy* data. Note that in some cases you will want to fix messy data, not remove it. In removing the data, I am going to assume that the same rows of data that produce errors in the GENDER field are the same rows of data that will produce errors in the `BIRTH_YEAR` data. To remove this data, we will leverage the fact that we want to keep all of the data from `BIRTH_YEAR` that is numeric, and get rid of every *row* of data that contains alphabetical *character* data. In the following code we will coerce this column into numeric data.

```
kor$BIRTH_YEAR <- as.numeric(kor$BIRTH_YEAR)
```

```
## Warning: NAs introduced by coercion
```

The `as.numeric` command coerces the data to the numeric class. Note that there is also an `as.character` and `as.factor` command that will coerce data to these respective data classes. This `as.numeric` command will create an NA value for every entry that is not numeric. It is now much easier to remove all rows that contain an NA in the `BIRTH_YEAR` column. The code below provides a way to subset the data by removing the rows that contain an NA value in the `BIRTH_YEAR` column. There are many ways to subset and cut data in R. Below we will use the bracket functionality that we discussed above. You can also use the `subset` command in the base R packages. Later in this tutorial we will use the `filter` command that comes in the `dplyr` package.

```
 kor <- kor[!is.na(kor$BIRTH_YEAR),]   ##Remove rows that contain an NA value in the BIRTH_YEAR column
```

In the code above, the `is.na` function produces a Boolean vector with TRUE values if an NA value is found. The exclamation point means NOT, and changes every TRUE to a FALSE (meaning it now produces a TRUE value if there is NOT an NA in that cell). By feeding this into our bracket functionality, we subset the data by removing all rows that contain an NA in the BIRTH_YEAR column. Now lets check the dimensions of our data:

```
dim(kor)
```

```
## [1] 33899    25
```

We now have 33,899 rows of data, meaning that we lost 2,675 rows of data. If we were conducting an in-depth study of the Korean War Casualties, we couldn't just delete this data, but would rather have to painstakingly clean it. For our purposes, we are just going to delete it.

Now let's see if that cleaned up the GENDER field. To do that, let's call on the `table` command again:

```
table(kor$SEX)
```

```
##
##     F     M
##     2 33897
```

Notice that the data is now clean, and that in our cleaned data we only have two female casualties recorded. Let's now use the table command to explore the data a bit more. Let's create a table by rank:

```
table(kor$RANK)
```

```
##
##    1LT 1STLT   2LT 2NDLT   A1C   A2C   A3C    AA    AB    AN    BG  CAPT
##    665   617   400   221    76    67    30     6     5    28     1   458
##    CDR   COL   CPL   CPO   CPT   CW2  CWO2 CWO-2    DN   ENS    FA    FN
##      8    24  6035    25   239     4     3     1     1    61    16    29
##    GEN    HA    HN  LCDR    LT   LTC LTCOL  LTJG   MAJ    MG   MSG  MSGT
```

```
##     1     2    52    12    55    24    37    79   165     1   471    68
##   PFC   PO1   PO2   PO3   PV1   PVT    SA   SFC   SGT    SN   SSG  SSGT
## 12826    44    32   119     7  6633    27  1154  2594    59     1   301
##  TSGT   WO1
##    97    18
```

From this we learn that the PFC rank sustained the highest casualty numbers, and that the highest ranking casualty was a General (assuming this means 4-star General). Now let's explore NATIONALITY. We assume that this is all US Nationality, but when we run this table command

```r
table(kor$NATIONALITY)
```

```
## 
##    CA    DA    EI    RP    UK    US 
##     6     1     1     1     1 33889
```

we find out that there are a few other nationalities represented in the data. It's interesting when we table the MARITAL_STATUS field that

```r
table(kor$MARITAL_STATUS)
```

```
## 
##      ANNULLED      DIVORCED       MARRIED NEVER MARRIED       UNKNOWN 
##             2            18          1129           993         31756 
##       WIDOWED 
##             1
```

we find out that the marital status of most of the casualties was unknown (which makes you wonder about the Defense Department data collection during the Korean War). Now let's move on to filtering (or extracting a subset) of our data.

2.4 Filter (or subset) data

Extracting a subset of data is one of the most fundamental tasks of data manipulation. There a many different ways to filter data in R. In addition to using the *bracket* functionality discussed above, you could use the `subset` command provided in Base R. Today, one of the foremost R Programming Developers (Hadley Wickam) has developed a special packages called `dplyr` [(Wickham and Francois, 2016) and `tidyr` (Wickham, 2016) just for data wrangling. For the sake of simplicity, we will attempt to primarily use these packages for data wrangling in this course.

Given a two dimensional data structure, we can think of several ways we might want to extract data. The first is to extract rows associated with a certain feature. For example, if we had some basic data from an Army Physical Fitness (APFT) test, we may want to extract rows based on GENDER, as seen below.

APFT_FEMALE <-

NAME	SEX	AGE	PU RAW	PU SCORE	SU RAW	SU SCORE	RUN TIME	RUN SCORE	SCORE TOTAL
SOLDIER2	M	23	85	100	85	100	13:10	98	298
SOLDIER3	M	22	59	82	70	87	12:40	100	269
SOLDIER4	F	24	53	75	68	84	14:44	80	239
SOLDIER5	M	24	90	100	95	100	14:10	87	287
SOLDIER6	F	22	78	100	95	100	12:43	100	300
SOLDIER7	M	24	76	100	86	100	13:22	96	296
SOLDIER8	F	21	80	100	78	100	12:28	100	300
SOLDIER9	F	22	83	100	94	100	12:45	100	300
SOLDIER10	M	22	89	100	96	100	12:27	100	300

Figure 2.2: Figure 2: Filtering Rows by Categorical Variable

If we were going to conduct this same operation (extract all FEMALE records) on our `kor` data frame with the `dplyr` package, we would execute the following command:

```
library(dplyr)
kor_female <- dplyr::filter(kor, SEX=="F")
```

This command should produce a new data frame in your environment that has has two rows and 25 columns. This new data frame only contains the two FEMALE casualties represented in the data. To explore this much smaller data set, we could now table the data frame based on state:

```
table(kor_female$HOME_STATE)
```

```
##
##          IOWA WEST VIRGINIA
##             1             1
```

and find out that one woman is from Iowa, and the other from West Virginia. If we table based on rank:

```
table(kor_female$RANK)
```

```
##
```

```
## 1STLT
##     2
```

we find out that both women were junior officers. If you take a look at the data further, you will learn that both women were in the Air Force and died in a non-hostile accident in 1952 on the same day (presumably the same accident).

Note that we can also filter rows based on a Boolean function. For example, if we wanted to only look at casualties that were over 30 years old in 1950, we could filter with the following `dplyr` command:

```
kor_Over30 <- filter(kor, BIRTH_YEAR < 1920)  ##Filter those older than 30 in 1950
```

When you run this command, you will find that our cleaned data produces 2220 records of casualties that were over 30 in the year 1950. If we wanted to only select those individuals that were in their 30's in 1950, we would use the following `dplyr` command:

```
kor_30s <- filter(kor, BIRTH_YEAR < 1920 & BIRTH_YEAR > 1910)
```

Running this command we find that 1,991 of the casualties were in their 30's in 1950.

Now that we've filtered by row, let's show how to filter by column. We've already demonstrated above how to do this with the bracket notation, now we will illustrate how to do this using the `dplyr` package. We often find that we've loaded data that has many columns that we're not interested in. In these cases, it is often helpful to extract the columns that we're interested in. This will also shrink the size of our data in memory, and make our code run faster. In the picture below, we illustrate this with some simple APFT data (in this case we're extracting the demographic and raw score columns):

APFT_RAW <-

NAME	SEX	AGE	PU RAW	PU SCORE	SU RAW	SU SCORE	RUN TIME	RUN SCORE	SCORE TOTAL
SOLDIER2	M	23	85	100	85	100	13:10	98	298
SOLDIER3	M	22	59	82	70	87	12:40	100	269
SOLDIER4	F	24	53	75	68	84	14:44	80	239
SOLDIER5	M	24	90	100	95	100	14:10	87	287
SOLDIER6	F	22	78	100	95	100	12:43	100	300
SOLDIER7	M	24	76	100	86	100	13:22	96	296
SOLDIER8	F	21	80	100	78	100	12:28	100	300
SOLDIER9	F	22	83	100	94	100	12:45	100	300
SOLDIER10	M	22	89	100	96	100	12:27	100	300

Figure 2.3: Figure 3: Filtering Specific Columns (or fields)

Let's say we were studying the Korean Casualty data to understand the time factor of those who died of wounds, and were particularly interested in the time between `INCIDENT_DATE` and `FATALITY_DATE`. Below we'll extract these two columns with the `dplyr` package:

```
kor_dates <- select(kor, one_of(c("INCIDENT_DATE","FATALITY_DATE"))) #Select two columns
```

Now if we look at the structure of this new data frame:

```
str(kor_dates)
```

```
## 'data.frame':    33899 obs. of  2 variables:
##  $ INCIDENT_DATE: chr  "19510412" "19500727" "19510316" "19530122" ...
##  $ FATALITY_DATE: chr  "20010402" "19500727" "19510316" "19530122" ...
```

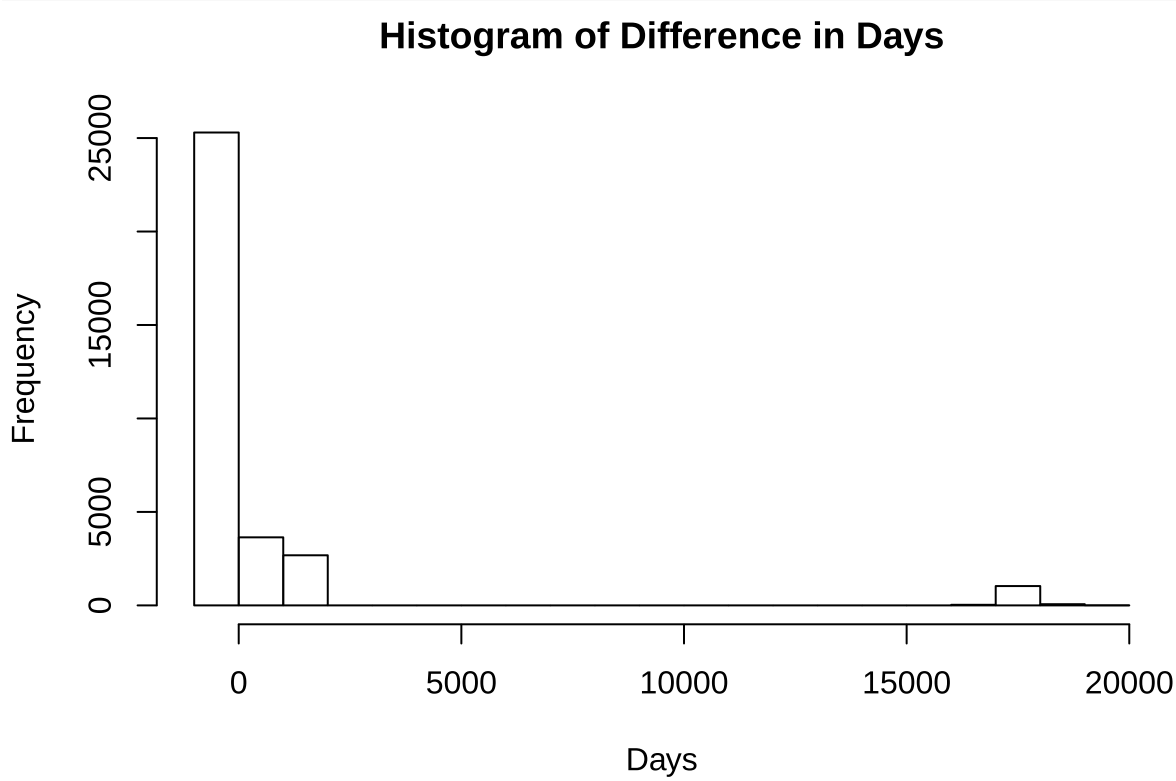
We see that we only have two columns, but still have all 33,899 rows. The code below is beyond the extent of this lesson on filtering (it contains some code we'll go over in Lesson 5) but is interesting to look at the difference between incident date and fatality date. In this code we will load the `lubridate` package (Grolemund et al., 2016) (another package written by Hadley Wickam) and use it to convert these two columns to date format and calculate the difference between them (i.e. the number of days between the incident that the death of the Service Member).

```
library(lubridate)
days <- ymd(kor_dates$FATALITY_DATE) - ymd(kor_dates$INCIDENT_DATE)
days[1:100]
```

```
## Time differences in days
##   [1] 18253     0     0     0     0     0     0     0     0     0 17697
##  [12]     0     0     0 18342    31     0     0     0  1153     0     3
##  [23]     0     0   355     0     0     0     0     0     0     0    53
##  [34]     0     0     0     0     0     0     0     0    NA     0     0
##  [45]    NA   469     0     0    NA     0     0   981     0     0     0
##  [56]     0     0  1131  1155    46     0     0     0     0     0    NA
##  [67]     0    NA     0  1125   959     0   469     0     0     0   969
##  [78]     0     0    31     0  1130     0     0     0 17568     0  1155
##  [89]   120     0   177     0   154     0     0     7     0     0    NA
## [100]     0
```

Looking at the first few entries makes us wonder. The very first entry had 18,253 days between the incident and the fatality. In fact, if you look closer at the dates, you will see that this Service Member had an incident on 12 April 1951, but wasn't considered a fatality until 2 April 2001. In fact, if we quickly plot a histogram of the difference in days (you'll learn this command next lesson):

```
#plot histogram of difference in days
hist(as.numeric(days), main="Histogram of Difference in Days", xlab="Days")
```



Here we see that there's a number of casualties that seem to have a fatality day around the year 2000. If you look at the original data will see that the first Service Member in the data (an Air Force Captain) is listed with an incident year of 1951 and `FATALITY_DATE` in 2001. Notice that the FATALITY status is *DECLARED*

DEAD. This officer, as part of a bombing group, must have had an MIA status for several decades until finally "declared dead" in 2001. The "declared dead" date became his fatality date, which means it would be difficult to evaluate the temporal aspect of wound care with this data.

2.5 Using the grep and aggregate commands

The following video illustrates how to use the grep and aggregate commands. This video will use movie rating data that you downloaded in Lesson 1.

```
download.file("https://s3.amazonaws.com/dscoe-data/rating2.csv", destfile = "rating2.csv")
```

2.6 Summary

What we have seen is that R produces a great platform to rapidly "wrangle" and explore data.

2.7 Practice Problem

1. Use grep to determine how many casualties had *INFANTRY somewhere in their title (use the `POSITION` field).

Chapter 3

Basic Visualization

The R Programming language has some of the most powerful data visualization packages available. These packages are continually expanded upon, with new data visualizations packages being created on a regular basis. In addition to packages that create your basic statistical visualization (line plots, bar plots, pie plots, etc) there are packages that create geospatial visualizations, 3D visualizations, as well as interactive visualizations.

Lets start by reading in the Korean Conflict data and perform the primary cleaning functions that we performed last lesson.

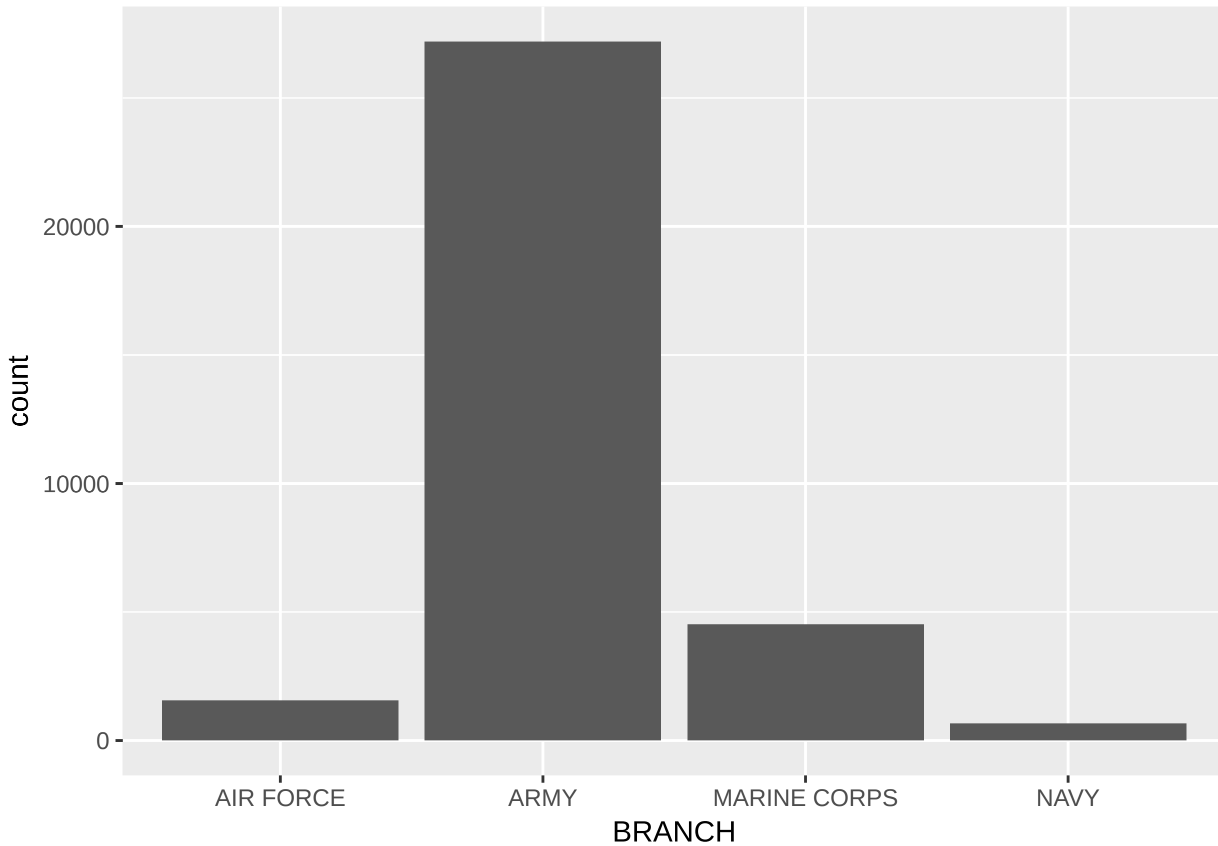
```
kor <- read.csv('KoreanConflict.csv', as.is=TRUE)
kor$BIRTH_YEAR <- as.numeric(kor$BIRTH_YEAR)
kor <- kor[!is.na(kor$BIRTH_YEAR),]
```

Now that we have the data in memory, we will use some basic visualizations to explore the data. In this lesson, we will primarily use visualizations from the ggplot2 package (Wickham and Chang, 2016). If you haven't installed this package yet, run the command `install.packages('ggplot2')`.

3.1 Bar Plot

We will start by producing a basic barplot of categorical variables. First we'll look at the `BRANCH` field for the Korean Casualties.

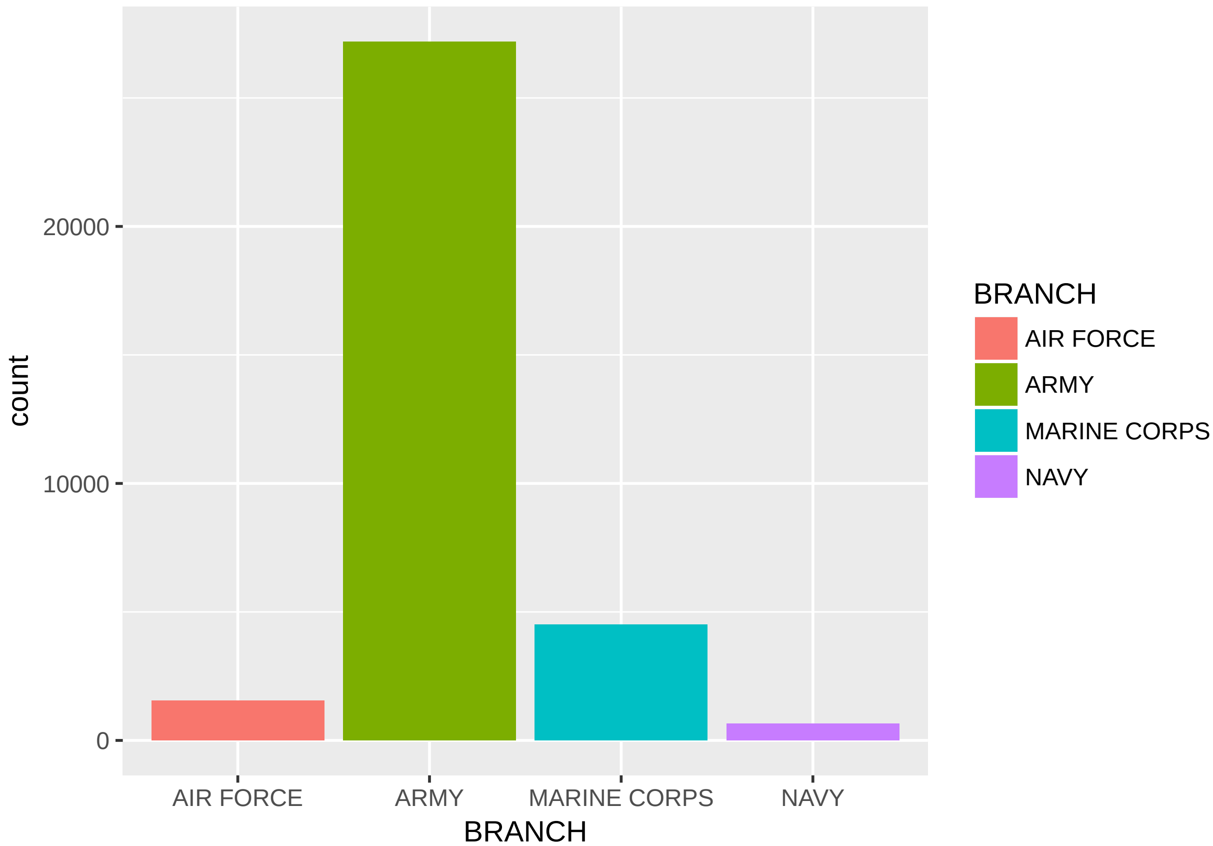
```
library(ggplot2)
ggplot(kor, aes(BRANCH)) + geom_bar()
```



At first this command isn't very intuitive. The `ggplot` command is used with all visualizations. In this command it says that we will create a visualization of the `kor` data set, and particularly look at the `BRANCH` variable. The next command says to take this specific field and create a barplot (we could create other plots with this data as well.)

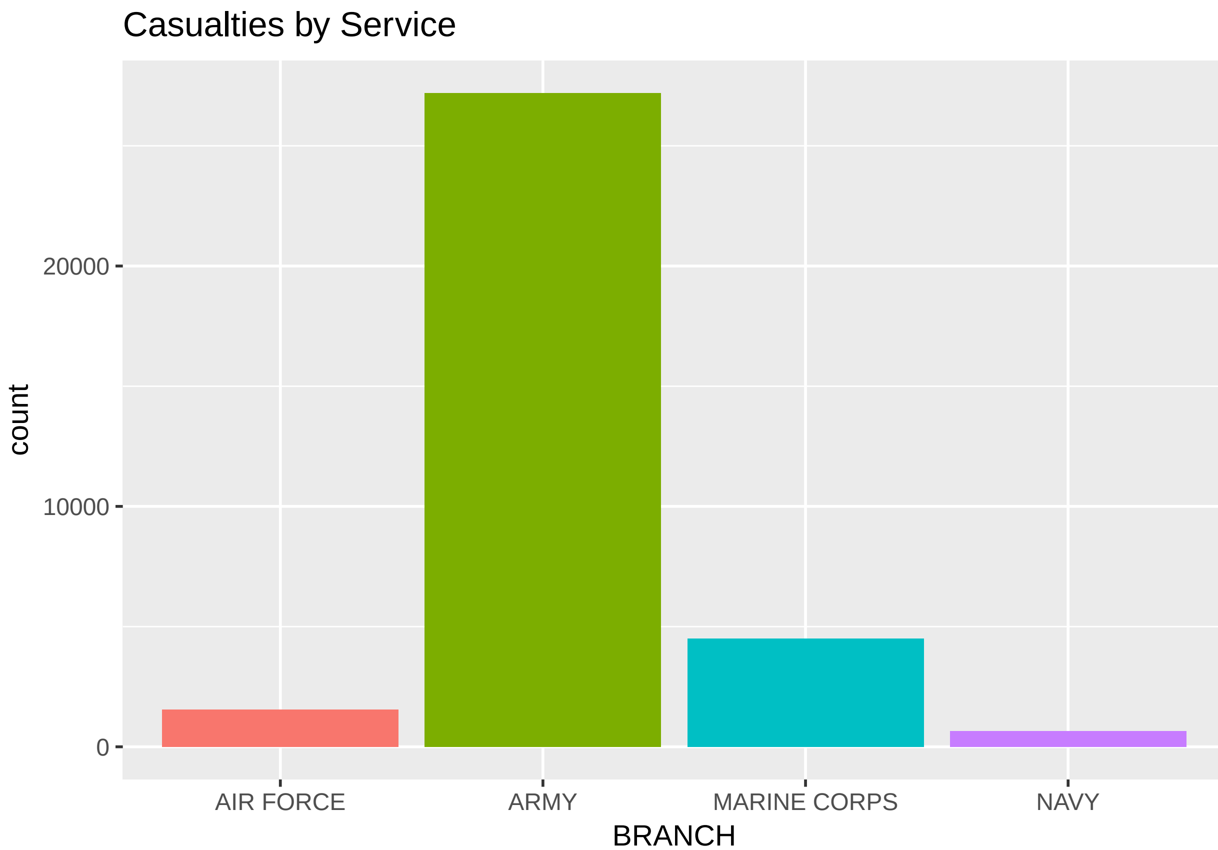
If we wanted to improve the color scheme, we could add `fill = BRANCH`.

```
ggplot(kor, aes(BRANCH, fill = BRANCH)) + geom_bar()
```



Now, let's add a title. Additionally, we will get rid of the legend, since it the labels are already on the axis.

```
ggplot(kor, aes(BRANCH, fill = BRANCH)) + geom_bar() +
  ggtitle("Casualties by Service")+ theme(legend.position="none")
```



What if we wanted to stacked barplot? Say we wanted to see how the distribution of ETHNICITY in the service BRANCHES. First, let's take a look at our categories:

```
table(kor$ETHNICITY_2)
```

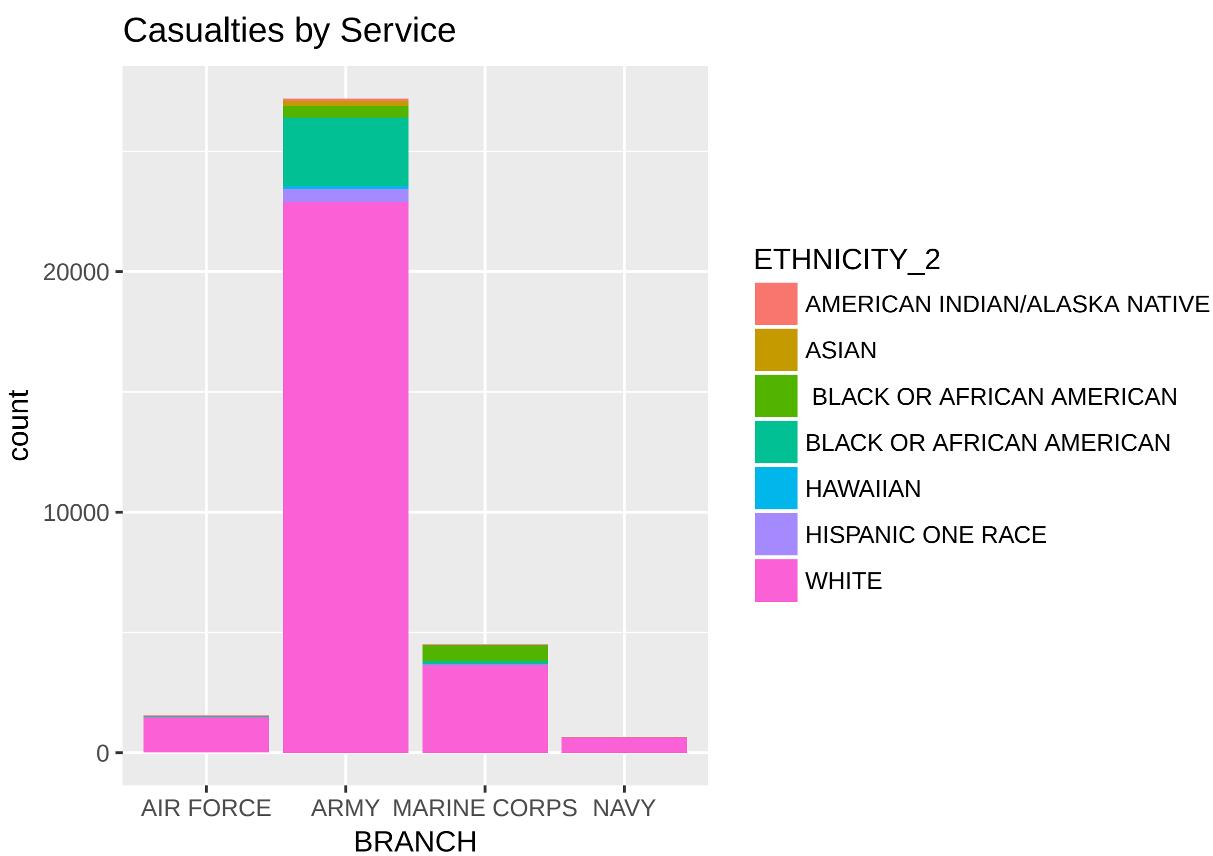
```
##
##             AMERICAN INDIAN/ALASKA NATIVE
##                                       103
##                                     ASIAN
##                                       229
##                 BLACK OR AFRICAN AMERICAN
##                                      1146
##                 BLACK OR AFRICAN AMERICAN
##                                      3022
##                         HISPANIC ONE RACE
##                                       566
## NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER
##                                       142
##                                     WHITE
##                                     28691
```

These are rather long names to display on a chart. We will start by creating shorter names. We can do this in the code below using the "grep command that we learned in Lesson 2:

```
kor$ETHNICITY_2[grep("HAWAIIAN",kor$ETHNICITY_2)] <- "HAWAIIAN"
```

Now all we have to do to change in our previous code is to change the `fill = BRANCH` to `fill = ETHNICITY_2`:

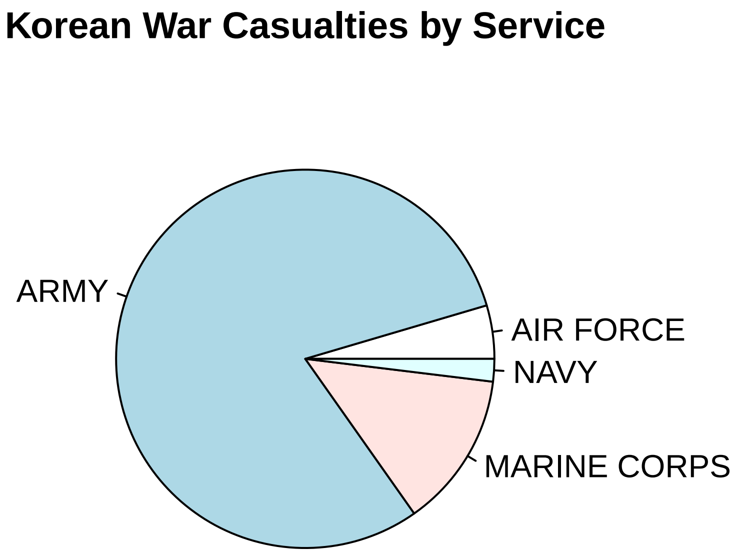
```r
ggplot(kor, aes(BRANCH, fill = ETHNICITY_2)) + geom_bar() +
  ggtitle("Casualties by Service")
```



3.2 Pie Plot

Statisticians will generally tell you that you should never use a Pie Plot (usually a bar plot is recommended because it is easier for the human eye to distinguish differences in magnitude). That being said, there are still a few occasional times when a pie plot is necessary. For this plot, we are going to use a function from the BASE graphics package (this comes with R and you don't have to load it). The pie plot is very easy to produce if we wrap the `pie()` command around the `table` command:

```r
pie(table(kor$BRANCH), main = "Korean War Casualties by Service")
```



3.3 Histogram Plot

Now we'll take a look at several ways to create *histograms* in R. This is where R will quickly outshine Microsoft Excel and other spreadsheet programs. While Excel can create a barplot just like R, it is extremely time consuming to create a *histogram* in Excel, whereas R can create one in one line code.

Let's create a histogram of the age or each Korean Casualty. Notice that we don't have a field that has *age* in it, but we do have the `BIRTH_YEAR` and `FATALITY_YEAR`. The calculation below will create a new field that is the `AGE` of the casualty at death (notice that we remove all records with a `FATALITY_DATE` after 1960 in order to remove those who were MIA and declared dead at a later time).

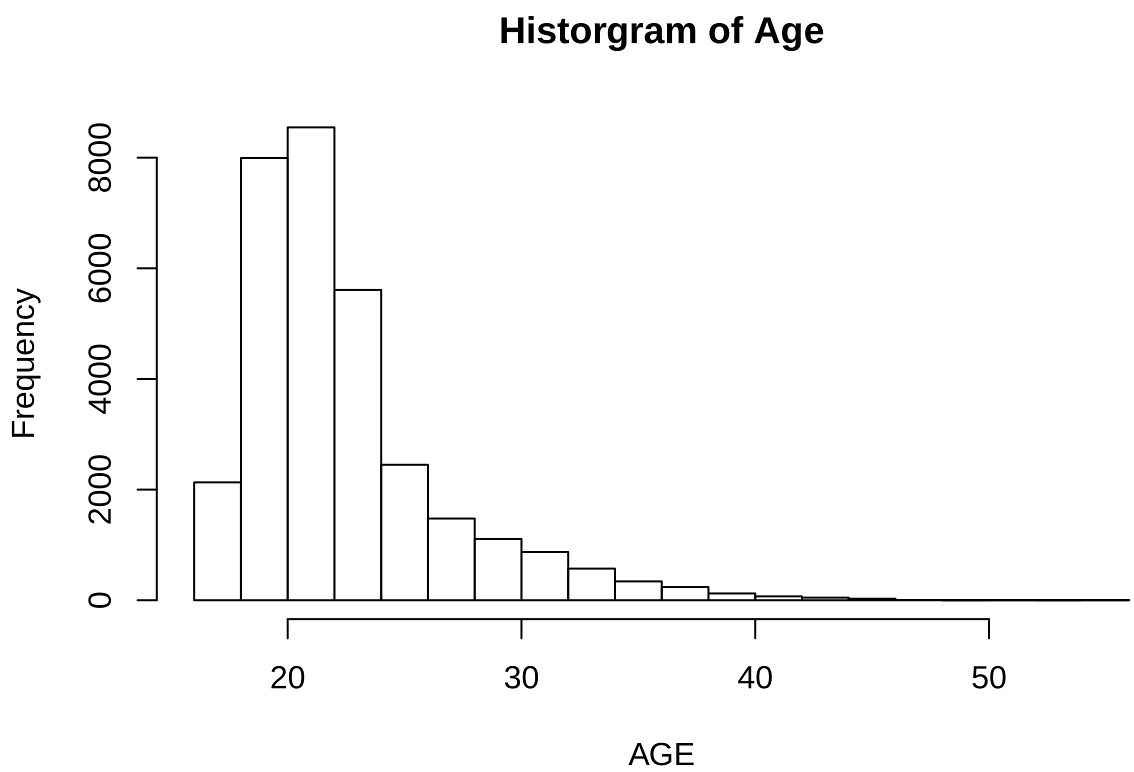
```r
# Filter our MIA
kor2 <- dplyr::filter(kor, FATALITY_DATE < 1960)

# Coerce to numeric
kor2$FATALITY_YEAR <- as.numeric(kor2$FATALITY_YEAR)
kor2$BIRTH_YEAR <- as.numeric(kor2$BIRTH_YEAR)

# Create AGE field
kor2$AGE <- kor2$FATALITY_YEAR - kor2$BIRTH_YEAR
```

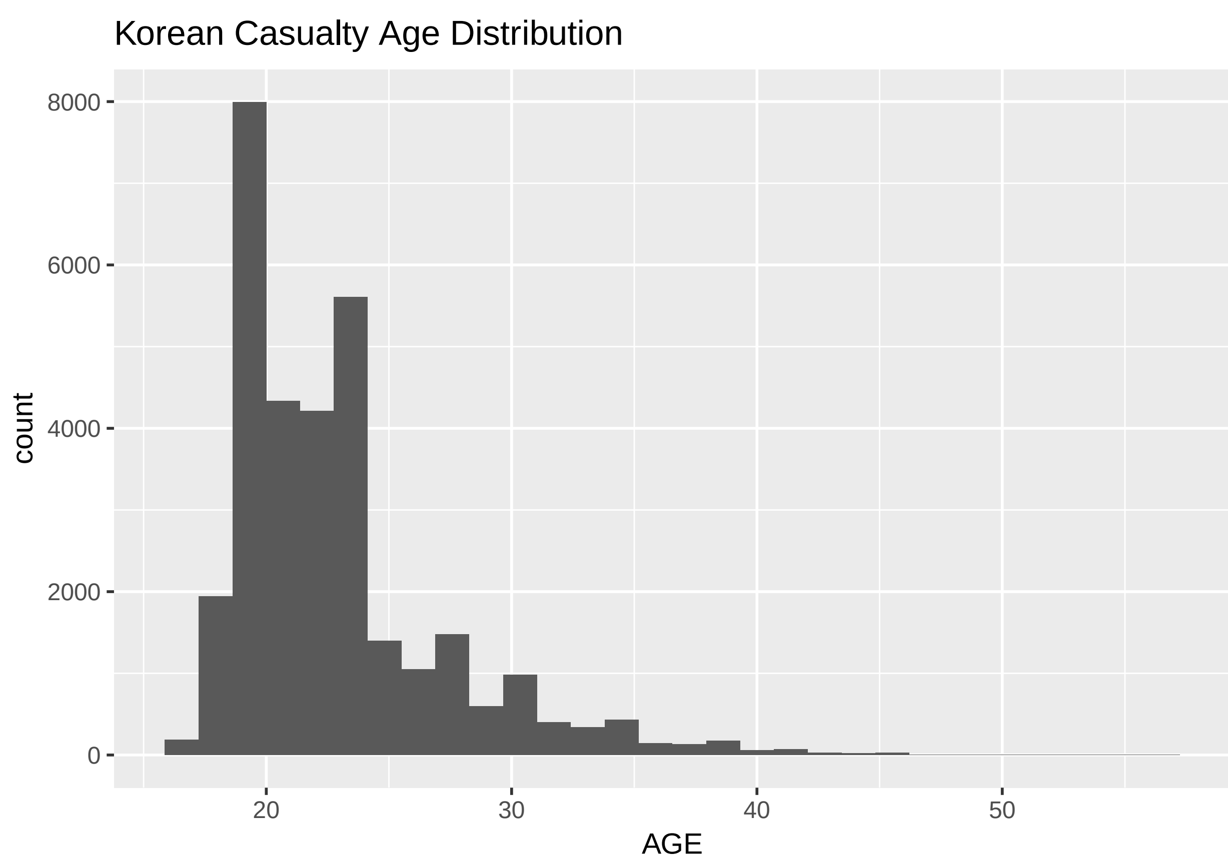
Now that we've created the `AGE` field, we will use two different techniques to create a histogram. The Base R package has a histogram function that is easy to use and helpful for exploring data.

```r
hist(kor2$AGE, freq=TRUE, main = "Historgram of Age", xlab = "AGE")
```



The ggplot2 package also has the ability to generate a histogram that is generally better for presentations.

```
ggplot(kor2, aes(AGE)) + geom_histogram() + ggtitle('Korean Casualty Age Distribution')
```



3.4 Time Series Plot

Time series plots (and line plots in general) are helpful in visually identifying trends and anomalies in data. We are going to change our data sets to look at the Olympic data. This data set is the data on all Olympic metals for the Summer Olympics from 1896 to 2012. This data set is available here:

https://s3.amazonaws.com/dscoe-data/summer.csv

You can download the file with the following command:

```
download.file("https://s3.amazonaws.com/dscoe-data/summer.csv", destfile = "summer.csv")
```

Now that you've acquired the data, read it into R and take a look at its structure with the following two commands:

```
oly <- read.csv('summer.csv', as.is = TRUE)
str(oly)
```

```
## 'data.frame':    31165 obs. of  9 variables:
##  $ Year      : int  1896 1896 1896 1896 1896 1896 1896 1896 1896 1896 ...
##  $ City      : chr  "Athens" "Athens" "Athens" "Athens" ...
##  $ Sport     : chr  "Aquatics" "Aquatics" "Aquatics" "Aquatics" ...
##  $ Discipline: chr  "Swimming" "Swimming" "Swimming" "Swimming" ...
## $ Athlete  : chr "HAJOS, Alfred" "HERSCHMANN, Otto" "DRIVAS, Dimitrios" "MALOKINIS, Ioannis" ...
##  $ Country   : chr  "HUN" "AUT" "GRE" "GRE" ...
##  $ Gender    : chr  "Men" "Men" "Men" "Men" ...
## $ Event    : chr  "100M Freestyle" "100M Freestyle" "100M Freestyle For Sailors" "100M Freestyle For Sailo
##  $ Medal     : chr  "Gold" "Silver" "Bronze" "Gold" ...
```

Let's say that we want to explore the trend of increased numbers of women participating in the Summer Olympics over the past century. To do this we could start by aggregating the number of Olympic medals by `Year` and `Gender`:

```
oly_sum <- aggregate(Athlete ~ Year + Gender, data=oly, length)
```

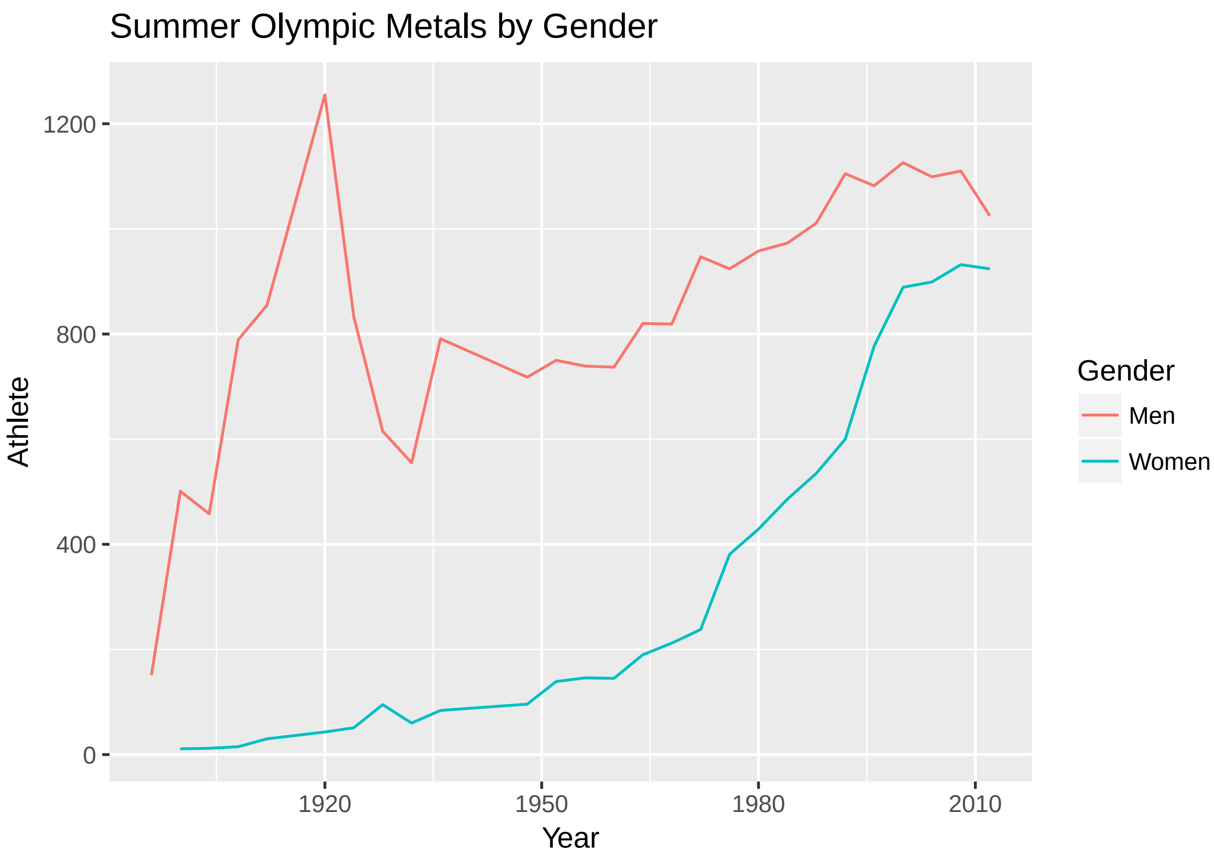
Let's take a look at the first few lines of this new data set to make sure that it aggregated the data as we anticipated:

```
head(oly_sum)
```

```
##   Year Gender Athlete
## 1 1896    Men     151
## 2 1900    Men     501
## 3 1904    Men     458
## 4 1908    Men     789
## 5 1912    Men     855
## 6 1920    Men    1255
```

Everything looks good. Now we have a data set that we can use to create a time series line plot. We create this plot below:

```
ggplot(oly_sum, aes(Year, Athlete, group = Gender, color = Gender)) + geom_line() + ggtitle('Summer Olym
```



From this graph we see that there was significant growth in the participation of women in the Olympics starting in the 1970's (we also see an interesting spike in the number of medals for men in the 1920's that we could explore if desired).

Now let's select a few of the prominent countries in the Summer Olympics (we'll look at the permanent members of the UN Security Council: United States, Great Britain, France, China, and Russia). To start this let's first look at the table of countries listed in the data set.

```
table(oly$Country)
```

```
##
##        AFG  AHO  ALG  ANZ  ARG  ARM  AUS  AUT  AZE  BAH  BAR  BDI  BEL  BER
##    4     2    1   15   29  259   11 1189  146   26   27    1    1  411    1
##  BLR  BOH  BOT  BRA  BRN  BUL  BWI  CAN  CHI  CHN  CIV  CMR  COL  CRC  CRO
##  113    7    1  431    1  333    5  649   33  807    1   23   19    4  114
##  CUB  CYP  CZE  DEN  DJI  DOM  ECU  EGY  ERI  ESP  EST  ETH  EUA  EUN  FIN
##  410    1   56  507    1    6    2   28    1  442   39   45  260  223  456
##  FRA  FRG  GAB  GBR  GDR  GEO  GER  GHA  GRE  GRN  GUA  GUY  HAI  HKG  HUN
## 1396  490    1 1720  825   25 1305   16  148    1    1    1    8    4 1079
##  INA  IND  IOP  IRI  IRL  IRQ  ISL  ISR  ISV  ITA  JAM  JPN  KAZ  KEN  KGZ
##   38  184    3   61   30    1   17    7    1 1296  127  788   49   93    3
##  KOR  KSA  KUW  LAT  LIB  LTU  LUX  MAR  MAS  MDA  MEX  MGL  MKD  MNE  MOZ
##  529    6    2   20    4   55    2   22    8    6  106   24    1   14    2
##  MRI  NAM  NED  NGR  NIG  NOR  NZL  PAK  PAN  PAR  PER  PHI  POL  POR  PRK
##    1    4  851   84    1  554  190  121    3   17   15    9  511   33   58
##  PUR  QAT  ROU  RSA  RU1  RUS  SCG  SEN  SGP  SIN  SLO  SRB  SRI  SUD  SUI
```

```
##    8    4  640  106   17  768   14    1    4    4   26   31    2    1  380
##  SUR  SVK  SWE  SYR  TAN  TCH  TGA  THA  TJK  TOG  TPE  TRI  TTO  TUN  TUR
##    2   34 1044    3    2  329    1   25    3    1   44   20   10   10   86
##  UAE  UGA  UKR  URS  URU  USA  UZB  VEN  VIE  YUG  ZAM  ZIM  ZZX
##    1    7  173 2049   76 4585   20   12    2  435    2   23   48
```

Studying this data a bit, we see that the ISO-3 code for Russia changed from *URS* to *RUS* after the fall of the Soviet Union. For our analysis, we will change all *USR* data to *RUS*.

```
oly$Country[oly$Country=="URS"] <- "RUS"
```

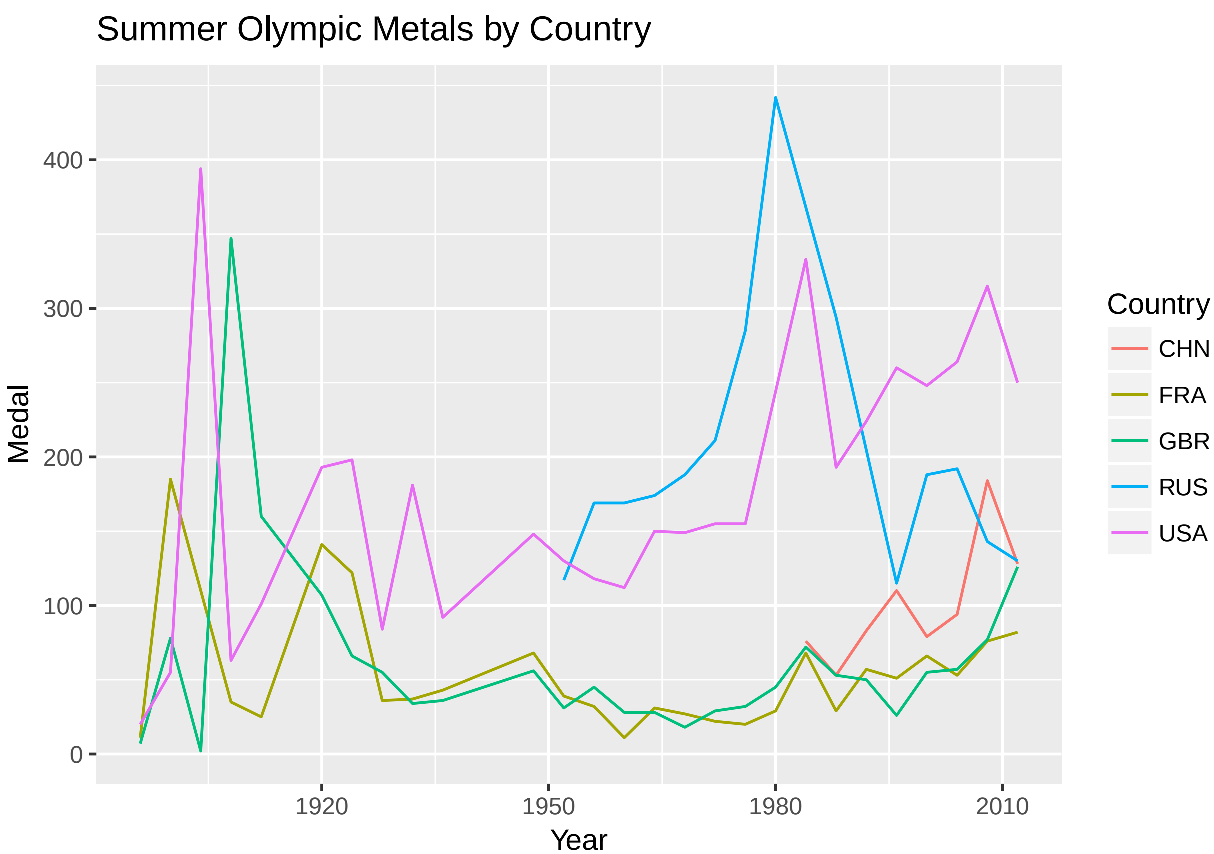
Now will will aggregate the data by `Year` and `Country` as well as filter out the counties we are interested in (notice how we create a vector of our five countries and then use the `%in%` function to filter out multiple countries):

```
library(dplyr)
oly_country <- aggregate(Medal ~ Country + Year, data = oly, length)

countries <- c('USA','GBR','FRA','CHN','RUS')
oly_country2 <- filter(oly_country, Country %in% countries)
```

Having completed this, we will now plot a time series plot by country for the last century.

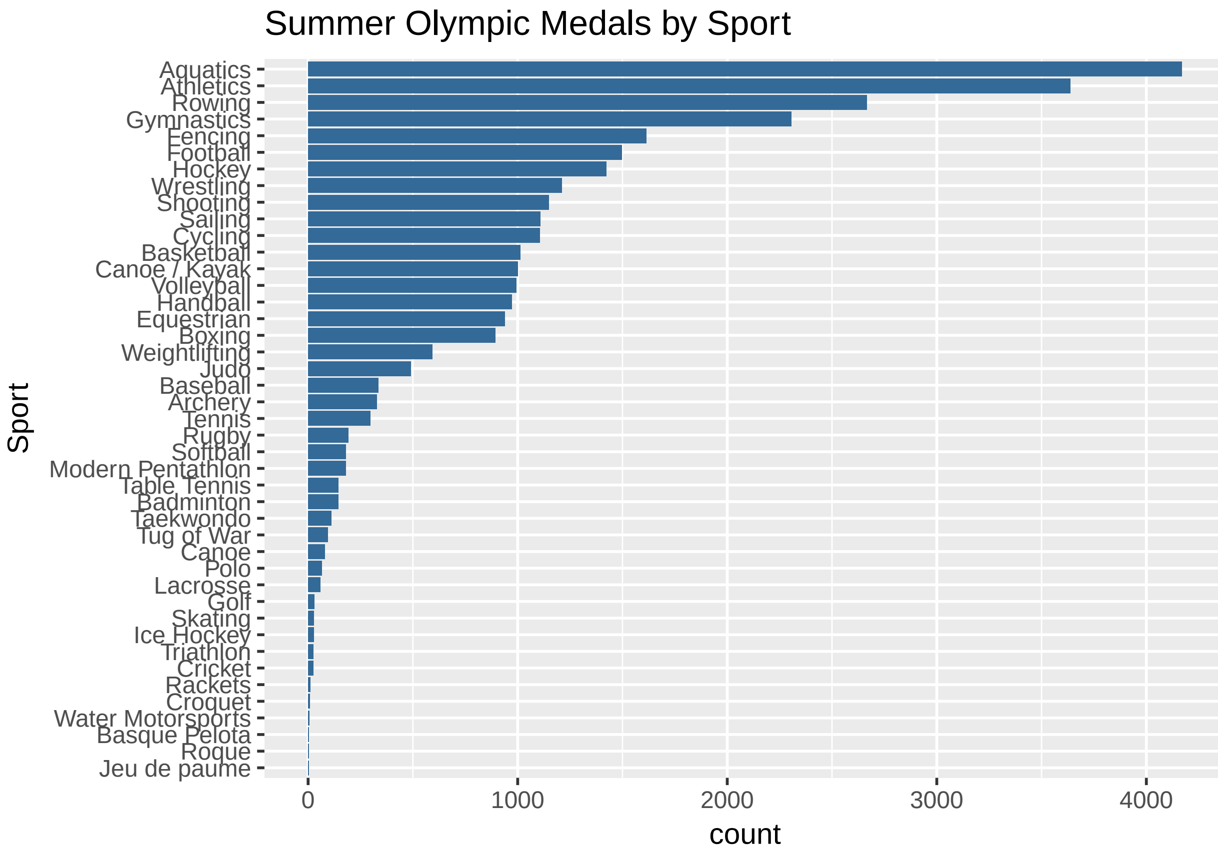
```
ggplot(oly_country2, aes(Year,Medal,group = Country, color = Country)) + geom_line() + ggtitle('Summer (
```



This plot highlights a bit of the history of the summer Olympics. Note that Russia began competing in the 1950's and China didn't begin competing until the mid-1980's. While the US has the longest sustained volume, Russia has the highest number in a single year (1980). While Great Britain had a surge at the turn of the century, it has decreased to ~50 medals in the 1920's, and stayed near this mark for most the century (as has France).

3.5 Practice Problem

Using the Olympic Data and Google, try to recreate the plot below with a horizontal barplot and bars ordered by volume of medals.



Chapter 4

Introduction to Control Structures

This lesson will cover the `if-then` statement as well as the `for` loop and `while` loop. These are two very common control structures for all computer programming languages, and are used extensively in the R Programming Language.

4.1 If - else Statements

The `if-then` statement allows us to automate decision points and guide the computer through a data flow diagram. The basic syntax is given below:

```
if(<condition>) {
  ## do something
} else if{
  ## do something else
} else {
  ## do something completely different
}
```

The else clause is not always necessary, and many times we just need an if statement:

```
if(<condition>) {
  ## do something
}
```

The `if-else` statement is most often used in *loops* and *functions*. We'll illustrate the use of the `if-then` statement in *loops* below.

4.2 Loops

Loops provide a way to systematically walk down a data structure (usually a vector, data frame, or list) and accomplish a task. The `for` loop and the `while` loop will be the primary loops for this class. The `for` loop is used when we know ahead of time a finite number of iterations that we need to execute (for example, we execute a task for every object in a vector/list or every row in a data frame).

The `for` loop below iterates over the values 1, 2, 3, 4, and 5 and prints each of the values.

```
for(i in 1:5){
  print(i)
}
```

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

Notice that we can also use i to access a value in a vector, a row in a data frame, or an object in a list.

```
letters <- c("a", "b", "c", "d", "e", "f")

for(i in 1:4){
  print(letters[i])
}
```

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

You also don't have to start with 1, or use the letter *i*:

```
for (year in 2010:2015){
  print(paste("The year is", year))
}
```

```
## [1] "The year is 2010"
## [1] "The year is 2011"
## [1] "The year is 2012"
## [1] "The year is 2013"
## [1] "The year is 2014"
## [1] "The year is 2015"
```

The `next` command is often used to skip an iteration if a certain condition is met. The code below illustrates how to use the `next`

```
for(i in 1:5){
  if(letters[i]=="c"){
    next
  }
  print(letters[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "d"
## [1] "e"
```

Loops can also be nested inside of each other. This is useful when working with multiple dimensions (like matrices) or subsets of subsets (for example, the outer `for` loop iterates over countries, the inner `for` loop iterates over each city in a given country. An example of a nested `for` loop is given below, creating a multiplication table:

```
# nested for: multiplication table
mymat = matrix(nrow=10, ncol=10) # create a 30 x 30 matrix (of 30 rows and 30 columns)
```

```
for(i in 1:nrow(mymat)) {  # for each row
  for(j in 1:ncol(mymat)){ # for each column
    mymat[i,j] = i*j     # assign values based on position: product of two indexes
  }
}

mymat
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    2    3    4    5    6    7    8    9    10
##  [2,]    2    4    6    8   10   12   14   16   18    20
##  [3,]    3    6    9   12   15   18   21   24   27    30
##  [4,]    4    8   12   16   20   24   28   32   36    40
##  [5,]    5   10   15   20   25   30   35   40   45    50
##  [6,]    6   12   18   24   30   36   42   48   54    60
##  [7,]    7   14   21   28   35   42   49   56   63    70
##  [8,]    8   16   24   32   40   48   56   64   72    80
##  [9,]    9   18   27   36   45   54   63   72   81    90
## [10,]   10   20   30   40   50   60   70   80   90   100
```

The `while` loop is used when we don't know how many iterations we need to go through, but we know the condition that needs to be met before we are done.

```
i <- 5
while(i <= 25) {
  print(i)
  i <- i + 5
}
```

```
## [1] 5
## [1] 10
## [1] 15
## [1] 20
## [1] 25
```

To finish out this lesson, we'll provide an example below of a some more sophisticated code. This function simulates a round of play in the board game *RISK*. In the board game *RISK*, and attacker begins an assault against a defender, and the win/loss is adjudicated as both players begin rolling dice and comparing values. The simulation below plays through this entire series, declares whether the attacker or the defender won, and declares the number of armies left on the board for each player. Instead of doing this once, this code plays through this 10,000 times, and in the process calculates the probability of the *attacher* winning. This type of process is known as Monte Carlo Simulation. While you may not fully understand all aspects of this code, note throughout this function how important *loops* and `if-else` statements are:

```
risk <- function(attacker, defender, n=10000) {
  results <- rep(NA, n)
for(j in 1:n){
while(attacker > 1 & defender > 0) {
 atk.dice <- min(attacker-1, 3)
 def.dice <- min(defender, 2)
 atk.roll <- ceiling(runif(atk.dice)*6)
 def.roll <- ceiling(runif(def.dice)*6)
 atk.roll <- atk.roll[order(atk.roll,decreasing=T)]
 def.roll <- def.roll[order(def.roll,decreasing=T)]
 comparison <- min(atk.dice, def.dice)
 for (i in 1:comparison) {
```

```
  if (atk.roll[i] > def.roll[i]) defender <- defender-1
  if (atk.roll[i] <= def.roll[i]) attacker <- attacker-1
 }
}
if (defender==0) results[j] <- "Attacker"
if (defender>0) results[j] <- "Defender"
}

print(paste("The Probability of the Attacker winning is: ",length(results[results=="Attacker"])/n))
}
```

Now will illustrate how this function is used:

```
risk(attacker=12,defender=6)
```

```
## [1] "The Probability of the Attacker winning is:  1"
```

4.3 Practice Problem

Use the instructions below to determine and visualize the average rating by movie genres

- Read the Movie Ratings Data into R
- Use the code below to build a vector called `genres` that contains all unique genre names contained in the data set.

```
library(stringr)
genres <- unique(unlist(str_split(rating$genres,"\\|")))
```

- Create a vector object that is the same length as genres to contain the `mean` ratings
- Iterate over your original data set in a `for` loop, use the `grep` command to extract the subset that is related to a specific genre, and calculate the mean rating for that genre
- Create a barplot visualization of average ratings by genre

Chapter 5

Introduction to Dates in R

We often see dates and times in data. Often each record (or row) of data is connected to at least one date or time. Similar to Microsoft Excel, R has a special class or format that it uses to work with dates.

5.1 Dates with Base R

We will start by showing a few of the date commands that are built into the Base R package (later on we will take a look at the *lubridate* package (Grolemund et al., 2016), which has some more user friendly functions.)

First we will demonstrate a couple commands that will generate the current date for your system (either your physical computer or your cloud computer). Below is the system date:

```
Sys.Date()
```

```
## [1] "2017-04-11"
```

Next we will show the system time down to hours, minutes, and seconds with the Time Zone:

```
Sys.time()
```

```
## [1] "2017-04-11 23:14:29 EDT"
```

Note that that these are not character objects:

```
class(Sys.Date())
```

```
## [1] "Date"
```

This is a special class called the *Date* class. When you read data into R, your fields that have dates are normally converted to the *character* class, not the *date* class. In order to convert from a *character* class to the *date* class in Base R, use the code below.

```
# Create a character vector of random dates
myDates <- c("2016-02-07", "2016-04-02","2016-06-28")

#Convert character vector to dates vector
myDates <- as.Date(myDates)

myDates
```

```
## [1] "2016-02-07" "2016-04-02" "2016-06-28"
```

Now we'll check to make sure we've converted it to the proper class of data:

```
class(myDates)
```

```
## [1] "Date"
```

Now that this is a date object, we can conduct mathematical operations that we could not conduct with a character vector, like subtracting 5 days from all dates:

```
myDates - 5
```

```
## [1] "2016-02-02" "2016-03-28" "2016-06-23"
```

or checking the difference between dates:

```
Sys.Date() - myDates[1]
```

```
## Time difference of 429 days
```

The date formatting code above will only work as described if my input data are formatted exactly as shown, with four-digit years, two-digit months and days, and hyphens in between. In order to convert dates in a different format, you will use the format parameter and describe your unique date format as seen below:

```
# Create a character vector of random dates
myDates <- c("02/07/2016", "04/02/2016","06/28/2016")

#Convert character vector to dates vector
myDates <- as.Date(myDates, format = "%m/%d/%Y")

myDates
```

```
## [1] "2016-02-07" "2016-04-02" "2016-06-28"
```

Below is a table of all the most common date components and their abbreviation.

Conversion Specification	Definition
%a	Abbreviated weekday
%A	Full weekday
%b	Abbreviated month
%B	Full month
%d	Day of the month as decimal number (01–31).
%H	Hours as decimal number (00–23)
%I	Hours as decimal number (01–12)
%m	Month as decimal number (01–12)
%M	Minute as decimal number (00–59)
%p	AM/PM indicator in the locale. Used in conjunction with %I and not with %H
%S	Second as integer (00–61), allowing for up to two leap-seconds
%w	Weekday as decimal number (0–6, Sunday is 0).
%y	Year with two digits (87)
%Y	Year with century (1987)
%Z	Time zone abbreviation as a character string (empty if not available)

5.2 Dates with the Lubridate Package

The *lubridate* package was developed to make date conversions faster and simpler. This package contains a few basic commands that will convert all of the most common date formats without the user having to specify their unique data format.

The basic *lubridate* date conversions are `ymd` (year-month-day), `mdy` (month-day-year), and `dmy` (day-month-year).

We've illustrated how to use these functions below:

```
library(lubridate)
ymd("2016-02-07", "2016-04-02","2016-06-28")
```

```
## [1] "2016-02-07" "2016-04-02" "2016-06-28"
```

Now we'll use `mdy` to convert in a different format.

```
mdy("02/07/2016", "04/02/2016","06/28/2016")
```

```
## [1] "2016-02-07" "2016-04-02" "2016-06-28"
```

To show the flexibility of this code, we'll do a final example with `dmy` used on a different format data:

```
dmy("1jan16", "1nov15","15mar17")
```

```
## [1] "2016-01-01" "2015-11-01" "2017-03-15"
```

The *lubridate* commands can be expanded to include hour-minute-seconds as well

```
ymd_hms("2016-10-10 17:46:52", "2016-11-14 12:04:05", "2016-10-22 22:44:58")
```

```
## [1] "2016-10-10 17:46:52 UTC" "2016-11-14 12:04:05 UTC"
## [3] "2016-10-22 22:44:58 UTC"
```

If you have times in different time zones, you can add a time zone parameter:

```
ymd_hms("2016-10-10 17:46:52", tz="Pacific/Aukland")
```

```
## [1] "2016-10-10 17:46:52 Pacific"
```

Note: UTC and GMT are Greenwich Mean Time (also known as "Zulu" time).

5.3 POSIXct and POSIXlt

To fully understand how dates work in R, you will need study and understand the POSIXct and POSIXlt classes. You can learn more about these by typing `?POSIXct` or `?POSIXlt` respectively.

Chapter 6

Exam

The final exam is a 20 question online test to evaluate your understanding of fundamental R principles. This exam is found at http://ec2-34-200-213-225.compute-1.amazonaws.com/shiny/rstudio/exam.

You must attain a 70% on this exam.

Bibliography

Grolemund, G., Spinu, V., and Wickham, H. (2016). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.6.0.

Wickham, H. (2016). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.6.0.

Wickham, H. and Chang, W. (2016). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 2.2.0.

Wickham, H. and Francois, R. (2016). *dplyr: A Grammar of Data Manipulation*. R package version 0.5.0.