

Introduction to R Programming

DSCOE Team

2017-09-14

Contents

1	Basic Data Structures	5
1.1	The R Programming Language	5
1.2	Installation	6
1.3	R Environment and Workspace	6
1.4	Data Types	9
1.5	Data Structures	10
1.6	Input/Output Data	14
1.7	Getting Help	16
1.8	Practice Problem	17
2	Basic Data Manipulation	19
2.1	Data	20
2.2	Cell-level Data Access	20
2.3	<code>table()</code> Command (and an example of data “cleaning”)	21
2.4	Filter (or subset) data	25
2.5	Using the <code>grep</code> and aggregate commands	28
2.6	Summary	28
2.7	Practice Problem	29
3	Basic Visualization	31
3.1	Bar Plot	31
3.2	Pie Chart	36
3.3	Histogram Plot	37
3.4	Time Series Plot	39
3.5	Practice Problem	43
4	Introduction to Control Structures	49
4.1	<code>if()</code> Statements	49
4.2	Loops	50
4.3	Practice Problem	53
5	Introduction to Dates in R	57
5.1	Dates with Base R	57
5.2	Dates with the <code>Lubridate</code> Package	58
5.3	<code>POSIXct</code> and <code>POSIXlt</code>	59
6	Exam	61

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, and 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 Wall Street 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 five 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 that support our analysts as they tell the data narrative. Additionally, R is generally more accessible across the Department of Defense.

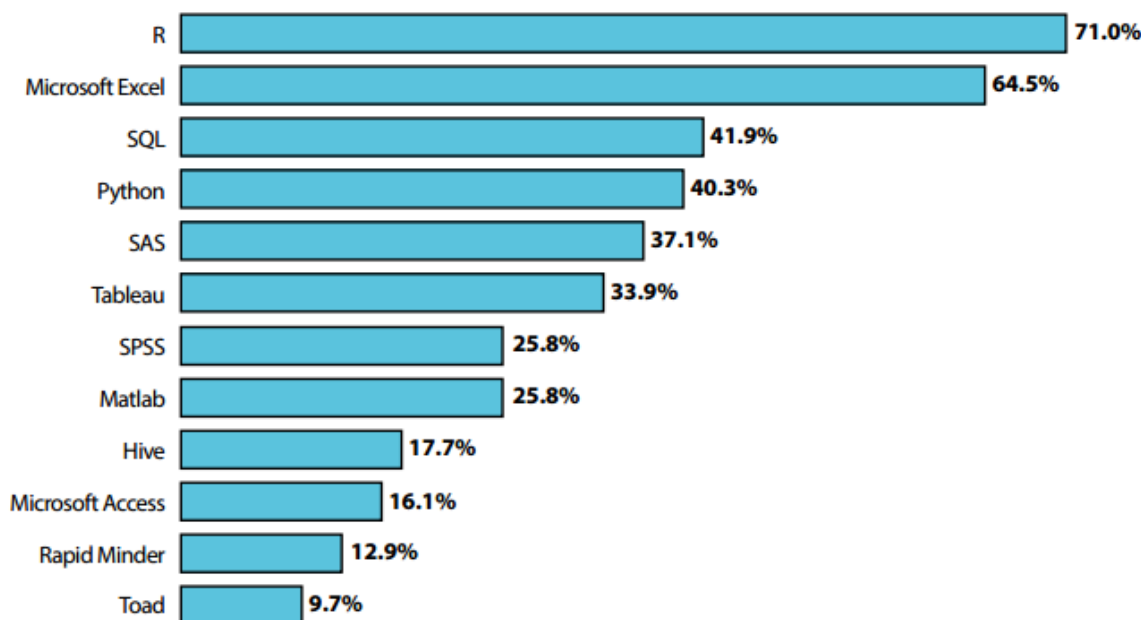


Figure 1.2: Figure 2: LAVASTORM 2014 Survey of Industry: Primary tools used by data scientists

R is open-source and is freely available to download. You can use base R as-is to write and run R scripts. 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 exclusively use RStudio in this course.

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

1.2 Installation

1. Download and install the current version of R on CRAN.
2. Download and install the RStudio Desktop Open Source License by going to the RStudio Downloads page and selecting your operating system.

1.3 R Environment and Workspace

Watch this brief introductory video to explore the RStudio interface:

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 the `getwd()` command in the console:

```
getwd()
```

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

If you want to change where your working directory is, you can do this three different 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 the following:

```
setwd("C:/Users/My.User.Name/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 `dir()` 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"       "apft.csv"
## [9] "_book"                   "book.bib"
## [11] "bookdown-demo_files"     "bookdown-demo.Rmd"
## [13] "bookdown-demo.Rproj"     "_bookdown_files"
## [15] "_bookdown.yml"           "_build.sh"
## [17] "ctx-10008"               "dataframe.PNG"
## [19] "dataWrangling.jpg"       "DD508009"
## [21] "_deploy.sh"              "DESCRIPTION"
## [23] "dplyr.png"               "environment.PNG"
## [25] "facebook.png"            "filterColumn.PNG"
## [27] "filterRow.PNG"           "index.Rmd"
## [29] "KoreanConflict.csv"      "LICENSE"
## [31] "list.PNG"                "matrix.PNG"
## [33] "notebooks"               "_output.yml"
## [35] "packages.bib"            "pcs"
## [37] "per"                     "persistent-state"
## [39] "preamble.tex"            "prerequisite-book.Rproj"
## [41] "prop"                    "rating2.csv"
## [43] "README.md"               "rmd-outputs"
## [45] "s-6E3A07E1"              "saved_source_markers"
## [47] "screen1.png"             "session-persistent-state"
## [49] "style.css"               "summer.csv"
## [51] "tidyr.png"               "toc.css"
## [53] "vector.PNG"              "whyR2.PNG"
## [55] "whyR.PNG"
```

Note that `dir()` lists 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.

Before we get into the many types and shapes of data, let's first see that your RStudio *Console* (appears as the lower left pane in RStudio by default) can execute commands just 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 the result 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
```

I can also use it in future computations:

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

The variable `y` is now stored in your *Global Environment* alongside the stored variable `x`. Think of the *Global Environment* as your “workbench” that contains all of the data and values that you have ready for use. In RStudio, you can usually see what is in your *Global Environment* in the *Environment* pane (appears as a tab in the upper right pane by default).

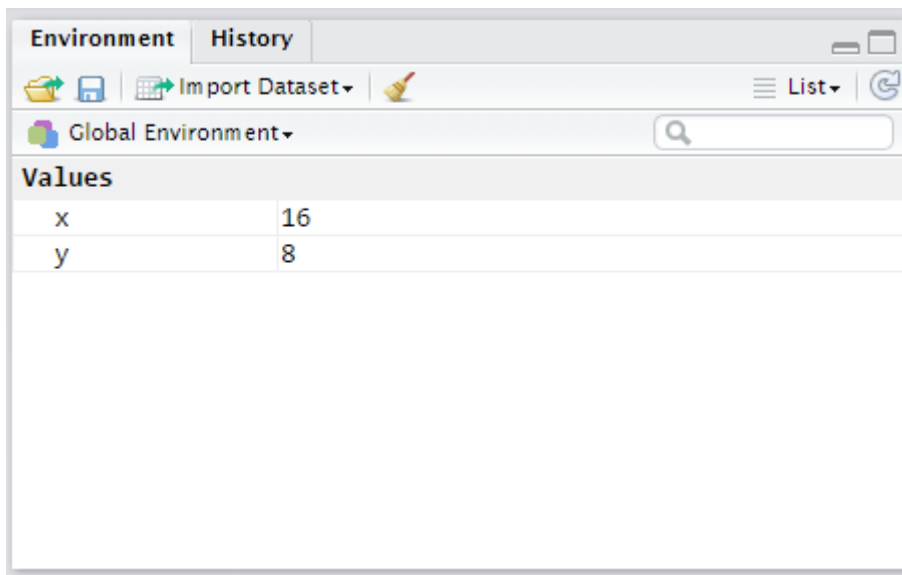


Figure 1.3: Figure 3: The “Environment” pane shows the name and type of data held in memory

If you’re using base R, you can list the variables that are in your *Global Environment* using the `ls()` command:

```
ls()
```

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

When you close either RStudio or base R, it will ask you if you want to save your *workspace*. 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 a long time to run, then it is best to save these items in a workspace so that you don't have to wait hours or even days a second time to recreate them. Generally your clean R code takes only seconds to prepare your data for analysis, so it is best to not save your workspace each time you close R or RStudio.

1.4 Data Types

Now that we have R and RStudio installed, let's look at different classes of data. The basic building blocks are the *integer*, *numeric*, *character*, *date*, *boolean* (logical), and *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, 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 explicitly change it (we will go over this in more detail later):

```
x <- "2014-01-01"
class(x)
```

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

```
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")
```

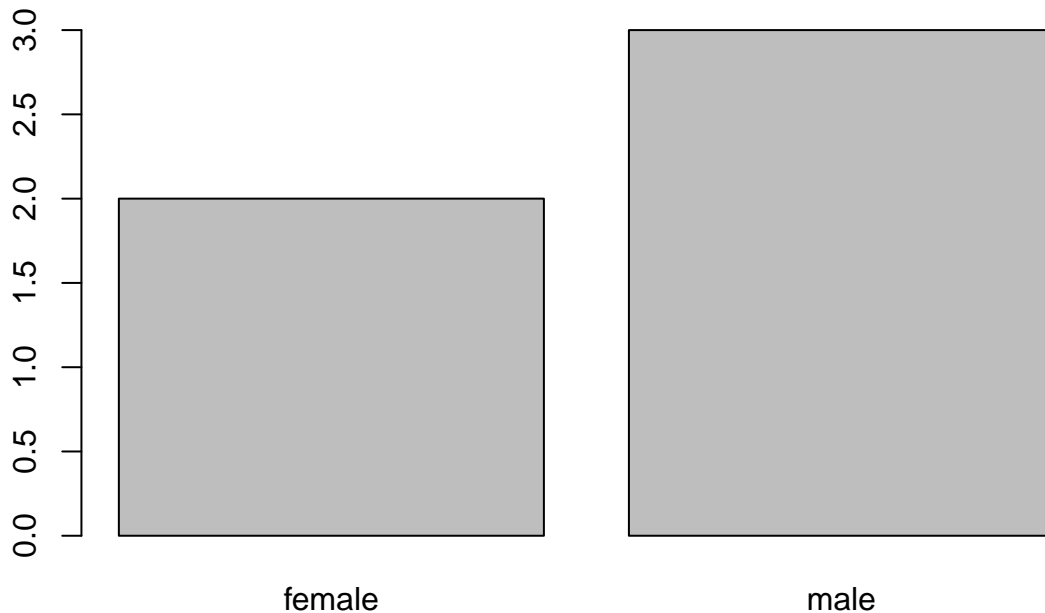
This is character data. If I tried to plot `y` right now, R would show an error, since you can't print *character* data. Let's convert this to a factor now:

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

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

Now we can try to plot `y`:

```
plot(y)
```



It plots a barchart because R recognizes `y` as a factor and has a numeric value associated with both of the “levels” in the factor.

1.5 Data Structures

The data that we showed above is trivial (and very small) data. To work with larger 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

Data types can also be described by the dimensionality of the data they can handle. So far we’ve been using *scalars*, in which our variable `x` is a single value. Data can have 1, 2, or even more dimensions if required.

1.5.1 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:

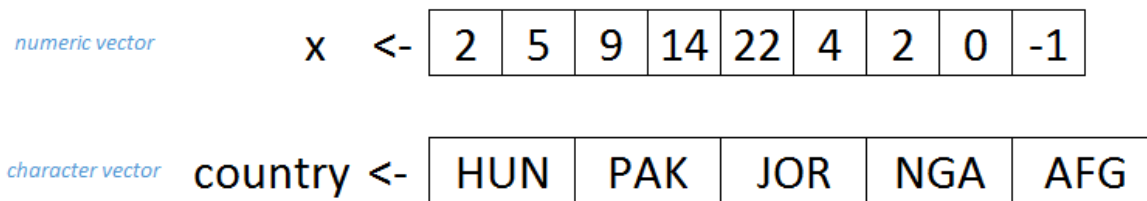


Figure 1.4: Figure 4: Understanding the “Vector” Data Structure in R

An example of creating a vector in R is given below:

```
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 <- 1:10
x

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

If you need to create a vector of a specific sequence, you can use the sequence command:

```
x <- seq(from=2, to=20, by=6)
x
```

```
## [1] 2 8 14 20
```

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

```
x <- rep(1,10)  # Repeat 1 ten times
x

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

1.5.2 Data Frame Data Structure

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

Each column of a data frame is a vector, and each vector 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 also 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
```

apft <-

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 the “Data Frame” Structure in R

```
##      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
```

It is often the case that you might want to extract a subset of your data frame.

If you want just one element from the data frame, we can specify which row and which column to keep.

```
apft[3,1] #This returns the 3rd row, 1st column.
```

```
## [1] Jim
## Levels: Jim John Laura
```

If you want more than one element from the data frame, we can specify which row(s) and which column(s) to keep.

```
apft[2:3,1:2] #This returns the 2nd and 3rd rows, 1st and 2nd columns.
```

```
##      Name Gender
## 2  Laura      F
## 3   Jim      M
```

If you want just a few rows, we can specify which rows to keep.

```
apft[3,] #This returns the 3rd row, all columns.
```

```
##      Name Gender PU SU   Run
## 3   Jim      M 49 60 12:36
```

If you want just a few columns, we can specify which columns to keep.

```
apft[,1] #This returns all rows, first column.
```

```
## [1] John Laura Jim
## Levels: Jim John Laura
```

Extracting one column in a data frame is typically accomplished with the \$ operator.

```
apft$Name
```

```
## [1] John Laura Jim
## Levels: Jim John Laura
```

1.5.3 List Data Structure

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

A list is helpful in several situations. For example, there are times you might have vectors that do not all have the same length. For example, lets say we extracted hashtags from tweets at the Rio Olympics. The number of hashtags per tweet can range from zero to seven or eight (see Figure 6 below).

[[1]]	#USA	#Olympics	#Swimming	#phelps
[[2]]	#Rio2016			
[[3]]	#rio	#soccer		
[[4]]	#olympics	#cycling	#brazil	

Figure 1.6: Figure 6: Understanding the “List” Data Structure in R

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. 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 1 1 1 1 1 1 1 1 1
##
## [[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.

1.5.4 Matrix Data Structure

While matrices exist as 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 the “Matrix” Data Structure in R

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

```
## Create a 2x3 matrix filling row by row.
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
```

With the exception of the \$ operator, matrices are indexed in the same way as data frames. As mentioned previously, we will not use matrices much in this course.

1.6 Input/Output Data

Let’s now 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-2/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. You can always convert from the provided character strings to another data type, but using `as.is=TRUE` ensures you know your starting point. One example where the factor interpretation is particularly problematic involves working with dates.

Now that we've read the file in, we'll briefly explore this data object with a few helpful commands. 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 King (1994)" ...
## $ genres : chr "Adventure|Animation|Children|Comedy|Fantasy" "Action|Drama|War" "Action|Adventure|Sci-Fi" ...
```

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
```

You can also use the command `head()` to print only the first 6 rows. This gives titles of the variables (columns) as well as a feel for the data:

```
# The head command prints the first six 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
```

1.7 Getting Help

There are 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 the R documentation and any 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
```

1.8 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-2/KoreanConflict.csv", destfile = "KoreanConflict.csv")
```

Read the file `KoreanConflict.csv` into your R environment. Explore the data using the commands that we learned during this lesson. At this point, you should be able to discover the following:

1. How many observations (rows) are in your Korean Conflict data frame?
2. How many variables (columns) describe each observation?
3. Which column of your data frame describes an individual's duty position? (Hint: Try adding a `$` after the name of your data frame to access a specific column of data.)
4. What is the range of birth years in your Korean Conflict data frame? (Hint: Use the `min()` and `max()` functions on the appropriate column.)

Chapter 2

Basic Data Manipulation

The most time-intensive task in any data science endeavor is often pre-processing the 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 data.

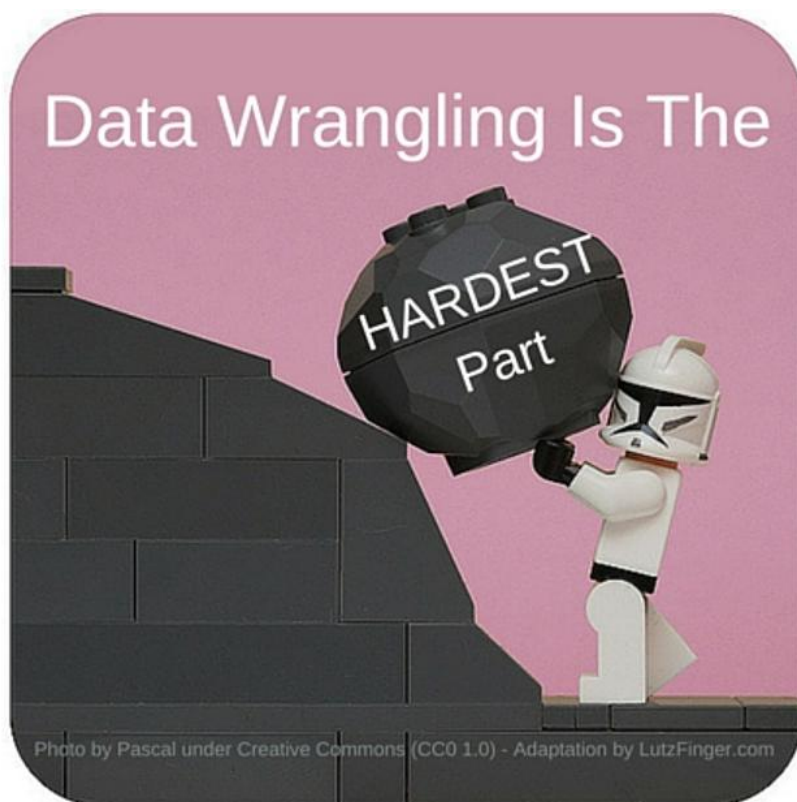


Figure 2.1: Figure 1: “Data Wrangling” is often the most difficult part of data science

2.1 Data

For this lesson we will use casualty data from the Korean War. This data set and many other interesting data sets are 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/RESERVE" ...
## $ 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 COMMANDER" ...
## $ 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

Let's briefly review some common indexing techniques for data frames. This data set contains two dimensions (rows and columns). To access specific rows and columns in R, we use the `[row, column]` format. For example, to access the data in the first row and first column of our Korean Conflict 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"
```

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)] # First three rows, columns 1, 3, and 8.
```

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

If you want a subset of the rows and all columns, we can simply specify which row(s) to keep.

```
kor[3,] # This returns the 3rd row, all 25 columns.
```

If you want all rows and a subset of the columns, we can simply specify which columns to keep.

```
kor[,1] # This returns all 36,574 rows, first column.
```

Extracting one column in a data frame is typically accomplished with the \$ operator.

```
kor$RANK # This returns all 36,574 ranks.
```

You can also use multiple 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 command:

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

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

The command above selects the RANK column from the kor data frame and then prints 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, there are numerous entries that don't look like a year. We can see this again if we table data by sex:

```
table(kor$SEX) # Table by sex.
```

```
##
##              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 VEHICLE (3D ECHELON)
##           1           9
```

Note that this doesn't give just *male* and *female*. If you explore the data frame extensively, you'll discover a number of instances where particular values of POSITION always lead to issues in BIRTH_YEAR and SEX. For our purposes, we're simply going to remove this *messy* data. Note that in some cases you will want to fix messy data, which typically requires figuring out how the data was improperly read or stored and then correcting it. We will do this in two steps: 1) remove any rows with non-numeric birth years, and 2) remove any remaining rows containing an invalid sex.

First, 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. The following code coerces the BIRTH_YEAR 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. Reexamining the birth year table shows only reasonable birth years now while excluding any NA values from the counts.

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

```
##
## 1889 1894 1895 1896 1900 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911
##    1    1    1    1    5    7    2   15   14   25   22   26   48   61   76
## 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926
##  104  116  143  183  224  300  424  421  506  624  657  781  888 1107 1278
## 1927 1928 1929 1930 1931 1932 1933 1934 1935
## 1988 3621 4358 5479 5077 3630 1296  328   61
```

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.

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

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 (or equivalently, retaining all rows that do NOT contain an NA in BIRTH_YEAR). Now let's 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 data wrangling tutorial purposes, we are just going to delete it.

Now let's see what still needs to be cleaned up in the `SEX` column. To do that, let's call the `table()` command again:

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

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

Notice that all of the bizarre `SEX` labels have disappeared. Perhaps something about the `BIRTH_YEAR` errors were causing or at least related to the `SEX` errors. The data we've examined is now clean, and notice that 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  CW02  CW0-2  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  W01
##    97   18
```

From this we learn that the PFC rank sustained the highest casualty numbers, and the highest ranking casualty was a General (assuming this means 4-star General). Now let's explore `NATIONALITY`. We might have assumed that this data was exclusively US nationality, but we see otherwise when we table by `NATIONALITY`:

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

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

When we table the `MARITAL_STATUS` field, we find something interesting:

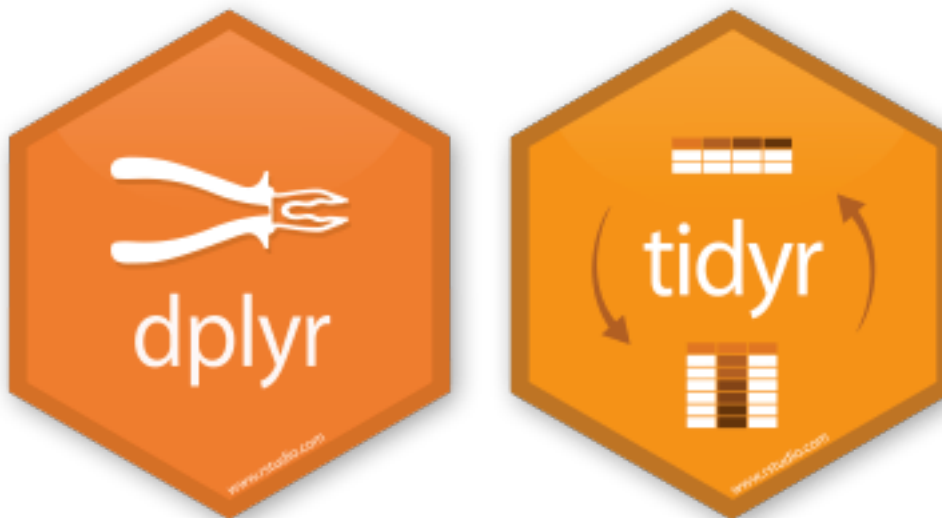
```
table(kor$MARITAL_STATUS)
```

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

The marital status of most of the casualties was unknown! This may make you wonder about the Defense Department data collection during the Korean Conflict. 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 are 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 special packages called `dplyr` (Wickham et al., 2017) and `tidyr` (Wickham, 2017) just for data wrangling. For the sake of simplicity, we will 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 Test (APFT), we may want to extract rows based on **SEX**, as seen below.

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

APFT_FEMALE <-

Figure 2.2: Figure 2: Filtering Rows by Categorical Variable

We can conduct this same operation (extract all FEMALE records) on our `kor` data frame with the `dplyr` package using 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 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
```

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

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

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

We now know 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 who were over 30 in the year 1950. If we want to only select those individuals that were in their 30s in 1950, we would use the following `dplyr` command:

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

Running this command shows that 2,052 of the casualties were in their 30s in 1950.

Now that we've filtered by row, let's 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 are not of interest. In these cases, it is often helpful to extract only 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

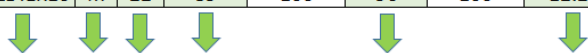


Figure 2.3: Figure 3: Filtering Specific Columns (or fields)

Let's say we are studying the Korean Casualty data to understand the time factor of those who died of wounds, and we are 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 let's 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 we 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) 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 and the death of the Service Member).

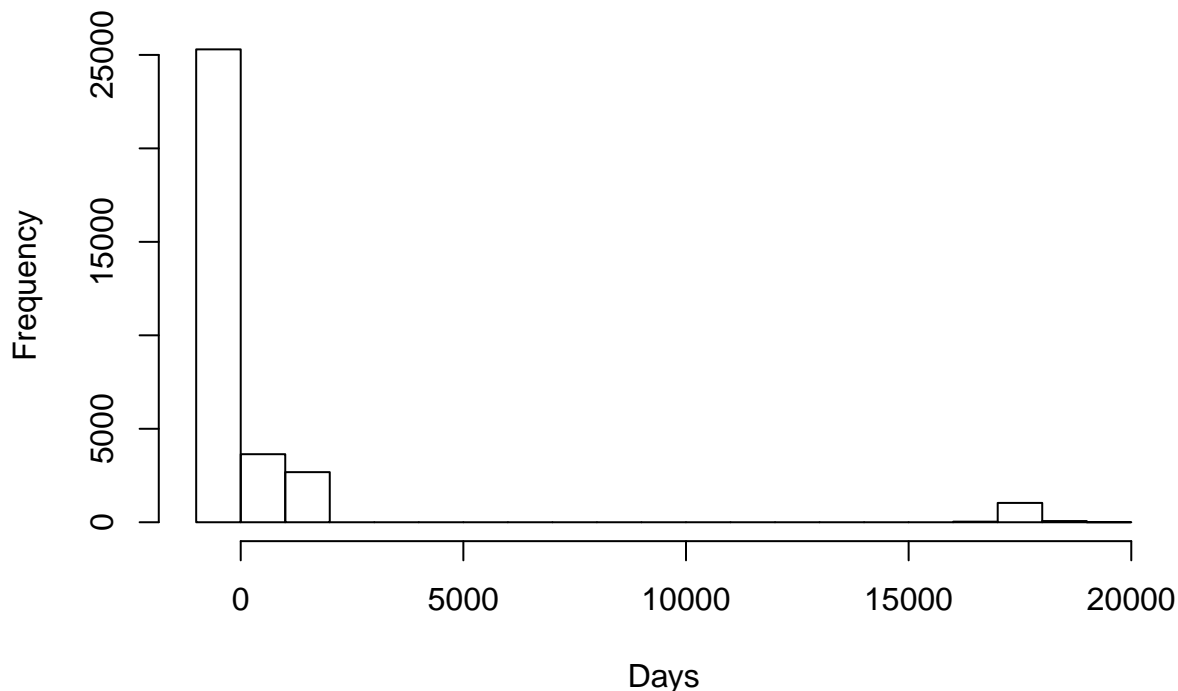
```
library(lubridate)
days <- ymd(kor_dates$FATALITY_DATE) - ymd(kor_dates$INCIDENT_DATE)
days[1:100]
```

```
## 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 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 should raise some questions. 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. Let's plot a histogram of the difference in days (you'll learn this command during the next lesson):

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

Histogram of Difference in Days



Here we see that there are many casualties that have a fatality day around the year 2000. If you look at the original data, you 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 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 without addressing the presence of lengthy MIA periods in the 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 a new APFT data set.

```
download.file("https://s3.amazonaws.com/dscoe-data-2/apft.csv", destfile = "apft.csv")
```

2.6 Summary

What we have seen is that R provides a great platform to rapidly “wrangle” and explore data. Exploring a data set allows you to find items of interest as well as any limitations that may not be obvious at first. There is no perfect approach to do this exploration of a data set, but there is great value in learning the commands necessary to follow your curiosity and intuition.

2.7 Practice Problem

1. Use `grep` to determine how many casualties had *INFANTRY* somewhere in the 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, and new data visualization packages are created on a regular basis. In addition to packages that create basic statistical visualization (line plots, bar plots, pie charts, etc.), there are packages that create geospatial visualizations, 3D visualizations, and interactive visualizations.

Let's start by reading in the Korean Conflict data and performing the primary cleaning functions that we performed last lesson.

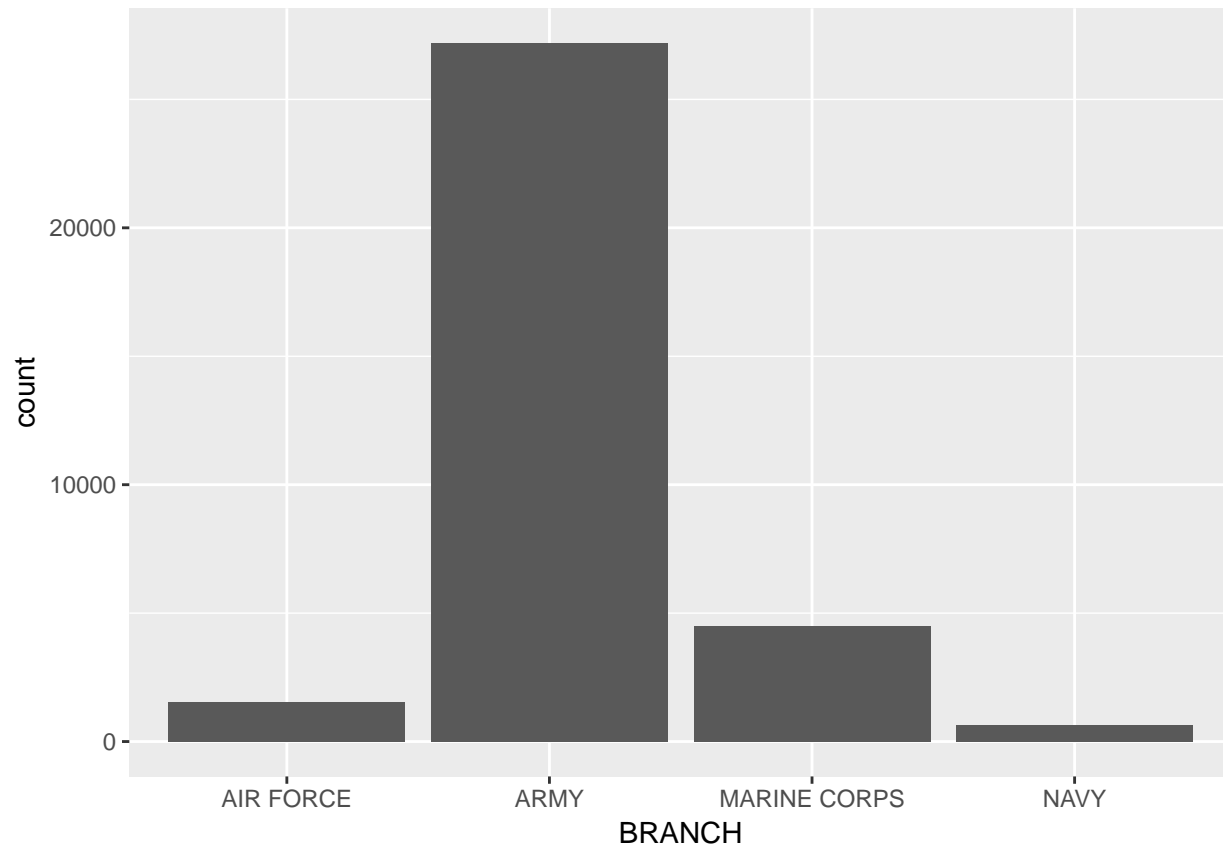
```
kor <- read.csv('KoreanConflict.csv', as.is=TRUE)
kor$BIRTH_YEAR <- as.numeric(kor$BIRTH_YEAR)
kor <- kor[!is.na(kor$BIRTH_YEAR),]
```

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, 2017). If you haven't installed this package before, 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.

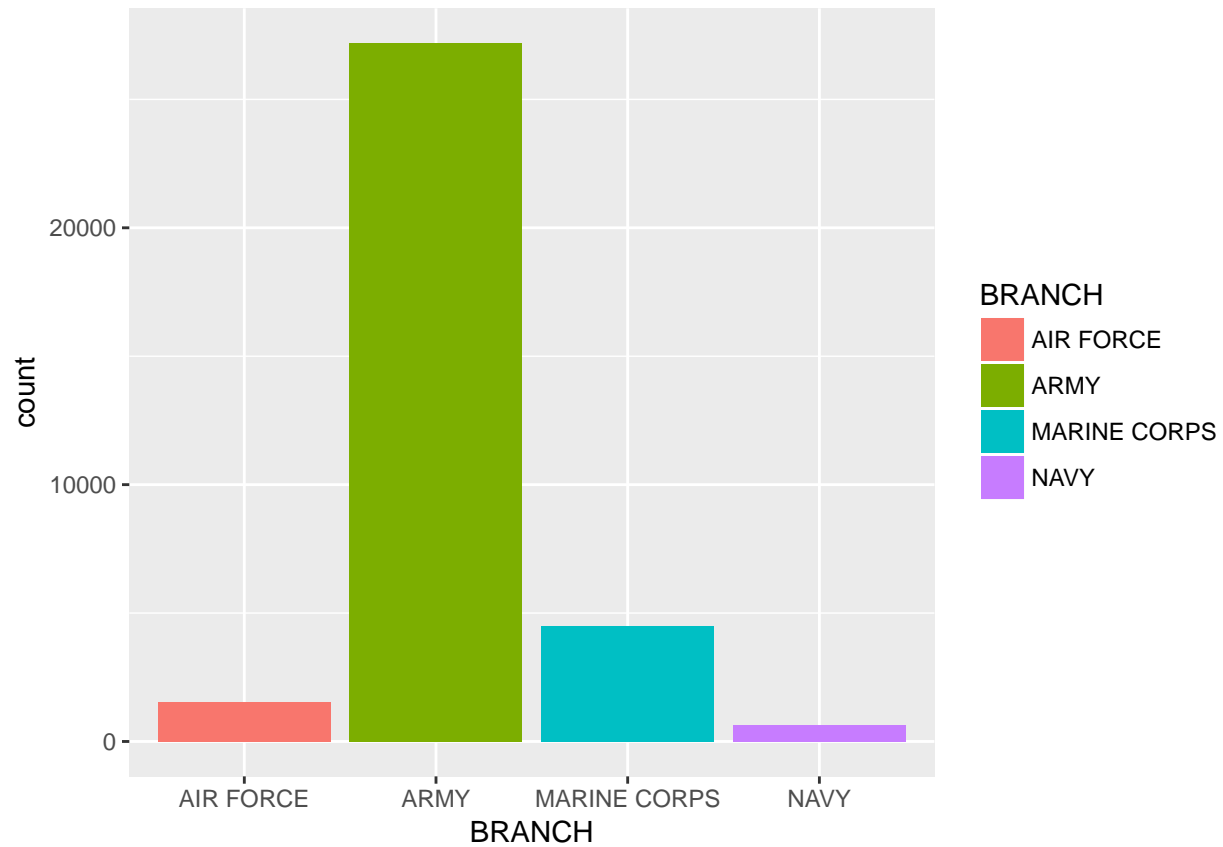
```
library(ggplot2)
ggplot(kor, aes(BRANCH)) + geom_bar()
```



Let's break down what this command is actually doing. The `ggplot` command is the start of any `ggplot2` visualization. In this `ggplot()` command, we say that we want a visualization of the `kor` data set with a focus on the `BRANCH` variable, referred to as the *aesthetic* (`aes`) in the `ggplot2` package. The next command says to take the data we've identified (`BRANCH` variable from the `kor` data frame) and create a barplot (`geom_` specifies the geometry or shape/style of plot we want).

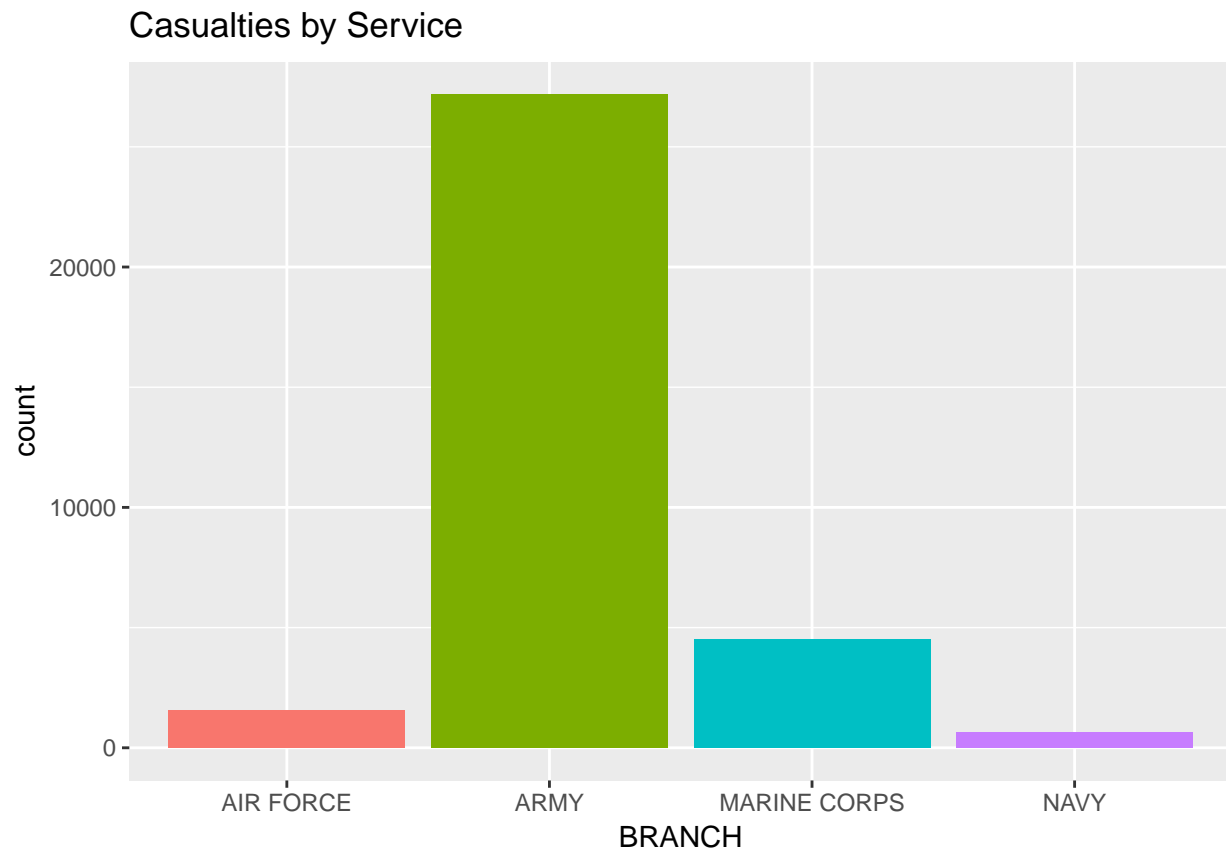
We can uniquely color the bars by adding `fill = BRANCH` to specify that each bar's fill color should be associated with the `BRANCH`.

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

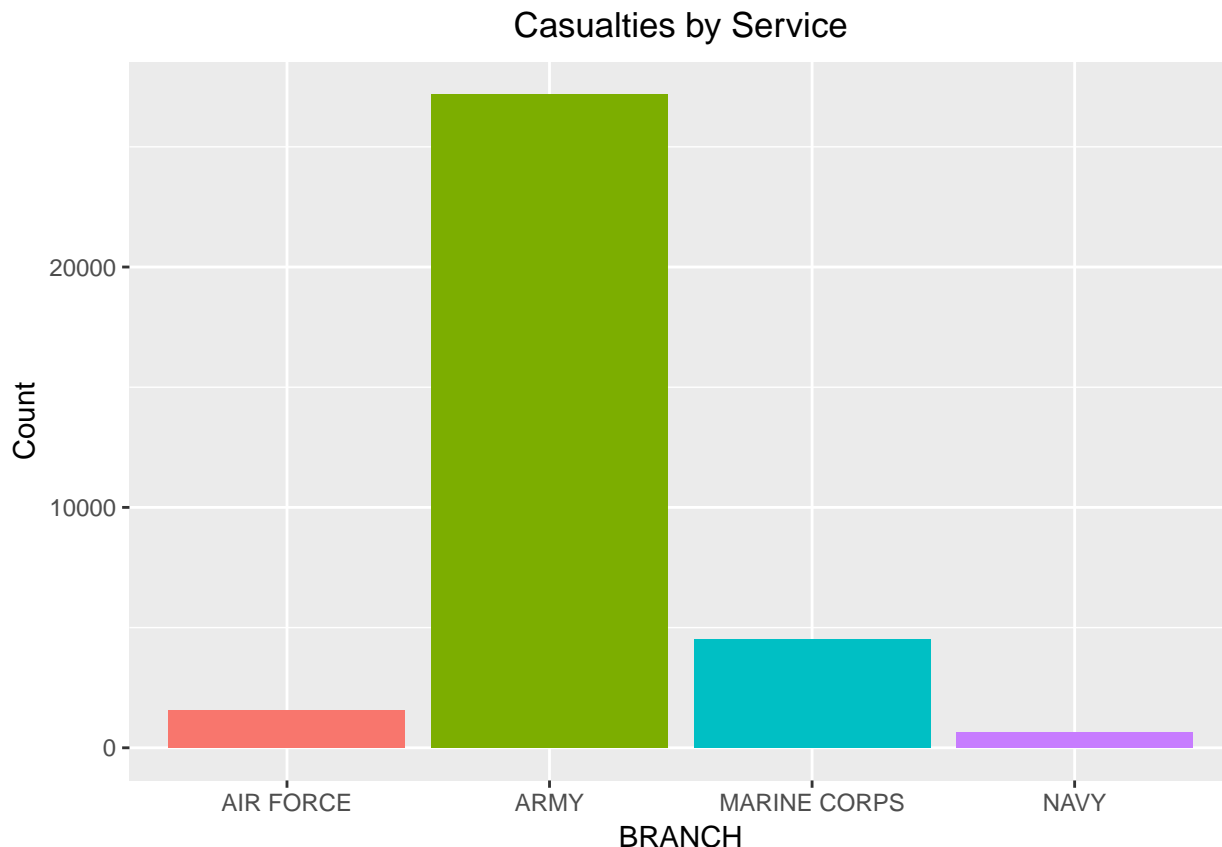
We may also want to add a title and get rid of the legend since the labels are already on the horizontal axis.

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



This plot is starting to look pretty nice, but centering the title and capitalizing the vertical axis label would be good finishing touches. While the code below may look complicated, a quick internet search will produce a “how-to” for just about any plot modification you can imagine. As an example, perform a Google search for “center a ggtitle” (you should find the solution below in the first linked search result).

```
ggplot(kor, aes(BRANCH, fill = BRANCH)) + geom_bar() +  
  ggtitle("Casualties by Service") +  
  theme(plot.title = element_text(hjust = 0.5)) +  
  ylab("Count") + theme(legend.position="none")
```



As we continued adding more and more pieces to our plot command, you'll notice each piece was simply connected with the `+` operator. The “gg” in `ggplot()` stands for “grammar of graphics,” and you can think about each component of our plot command being an aspect of a sentence. We first defined the subject when we specified `kor` and `BRANCH`. We added the “what” and “how” when we instructed R to build a certain type of plot in a certain way. Think about this “grammar of graphics” structure as we continue to build more and more complex and tailorable visualizations.

Now let's build a stacked barplot. Say we wanted to see the distribution of `ETHNICITY` in each of 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
```

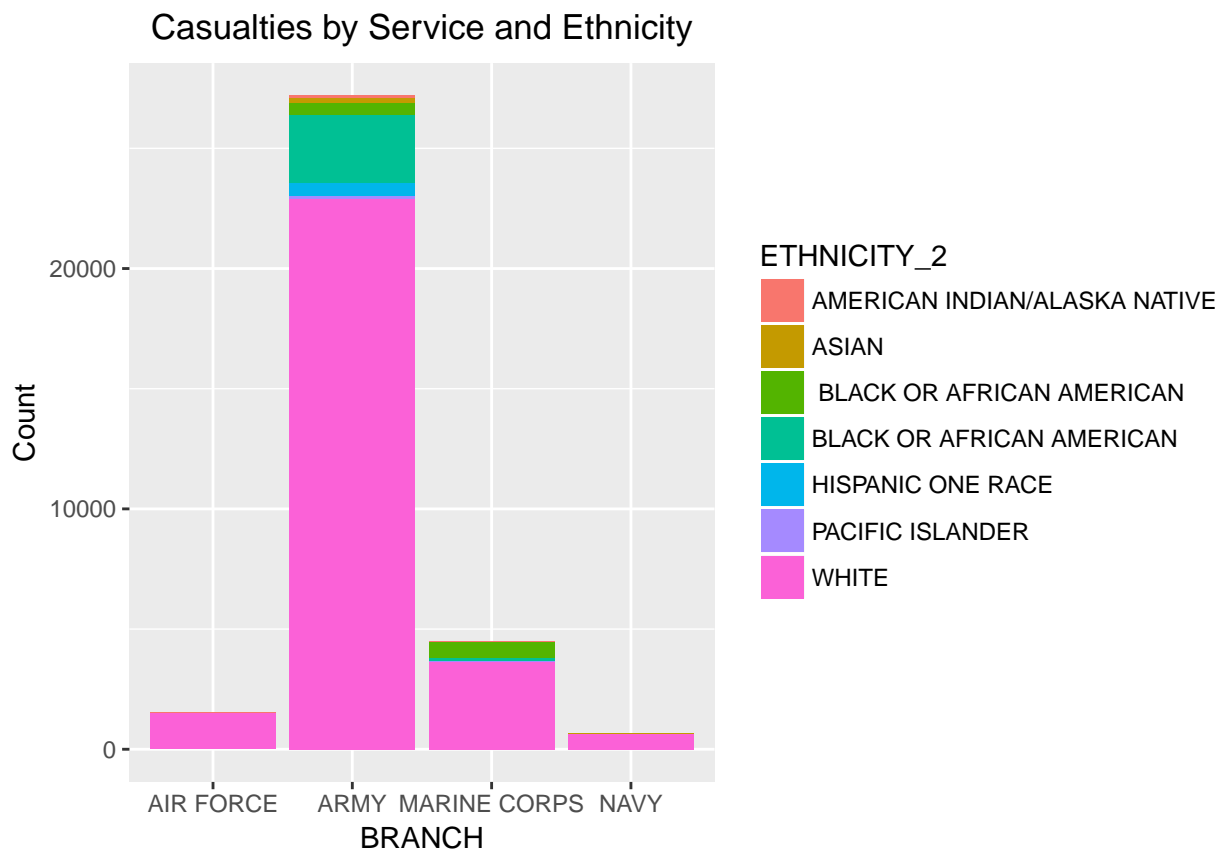
These are rather long names to display on a chart. We will start by creating shorter names. We can do this

in the code below using the `grep()` command that we learned in Lesson 2. We can replace “Native Hawaiian or other Pacific Islander” with simply “Pacific Islander” as follows:

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

Now all we have to do in our previous code is change the `fill = BRANCH` to `fill = ETHNICITY_2`. We also don’t want to discard the legend anymore.

```
ggplot(kor, aes(BRANCH, fill = ETHNICITY_2)) + geom_bar() +  
  ggtitle("Casualties by Service and Ethnicity") +  
  theme(plot.title = element_text(hjust = 0.5)) + ylab("Count")
```

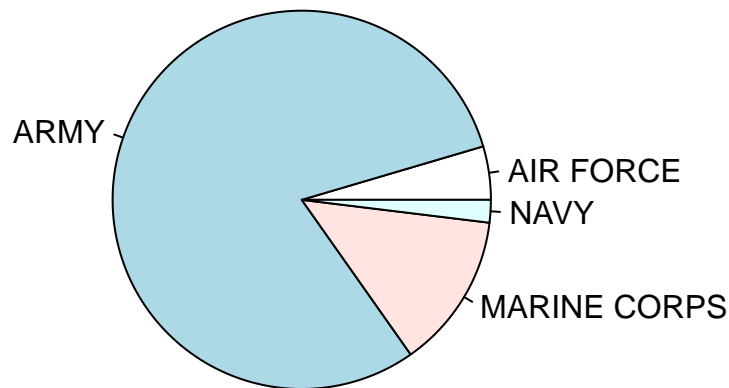


3.2 Pie Chart

Statisticians will generally tell you that you should never use a pie chart. 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 times when a pie chart 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 chart is very easy to produce if we wrap the `pie()` command around the `table` command:

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

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. R will create a histogram in one line of code.

Let's create a histogram of the age for each Korean Conflict 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).

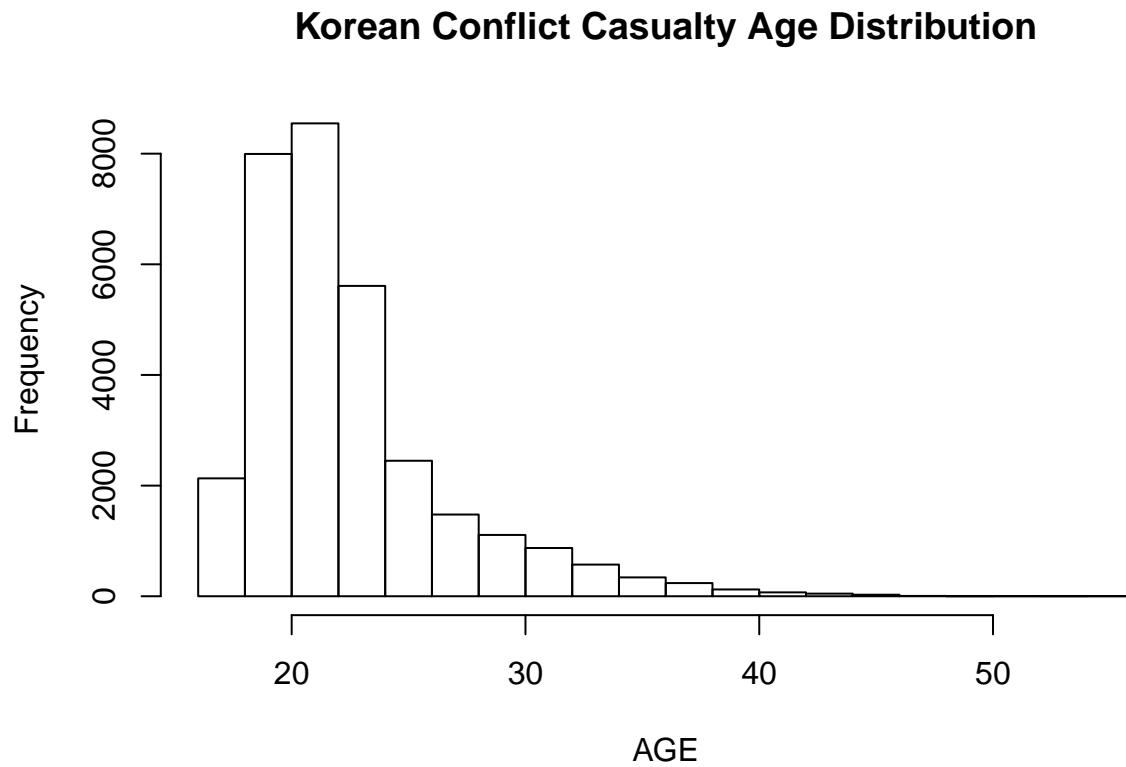
```
# Filter our MIA
kor2 <- dplyr::filter(kor, FATALITY_DATE < 1960)

# 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
```

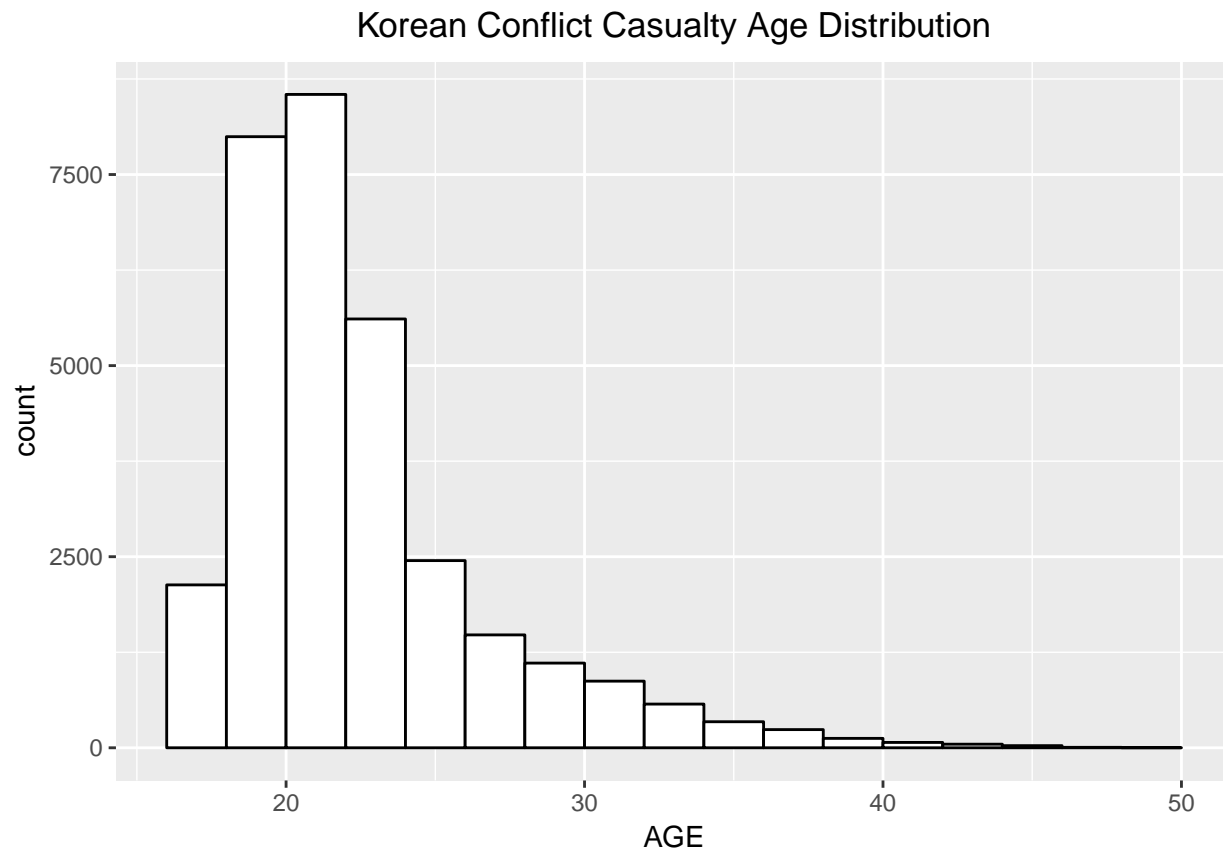
Now that we've created the *AGE* field, we will use two different techniques to create a histogram. The Base R package has a histogram function that is easy to use and helpful for exploring data.

```
hist(kor2$AGE, freq=TRUE, main = "Korean Conflict Casualty Age Distribution", 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(breaks=seq(16,50, by=2), col="black", fill="white") +  
  ggtitle('Korean Conflict Casualty Age Distribution') +  
  theme(plot.title = element_text(hjust = 0.5))
```



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 now to look at the Olympic medal counts. This data set lists all Olympic medals won during a Summer Olympics from 1896 to 2012. This data set is available here:

<https://s3.amazonaws.com/dscoe-data-2/summer.csv>

You can download the file with the following command:

```
download.file("https://s3.amazonaws.com/dscoe-data-2/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 Sailors"
## $ 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. The `dplyr` package offers an `aggregate()` function that will help us do this. Let's start by aggregating the number of Olympic medals by `Year` and `Gender`:

```
# Take the full list of athletes, split it up into many sublists
# by year and by gender, and return the length of each sublist.
oly_sum <- aggregate(Athlete ~ Year + Gender, data=oly, length)
```

Let's take a look at the first few and last 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
```

```
tail(oly_sum)
```

```
##   Year Gender Athlete
## 48 1992  Women     600
## 49 1996  Women     777
## 50 2000  Women     889
## 51 2004  Women     899
## 52 2008  Women     932
## 53 2012  Women     924
```

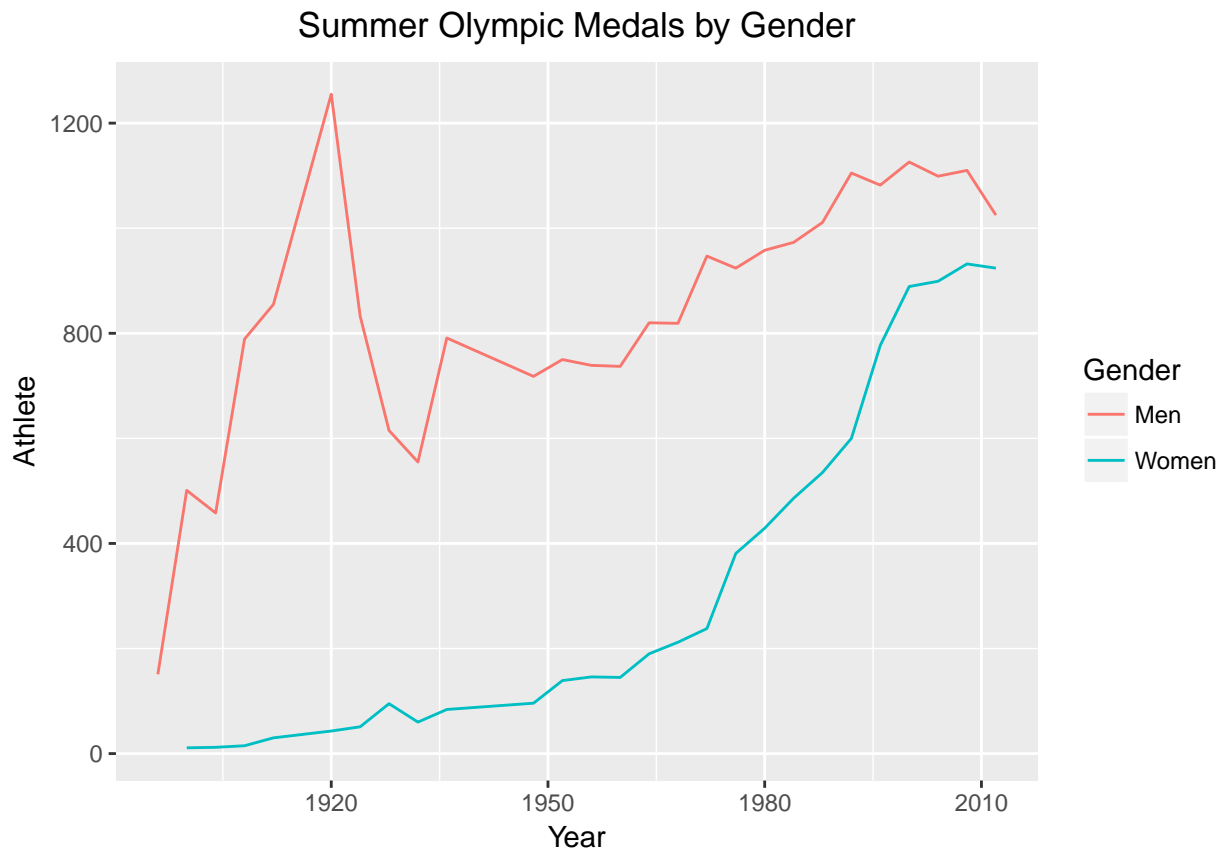
A final check for correctness requires us to sum the `oly_sum$Athlete` counts. If the sum equals the total number of athletes (31,165), that's a good sign.

```
sum(oly_sum$Athlete)
```

```
## [1] 31165
```

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 by specifying `Year` as our horizontal or x-axis and `Athlete` as our vertical or y-axis:

```
ggplot(oly_sum, aes(x=Year, y=Athlete, color = Gender)) +
  geom_line() + ggtitle('Summer Olympic Medals by Gender') +
  theme(plot.title = element_text(hjust = 0.5))
```

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

Now let's select a few of the prominent countries in the Summer Olympics. Specifically, we'll look at the permanent members of the UN Security Council: United States, Great Britain, France, China, and Russia. 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 *URS* data to *RUS*.

```
# Convert all URS medals to RUS medals.
oly$Country[oly$Country=="URS"] <- "RUS"
```

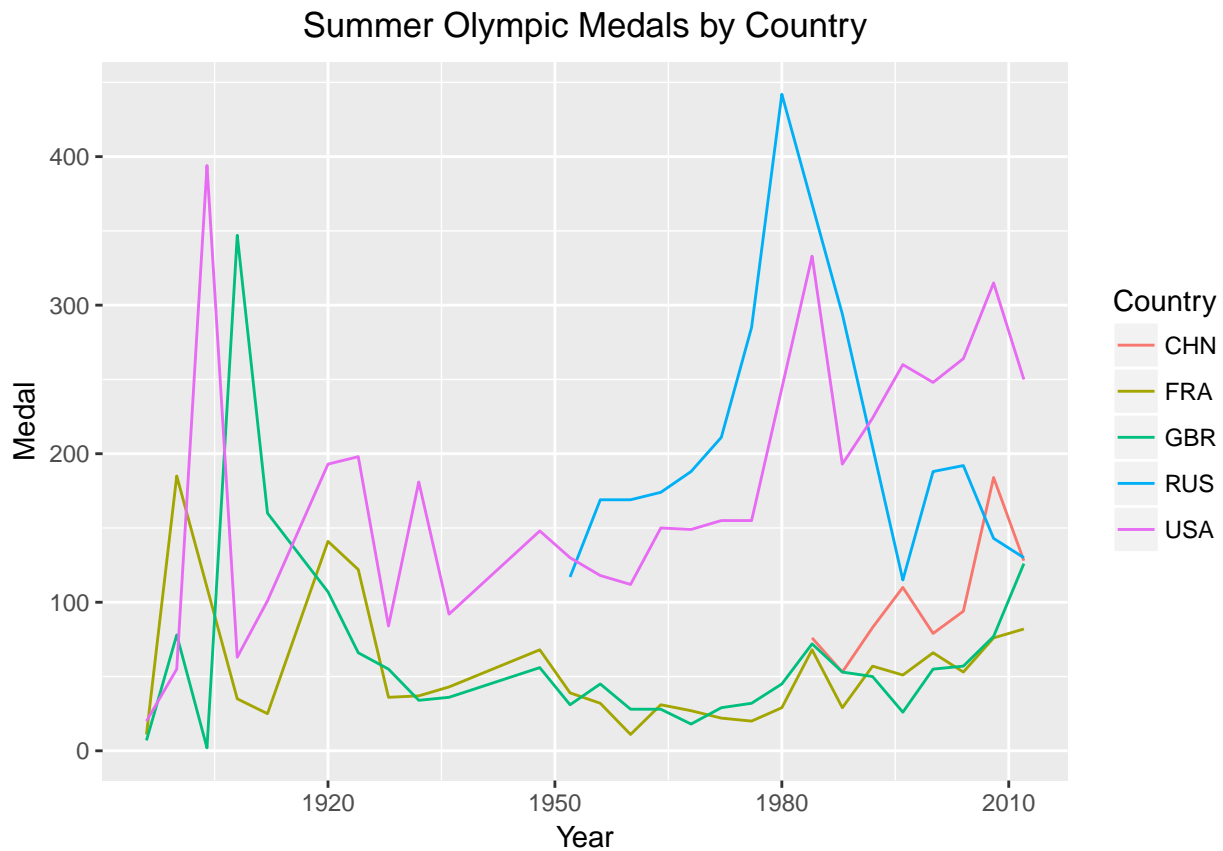
Now we will aggregate the data by **Year** and **Country** as well as reduce the country list to only those of interest. In the code below, notice how we create a vector of our five countries and then use the `%in%` function to filter out all countries not contained in our list. In other words, we consider each medal winner and retain that individual record only if his or her country is *in* our country list.

```
library(dplyr)
# Aggregate all medals counts by country and year.
oly_country <- aggregate(Medal ~ Country + Year, data = oly, length)
country.list <- c('USA', 'GBR', 'FRA', 'CHN', 'RUS')
# Remove all counts associated with countries outside our list.
oly_country2 <- filter(oly_country, Country %in% country.list)
head(oly_country2)
```

```
##      Country Year Medal
## 1      FRA 1896     11
## 2      GBR 1896      7
## 3      USA 1896     20
## 4      FRA 1900    185
## 5      GBR 1900     78
## 6      USA 1900     55
```

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

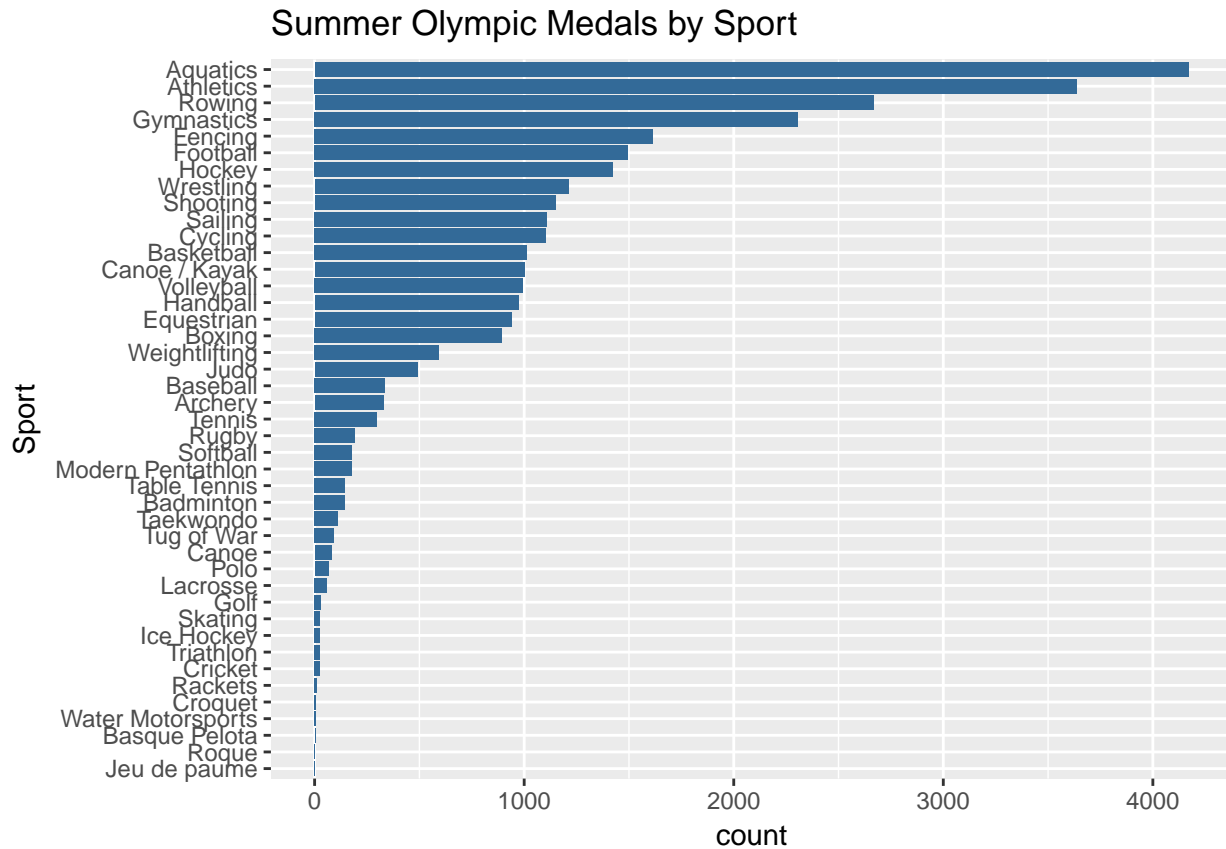
```
ggplot(oly_country2, aes(Year, Medal, color = Country)) +
  geom_line() + ggtitle('Summer Olympic Medals by Country') +
  theme(plot.title = element_text(hjust = 0.5))
```



This plot highlights a bit of the history of the summer Olympics. Note that Russia began competing in the 1950s, and China didn't begin competing until the mid-1980s. 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 decreased to ~50 medals in the 1920s and stayed near this mark for most of 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.



Do your best to work this out on your own, but if you get stuck, use these hints to help you figure out what to do next.

3.5.1 Hint #1

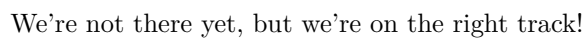
- 1) The plot above indicates we will need medal counts by sport. We've done this before with the Korean Conflict data. Let's adapt that code for our new purpose.

```
# This code produced a histogram of Korean Conflict casualties
# grouped by branch of service.
```

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

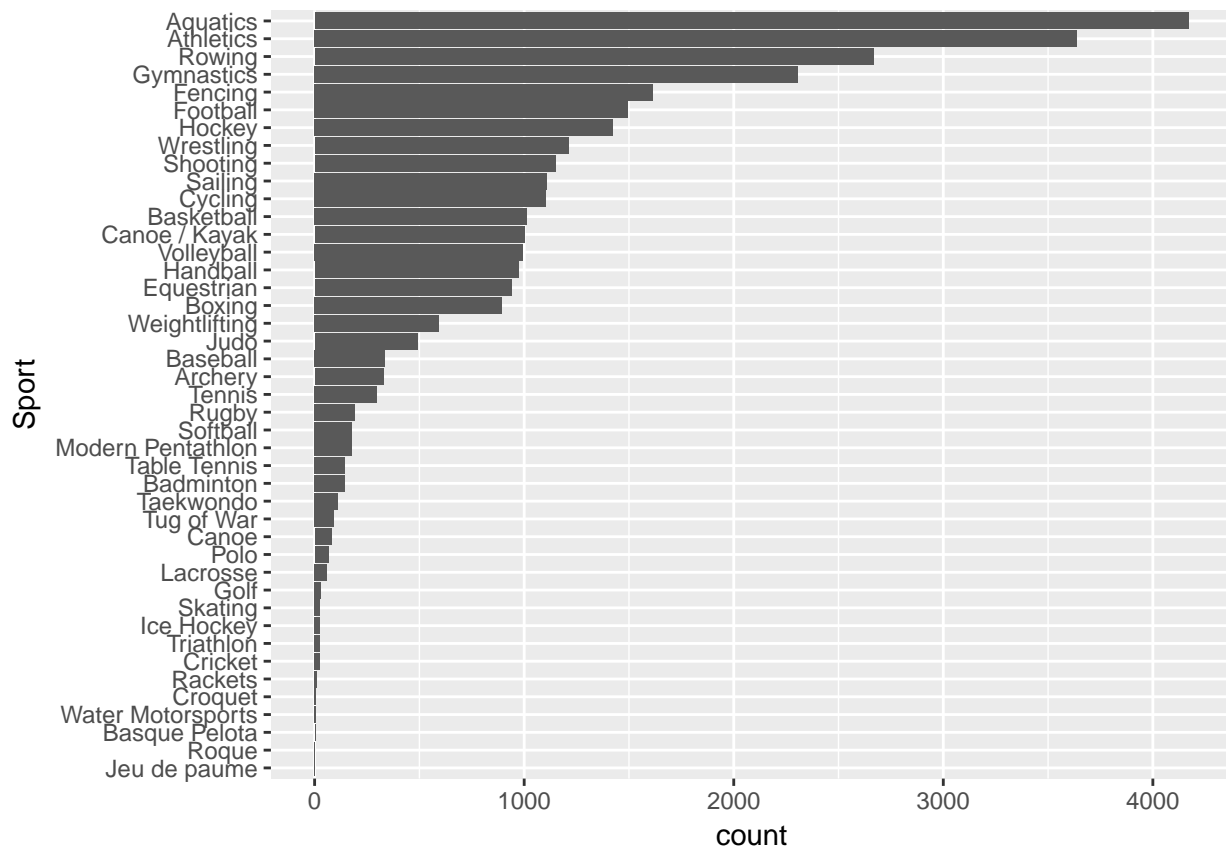
```
# Our new code should produce a histogram of Olympic medal winners
# grouped by sport.
```

```
ggplot(oly, aes(Sport)) + geom_bar()
```



2) Let's figure out the horizontal requirement for this histogram. This is a good time to point out some RStudio cheatsheets that are accessible from your RStudio interface. Go to the **Help** menu in RStudio, click **Cheatsheets**, and select the **Data Visualization with ggplot2** link. If you explore this sheet and/or do some quick Googling ("horizontal barplot in ggplot2"), you'll find the command `coord_flip()`.

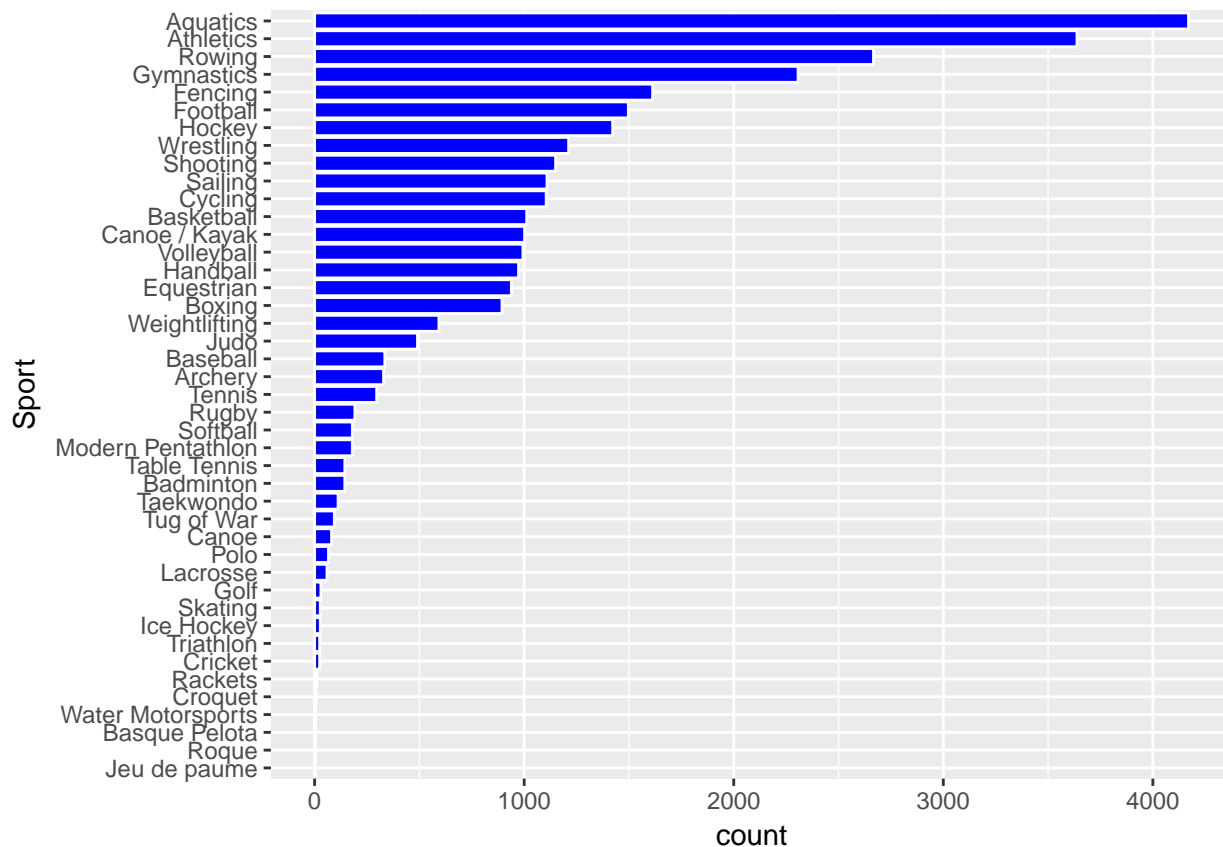
```
ggplot(oly, aes(Sport)) + geom_bar() + coord_flip()
```



3.5.3 Hint #3

- 3) The example plot has blue bars. Let's change that in our plot. Our cheatsheet says the `geom_bar()` command has `color` and `fill` options we could explore. It looks like `color` is the color of the rectangle edges, and `fill` is the color inside the rectangles. Let's use a white border and a gray fill.

```
ggplot(oly, aes(Sport)) + geom_bar(color="white", fill="blue") + coord_flip()
```

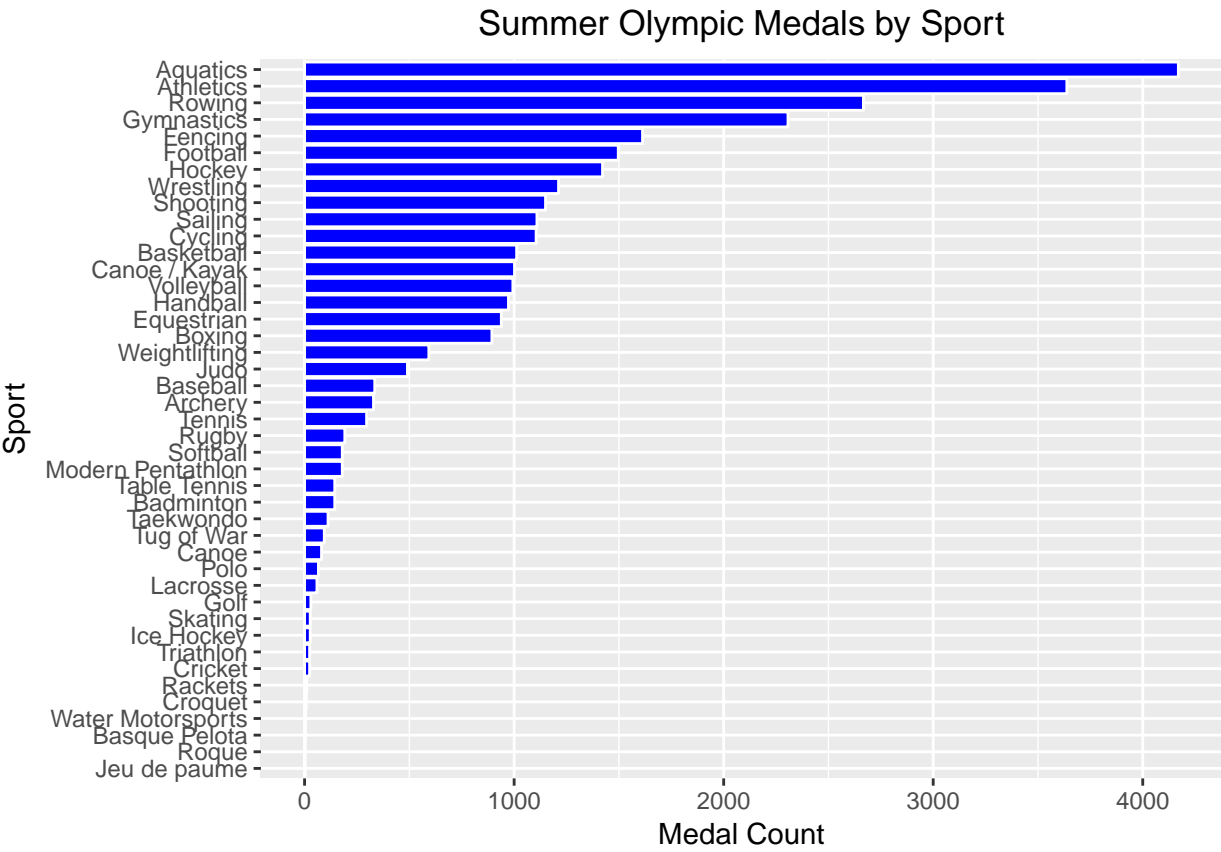


It starting to look pretty good now. Let's polish this plot with a title and better axes labels.

3.5.4 Hint #4

- 4) We have plenty of previous examples to copy some good title and axis label code. We can make our new plot look even better than the one we're trying to replicate.

```
# Note: the Medal Count is shown as the "ylab" instead
# of "xlab" because we used coord_flip()
ggplot(oly, aes(Sport)) + geom_bar(color="white", fill="blue") + coord_flip() +
  ggtitle('Summer Olympic Medals by Sport') +
  theme(plot.title = element_text(hjust = 0.5)) +
  ylab("Medal Count")
```



Mission accomplished!

Chapter 4

Introduction to Control Structures

This lesson will cover the `if()` statement as well as the `for()` loop and `while()` loop. These are three very common control structures for all computer programming languages and are used extensively in the R programming language.

4.1 `if()` Statements

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

Let's see this in practice:

- Case 1

```
my.number = 1  
if(!is.numeric(my.number)) {  
  print("You were supposed to store a number in 'my.number'.")  
} else if(my.number >= 10){  
  print("My number is greater than or equal to 10.")  
} else {  
  print("My number is less than 10.")  
}
```

```
## [1] "My number is less than 10."
```

- Case 2

```
my.number = 100  
if(!is.numeric(my.number)) {  
  print("You were supposed to store a number in 'my.number'.")  
} else if(my.number >= 10){  
  print("My number is greater than or equal to 10.")  
}
```

```

} else {
  print("My number is less than 10.")
}

```

```
## [1] "My number is greater than or equal to 10."
```

- Case 3

```

my.number = "huge"
if(!is.numeric(my.number)) {
  print("You were supposed to store a number in 'my.number'.")
} else if(my.number >= 10){
  print("My number is greater than or equal to 10.")
} else {
  print("My number is less than 10.")
}

```

```
## [1] "You were supposed to store a number in 'my.number'."
```

The `else` clause(s) can grow rather lengthy if many special cases must be handled uniquely, but it is quite often the case that we just need an `if()` statement. Think of this statement as a gate you must pass in order to run a chunk of code.

```

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

```

An example you might use in practice is boundary checking. Suppose you have age data, and you want to be notified if there is a negative age recorded.

```

my.age <- 1
if(my.age < 0) {
  print("Negative age detected. Check for data entry error.")
}

```

We didn't see our notification because we didn't pass the `if()` gate that reaches the `print()` command. We skipped right over it.

```

my.age <- -1
if(my.age < 0) {
  print("Negative age detected. Check for data entry error.")
}

```

```
## [1] "Negative age detected. Check for data entry error."
```

Now we get the message because we passed the `if()` condition.

The `if()` statement is most often used in *loops* and *functions*. We'll illustrate the use of the `if()` 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 potentially accomplish a task many times along the way. The `for()` loop and the `while()` loop will be the primary loops used in 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 might 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")
for(i in 1:5){
  print(letters[i])
}
```

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

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 `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 lists of lists. For example, the outer `for()` loop iterates over countries, and 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
# create a 10 x 10 matrix (10 rows and 10 columns)
```

```

mymat = matrix(nrow=10, ncol=10)

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

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)
  # Here we set a new value of i.
  i <- i + 5
  # Now we go back and check the while() condition.
}

```

```

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

```

To finish out this lesson, we'll provide an example of some more sophisticated code. This function simulates a round of play in the board game *RISK*. In the board game *RISK*, an 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 *attacker* 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()` statements are:

```

# Our function will take as inputs the number of armies the
# attacker has, the number of armies the defender has, and
# the desired number of simulated trials the user wants.
risk <- function(attacker, defender, n) {
  # Create an empty vector to hold the simulated win/loss outcomes.
  results <- rep(NA, n)
  # Create restoration values to reset the armies after each trial.

```

```

attacker.reset <- attacker
defender.reset <- defender
# For n total trials...
for(j in 1:n){
  # Reset the attacker and defender for a new trial.
  attacker <- attacker.reset
  defender <- defender.reset
  # ...as long as the attacker has more than one army
  # and the defender has an army...
  while(attacker > 1 & defender > 0) {
    # Both players pick up the right number of dice...
    atk.dice <- min(attacker-1, 3)
    def.dice <- min(defender, 2)
    # Both players roll the right number of dice...
    atk.roll <- ceiling(runif(atk.dice)*6)
    def.roll <- ceiling(runif(def.dice)*6)
    # Each player's dice are sorted from greatest to least...
    atk.roll <- atk.roll[order(atk.roll,decreasing=T)]
    def.roll <- def.roll[order(def.roll,decreasing=T)]
    # Compare the correct numbers of dice for each player...
    comparison <- min(atk.dice, def.dice)
    for (i in 1:comparison) {
      if (atk.roll[i] > def.roll[i]){
        # The attacker won, so the defender loses an army...
        defender <- defender-1
      }
      if (atk.roll[i] <= def.roll[i]){
        # The defender won, so the attacker loses an army...
        attacker <- attacker-1
      }
      # Keep going up to the number of required comparisons.
    }
    # Keep going until the attacker or defender is wiped out.
  }
  # If the defender ran out of armies, the attacker wins.
  if (defender==0) results[j] <- "Attacker"
  # If the defender did not run out of armies, the defender wins.
  if (defender>0) results[j] <- "Defender"
  # Keep going up to the number of required trials.
}
print(paste("The Probability of the Attacker winning is:",
            length(results[results=="Attacker"])/n))
}

```

The statement below illustrates how to use the `risk()` function:

```
risk(attacker=4,defender=3,n=1000)
```

```
## [1] "The Probability of the Attacker winning is: 0.456"
```

4.3 Practice Problem

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

- Read the movie ratings data you downloaded in Section 1.6 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)
rating <- read.csv("rating2.csv", as.is = TRUE)
genre.list <- unique(unlist(str_split(rating$genres,"\\|")))
```

Try to execute the following steps before looking at the code solution that shows just one way to accomplish this task.

- Create a vector object that is the same length as `genre.list` 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.

4.3.1 Hint #1

- Create a vector object that is the same length as `genre.list` to contain the `mean` ratings.

The `rep()` command replicates a value or set of values a specified number of times. The `length()` command returns the number of elements in a vector or list object. We can use these two functions to create a vector of the correct length.

```
# We'll replicate NA as our placeholder to be replaced by actual numbers.
mean.ratings <- rep(NA,length(genre.list))
```

4.3.2 Hint #2

- 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. Note that many movies are classified as belonging to multiple genres, and we will allow each movie's rating to contribute to the calculation of the mean rating for any genre to which it belongs.

```
# It appears most movies are associated with multiple genres.
head(rating$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"
```

We might be tempted to set up our `for()` loop from `1:length(rating)`, but that would only loop over the 7 objects inside our `rating` data frame. We might also be tempted to loop over the 283,886 movies, which we can do by looping over any column of `rating` (e.g. `1:length(rating$title)`). What we really want to do is loop over our vector of genres (i.e., `1:length(genre.list)`). This approach will let us complete an assessment of each genre all at once instead of moving through our list movie by movie.

```
# We'll iterate over our vector of genres.
for(i in 1:length(genre.list)){
  # Let's filter out any movies that don't belong to the genre of interest.
```

```
# First identify which movie indices correspond to the genre of interest.
this.genre.indices <- grep(genre.list[i],rating$genres)
# Now let's compute and store the mean rating of this genre.
# We'll compute the mean of "rating" column in the rows we just identified.
mean.ratings[i] <- mean(rating[this.genre.indices,"rating"])
}
```

Now we have a vector of mean ratings, which we can combine with the genre list into a data frame and display as text here.

```
df.rating <- data.frame(genre.list,mean.ratings)
df.rating
```

```
##          genre.list mean.ratings
## 1      Adventure    3.481759
## 2      Animation    3.579487
## 3      Children     3.420384
## 4        Comedy     3.358041
## 5        Fantasy     3.464419
## 6         Action     3.452660
## 7         Drama     3.641138
## 8          War      3.690831
## 9       Sci-Fi      3.499506
## 10      Musical     3.521342
## 11         IMAX     3.524784
## 12      Romance     3.457683
## 13      Horror     3.261698
## 14      Thriller     3.502560
## 15        Crime     3.654538
## 16        Western     3.601346
## 17        Mystery     3.639658
## 18    Documentary     3.552925
## 19      Film-Noir     3.780648
## 20 (no genres listed)  2.929924
```

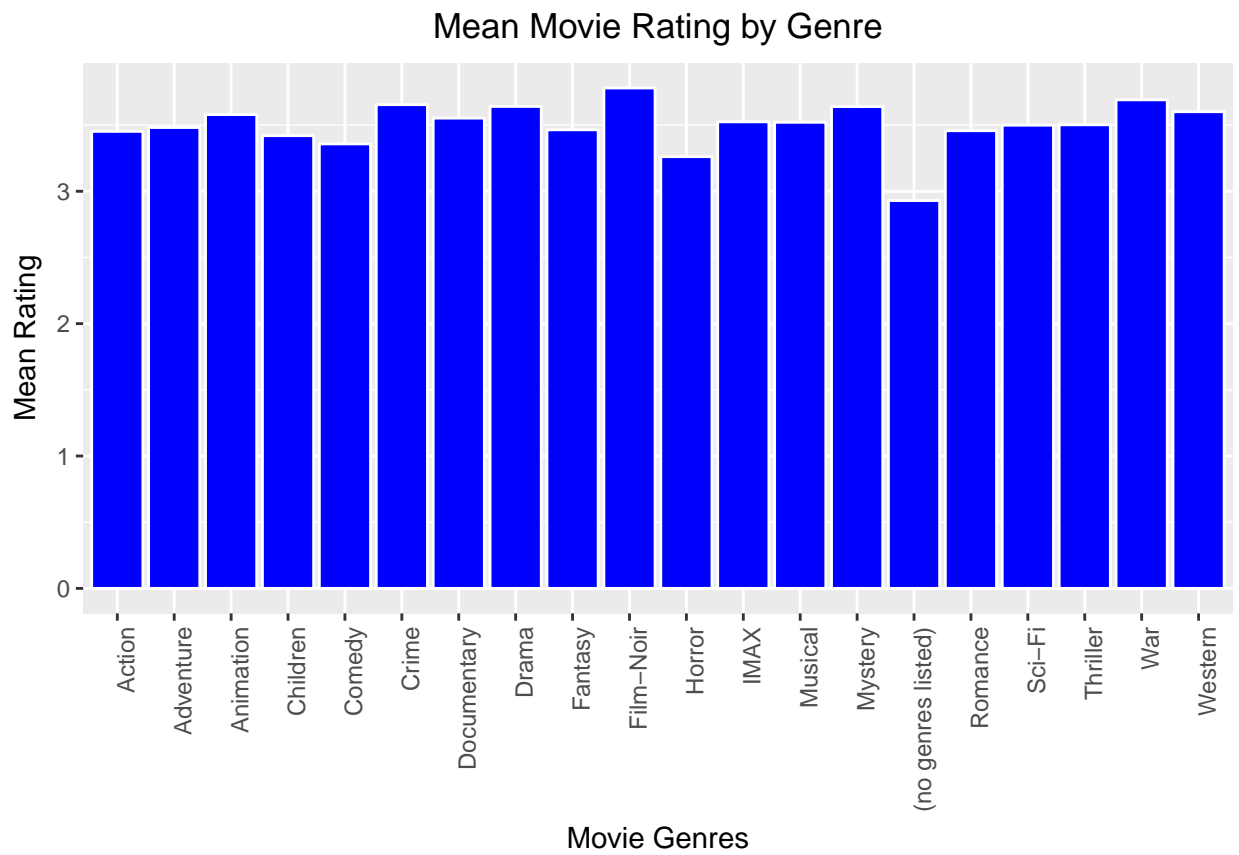
4.3.3 Hint #3

- Create a barplot visualization of average ratings by genre.

We already wrote some code to do exactly this with our Olympic medal data. Let's repurpose our old code and adjust it as necessary. Don't reinvent the wheel!

Note: You'll see the argument `stat="identity"` in the `geom_bar()` command. You have the option to let R compute means, counts, etc., from our chosen y-data. In our case, we already did that. The "identity" option means we just want the values to be presented exactly as we've provided them.

```
library(ggplot2)
ggplot(df.rating, aes(x=genre.list,y=mean.ratings)) +
  geom_bar(stat="identity", color="white", fill="blue") +
  ggtitle('Mean Movie Rating by Genre') +
  theme(plot.title = element_text(hjust = 0.5)) +
  ylab("Mean Rating") +
  xlab("Movie Genres") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



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 really handy, 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] "2017-09-14"
```

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

```
Sys.time()
```

```
## [1] "2017-09-14 13:11:49 UTC"
```

Note 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, any 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 `as.Date()` command as seen 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 585 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 from 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 date format:

```
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/Auckland")

## [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, it will be helpful to 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://ec2-34-207-128-58.compute-1.amazonaws.com/shiny/rstudio/exam/> Note that at times the “shiny” subdomain text will fall out of this link. If this happens, type it back in and press enter again.

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. (2017). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.6.3.
- Wickham, H. and Chang, W. (2017). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. <http://ggplot2.tidyverse.org>, <https://github.com/tidyverse/ggplot2>.
- Wickham, H., Francois, R., Henry, L., and Müller, K. (2017). *dplyr: A Grammar of Data Manipulation*. R package version 0.7.1.