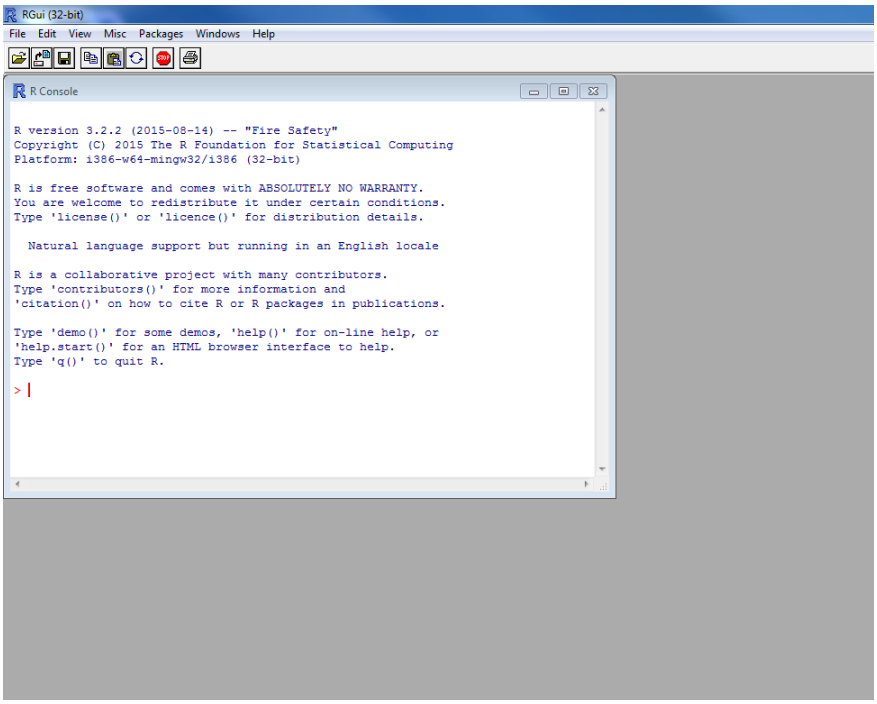
Welcome to R!

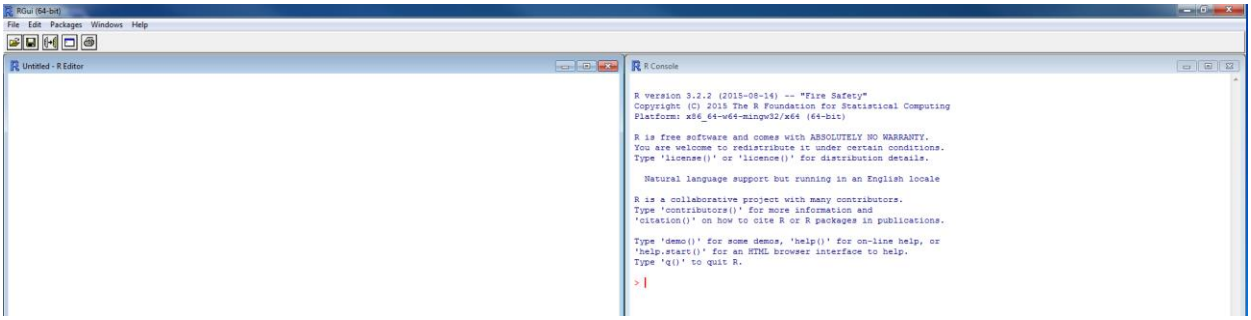
This guide is suitable for new R-users or advanced level R-users looking for information on specific topics. It was created using R version 3.2.2. You can find the most recent copy of R from the webpage below. If your computer already has R, make sure that it is R3.0.2 or higher in order to run some packages, especially Swirl, which we use later in the guide. The data used in most of the sections is a modified version of the 2014 Toronto Municipal Political Poll, and the two other data sets used are fabricated data sets created for the purpose of this guide.
https://cran.rproject.org/bin/windows/base/

Table of Contents

Opening R



When you first open R, it looks like this. The console window is how all commands are executed in R. As you enter commands in **red**, R will respond in **blue**. First, you should open the new script window, which is found under **File** and then **New Script**. For clarity as you work, next you should organize the windows, which can be done through **Windows** and then either **Tile Horizontally** or **Tile Vertically**.



Your screen should look like this, with your script editor on one side and the console on the other. Now, you are ready to begin using R!

Beginning Work

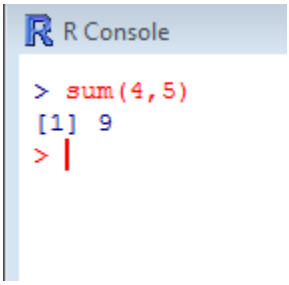
Scripting

Because R is a script based program (meaning that you have to type what you want it to do rather than clicking buttons), the editor window allows you to carefully write the script that you want and then enter it into the console window. Scripts can be saved so you can use them again and again. There are three options to run command lines or selections (you select by highlighting with your cursor) from the editor window:

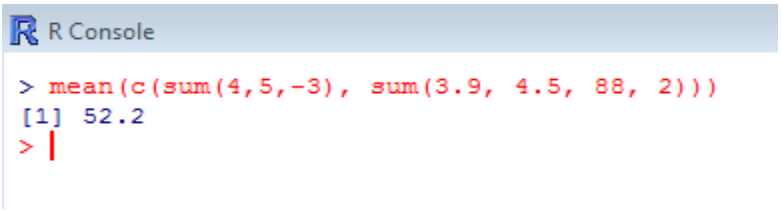
1. Right click to bring up a menu. “Run line or selection” is the first option.
2. Ctrl + R will automatically run the line or selection
3. The **Run line or selection** button is the centre button beneath **Edit** and **Packages** (the button does not appear if you’re in the console window).

If you are writing your commands directly into the console window, you just have to hit **enter** to have R run them.

The basic format for commands is the combination of functions and information. Functions are a word or a few words, sometimes just abbreviations of words, followed by brackets, for example: **read.table()**, **xrange()**, or **plot()**. The words give you a sense of what the command is supposed to do, and inside the brackets is where you place the useful information like which table you want read, what the x-range is, or what data you want plotted. Commands can be just one function or multiple functions strung together. Here are some examples:



This gives you the sum of 4 and 5.



This gives you the mean of the sums of the two different sets of numbers, thus combining the sum function and the mean function. Once you know more about R, you can even write your own functions. More on scripting will come later.

Troubleshooting tips:

- Commands are case-sensitive, i.e., load() will run and Load() will not
- Always check spelling – some commands are pluralized and some are not, i.e., names() will run and name() will not
- Check for spaces and misplaced symbols

Useful hotkeys:

- Save: 'Ctrl + S'
- New Script: 'Ctrl + N'
- Run line or selection: 'Ctrl + R'
- Undo: 'Ctrl + Z'
- Clear window: 'Ctrl + L'

Packages

R is a bare bones program. Its utility comes from the 3500+ packages available for installation. You may think of these packages as similar to extensions or apps, which some other programs use, like Google Chrome or Apple products. Each package serves a purpose and has specific commands you can use. Some are good for creating graphics; some are good for statistical analysis. Not to worry, you won't need to download all of them. Packages can be found on the Comprehensive R Archive Network (CRAN), and there are many blog posts about some of the best R packages to use. The CRAN webpage for the package is important to find because it will, typically, list all of the commands associated with the package and their functions.

Using packages is a twostep process. First you must install the package you want, and then you must load it into R. To begin, you set your CRAN mirror, which is the second option under the **Packages** button on the menu bar. You want the CRAN mirror closest to your location because this will affect the time it takes to download. Also, not every package is available on all mirrors, so if you don't see what you want, set your mirror to **0-Cloud [https]** if you want to view every single package. Because packages are open source, you will want to regularly check for updates.

A. Menu:
    1. Packages -> Install Packages -> Select [package]
    2. Packages -> Load package -> Select [package]
B. Command:
    1. install.packages("package")
    2. library(package)

Troubleshooting tips:

- Check that your desired package is loaded into R
- Packages are built on each other. When installing a new package, it should install the ones it depends on too. If a package is not working, double-check that its dependent packages are also installed.
- Because packages are open source, be sure you check for updates frequently; this does not occur automatically ("Update packages" is an option under **Packages**).
- When all else fails, there are, usually, comprehensive entries on CRAN on each package.

Learning R Online

Since you're new (or even if you're trying to re-familiarize yourself or learn new things), the best package to download is Swirl. Swirl is like a game which will take you through every aspect of R. There are many courses and lessons to choose from depending on your familiarity with the program and what you want to do with R. The lessons preloaded in Swirl from its R Programming course are:

1. Basic Building Blocks
2. Workspace and Files
3. Sequences of Numbers
4. Vectors
5. Missing Values
6. Subsetting Vectors
7. Matrices and Data Frames
8. Logic
9. Functions
10. lapply and sapply
11. vapply and tapply
12. Looking at Data
13. Simulation
14. Dates and Times
15. Base Graphics

Going through those will give you a solid foundation for you to begin using R all on your own. More advanced courses made by Swirl are Data Analysis, Mathematical Biostatistics Boot Camp, Open Intro, Regression Models, Getting and Cleaning Data, and Statistical Inference. All of this can be found at the following weblink:
https://github.com/swirldev/swirl_courses

The remainder of this guide will walk you through R if you don't want to use Swirl or if you want to quickly familiarize yourself with a function or process.

Data Types in R

Beginning note: it is highly recommended that you have R open and try things out as you learn them in this guide. The data files used to create this guide can be found in the folder, so you can follow along.

The R environment can be used to compute calculations and assign variables. As a new R-user, you might want to practice these simple exercises by typing them into the console window:

```
> 2+2
[1] 4
> sqrt(64)
[1] 8
> b<-"Hello"
> b
[1] "Hello"
> vector<-c(1,2,3,4)
> vector
[1] 1 2 3 4
```

There are different data types in R. These data types can be numeric, integer, logical/boolean, character/string, vector, matrix, array, list, data-frame etc. It is useful to know the data type in order to know what functions can be performed on the object. To determine the type of data, you can use the **class()** or **mode()** function. The following commands create different variables and check their type using the **class()** function. It is possible to convert from one data type to another by using functions such as **as.integer()**, **as.vector()**, **as.matrix()** etc.

```
> numbers <- pi
> class(numbers)
[1] "numeric"
> integers <- as.integer(numbers)
> class(integers)
[1] "integer"
> vector <- as.vector(integers)
> class(vector)
[1] "integer"
> matrix <- as.matrix(vector)
> class(matrix)
[1] "matrix"
```

As you can see, pi has been converted from a number to an integer to a vector to a matrix using various R functions.

Basics of R

Before you take a deep dive into learning R, here are some other basic tips, including some initial steps to take in case you encounter an error.

- Environment: The R environment is current workspace. You can think of it like the memory of all assigned values, which you can draw on as you work. For example, if 33 is assigned to black ("black<-33"), then "black" now exists within the R environment with a value of 33, and if the environment is cleared, then "black" will no longer have that value. You can view all of the assigned values using the **ls()** function.
- Functions: As previously described, functions perform various tasks in R, and as simple as they may seem on the surface, each function has a number of arguments and other details, which can be found on the CRAN website. Arguments are the most important because they tell you what information to add to improve the output of the function (type "?read.csv" into R for a good example, no quotations marks).

- Printing: Print is R jargon for viewing something; it gets 'printed' to the workspace. In order to print a dataset, you only need to type its name. For example if 67 is assigned to blue ("blue<-67"), then blue must be printed to see its assigned value.

```
> blue<-67
> blue
[1] 67
```

  R does not automatically display the inputted information because you may be running multiple commands before you want to view the information; therefore, you must print if you want to immediately see the data.
- There is no undo. Every time a command or script is run there is no way of undoing it. For this reason, you may choose to write your script in the editor window before running it.
- Errors: when facing an error, always double-check your script for mistakes and that R is set to the correct working directory and all necessary libraries are loaded. If all of that is fine, there are several other options for troubleshooting: 1) search the error on Google and the R internet forums, 2) ensure that the functions you are using are the best ones for what you want to accomplish, and 3) that all of the arguments associated with the functions have been properly defined. If none of this solves your error, the best option is to provide a detailed description of what you were trying to do and post it onto an R forum.

Here are some operators you may encounter.

?	One of the most important operators in R because this is the equivalent of a help button. When used in combination with a function, such as "?read.csv", a webpage on the CRAN will open explaining what the function does, what its arguments are, and what are some associated functions.
<-	This operator assigns the value on the right side to that of the left, so "five<-5" means that the numerical value 5 is assigned to the character value of five. Therefore, inputting "five+5" would result in 10.
$	To access one variable in a dataset, use the dollar sign "$". For example, ont14$vote1 returns the vote1 variable (the vote1 column).
""	All information put between quotation marks must be literal because R will search for those exact characters.

#	You might find this operator in at the start of a command line in CRAN files or any descriptions of R functions. This tells R not to run that line and to move on to the next, so it is a way to provide line-by-line commentary without interrupting R's ability to run the script.
	`> five<-5`
	`> #we have assigned 5 to five`
	`> six<-6`
	`> five + six`
	`[1] 11`

c()	As seen in the vector example on the previous page, **c()**, which stands for concatenate, will combine its arguments, both numbers and words, into a vector.
+	Another form of concatenation used typically in writing long scripts which span multiple lines, so rather than R interpreting each new line as a new command, it reads it all as one single command.
==	A 6boolean (meaning there are two possible values, true or false) operator which ensures that the values on the left side are the exact same as the values on the right, i.e., 5==5 would be true and 5==9 would be false.

Entering Data

Setting and exploring directory

The directory is the place on your computer that is the home for R; therefore, this is where R is saving files and also where R is looking for files. Because R might be using a folder buried deep in your computer's hard drive, there are two ways of finding and setting your working directory. First, you can use **getwd()** to find the current directory and **setwd()** to set the directory to a different path. NOTE: need to use the forward (/) instead of the backward slash (\) for directory paths in R.

1. getwd()
2. setwd("U:/STAFF/NM/R Guide")

The second way is by going under **File** on the menu bar and going down to **Change dir**. This way is best if you do not know the exact name and location of where you want to set your new working directory because it allows you to go through all of the files on your computer.

Reading CSV Files

Use **read.csv()** to read csv files. Set the header option to true if the file has column titles and false otherwise. The default option is always true.

1. csv <- read.csv("ont14.csv", header=TRUE)
2. summary(csv)

*Data: ont14.csv

Reading Excel Files

There are different methods to read excel worksheets. Method 1 involves copying the data set from excel and using **read.table()** to read the data set that is in clipboard. This is the best option for R versions 3.0.2 or older. Methods 2 and 3 require R version 3.0.3 or later.

Method 1

1. excel1 <- read.table("clipboard")
2. excel1

Method 2

1. install.packages('xlsx')
2. require(xlsx)
3. excel2 <- read.xlsx("excelfile.xlsx", sheetName="Sheet1", header=FALSE)
4. excel2

Method 3

1. install.packages('XLConnect')
2. require(XLConnect)
3. workbook <- loadWorkbook("excelfile.xlsx")
4. excel3 <- readWorksheet(workbook, sheet="Sheet1", header=TRUE)
5. excel3

*Data: excelfile.xlsx

Reading Fixed Files

Use the **read.fwf()** function from the **gdata** library to read fixed format files. Include the width of every variable in the option.

1. require(gdata)
2. fixedfile <- read.fwf("ontfixed.fix", widths=c(14, 24, 2, 15, 2, 2, 10))
3. fixedfile

* Data: ontfixed.fix

Reading Stata Files

Use the **read.dta()** function from the foreign library to read stata data sets. To read Stata 13 data sets, use **read.dta13()** function from the **readstata13** library.

1. library(foreign)
2. dates <- read.dta("dates.dta")
3. dates

* Data: dates.dta

Reading Other Types

The **read.table()** function is useful to read data sets that are in table format and create a data frame. It has options for header, delimiter (sep), skip (lines to skip before reading data) etc. The **scan()** function is also available, and it reads data into a vector or a list from a file.

Entering Data Manually

To enter data manually, create a vector of data for each variable or observation. Combine the vectors as columns of variables using **cbind()** or rows of observations using **rbind()** into a matrix. The data can be changed into a data frame using the **as.data.frame()** function or kept as a matrix. Note that we create a vector for variable here.

Entering this:

```
name = c("John", "Xu", "Aisha")
age = c(10, 15, 24)
gender = c("male", "female", "female")
matrixdata <- cbind(name, age, gender)
matrixdata
```

Gives us this:

```
     name     age  gender
[1,] "John"   "10" "male"
[2,] "Xu"     "15" "female"
[3,] "Aisha"  "24" "female"
```

Now that the information has been combined in a more easily interpreted way, it could be helpful to change it into a data frame. What you want to do determines whether you leave it as a matrix or change it to a data frame. This is how to change to data frame:

```
data <- as.data.frame(matrixdata)
data
```

Voilà!

```
    name age gender
1   John  10   male
2     Xu  15 female
3 Aisha  24 female
```

As you can see, a data frame is a visually cleaner way of working with the information.

Troubleshooting Tips:

- Check that the file you want is in your working directory; if it is not, either move the file to that folder or change your working directory
- Double-check the arguments associated with the function you are using to load the file; you may need to provide additional information in order to help R load the file
- Ensure that you are using the correct R function to load that file type, i.e., if you are loading Stata data, make sure you are using the **read.dta()** function

Exploring Data

To begin, open the corresponding data files for this section. You can do this using the **read.csv()** method discussed above; you will also want to create a name for this data file in R.

```
ont14<- read.csv("ont14.csv", header=TRUE)
```

Unless you opened the .csv file beforehand, you don't know much about the information you just loaded into R. To find out more, use the **dim()** function to find out the dimension of a data set. To view to contents of a data set, use the **ls.str()** function. The **class()** function returns the data type. For example, our data set ont14 is a data frame. Alternatively, the **str()** function does all three.

```
> dim(ont14)
[1] 889  29
> ls.str(ont14)
age :  Factor w/ 6 levels "25 to 34","35 to 44",..: 4 4 4 4 4 4 4 4 4 4 ...
ageavg :  int [1:889] 60 60 60 60 60 60 60 60 60 60 ...
children :  int [1:889] 0 0 0 0 0 0 0 0 0 0 ...
chowapp :  int [1:889] 1 0 1 1 1 1 1 1 1 1 ...
chowheard :  int [1:889] 1 1 1 1 1 1 1 1 1 1 ...
city :  Factor w/ 4 levels "Etobicoke or York",..: 2 2 2 2 2 2 2 2 2 2 ...
education :  int [1:889] 3 3 3 2 4 3 1 2 2 2 ...
fordapp :  int [1:889] 0 1 0 0 0 0 0 0 0 0 ...
fordleave :  int [1:889] 1 1 1 1 1 1 1 1 1 1 ...
fordres :  int [1:889] 1 1 1 1 1 1 1 1 1 1 ...
gender :  int [1:889] 2 2 2 2 2 2 2 2 2 2 ...
income :  int [1:889] 5 6 2 2 7 1 3 4 2 2 ...
party2011 :  int [1:889] 2 1 3 2 3 3 2 2 2 3 ...
property :  int [1:889] 1 1 0 1 1 0 1 1 0 0 ...
sokapp :  int [1:889] NA 1 NA 1 NA NA NA 0 1 NA ...
sokheard :  int [1:889] 0 1 0 1 0 0 0 1 1 0 ...
stinapp :  int [1:889] 1 0 1 1 1 1 1 0 1 1 ...
stinheard :  int [1:889] 1 1 1 1 1 1 1 1 1 1 ...
toryapp :  int [1:889] 0 1 0 0 0 1 1 1 1 0 ...
toryheard :  int [1:889] 1 1 1 1 1 1 1 1 1 1 ...
transport :  Factor w/ 5 levels "Another method",..: 4 2 4 3 4 4 3 3 3 4 ...
vote1 :  Factor w/ 6 levels "David Soknacki",..: 5 3 5 5 5 5 3 5 5 5 ...
vote2 :  Factor w/ 6 levels "David Soknacki",..: 2 1 5 4 2 5 5 3 4 4 ...
vote2010 :  Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
vote2011 :  Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
voteford :  Factor w/ 3 levels "Don't know","No",..: 2 2 2 2 2 2 2 2 2 2 ...
votefordafterrehab :  int [1:889] 0 0 0 0 0 0 1 0 0 0 ...
votenot :  Factor w/ 6 levels "David Soknacki",..: 6 5 6 6 2 6 6 6 6 6 ...
weightsforallotherqs :  num [1:889] 0.283 0.283 0.283 0.283 0.283 ...
> class(ont14)
[1] "data.frame"
```

There are a number of useful functions when it comes to exploring data. Here are some other commands you can use to get a better sense of the data you are working with, whether it's the file you just loaded or anything in the future (for a better understanding of each function, test them out on the ont14 file you just opened).

- **ls()** lists all of the objects available in the workspace, i.e., all the data and variables you have defined
- **str()** displays the structure of an object in a compact way
- **ls.str()** combines the above functions to lists all objects with details about data type and content
- **summary()**, similar to the previous functions, lists the structure and summarizes the variables rather than displaying all of them
- **class()** tells you the data type, i.e., vector, matrix, character, etc.
- **names()** can either change the name of an object or tells you all of the defined names in the object, i.e., all of the column titles in an excel file
- **object.size()** tells you how much memory is taken up on your computer by the object
- **dim()** tells you the dimensions of the object
- **length()** tells you the length (number) of the vectors and factors in the object
- **ncol()** tells you the number of columns
- **nrow()** tells you the number of rows
- **head()** tells you the first six lines of a vector, matrix, table, data frame, or function
- **tail()**tells you the last six lines of a vector, matrix, table, data frame, or function

If you tried all of those on the on14 dataset, you will notice that some of the functions produce overlapping information, so using each of these would be highly repetitive. However, some of these functions can be used to modify the data, so that is where the true utility of the functions comes into play. For more detailed information, you can always check the help page, which is done by placing a '?' before the name of the function.

To open or, in R terminology, print the content of a data set or a variable in the R-console, simply write the data set or the variable. A data set has 2 dimensions. The first dimension is the row number and the second dimension is the column number. The two dimensions are separated by a comma. For example, ont14[1,2] prints the value in the first row and second column of the ont14 data set, ont14[1,] prints the first row and ont14[,2] print the second column. As you may notice, we use square brackets when isolating data and round brackets when working with functions. To print a range of rows or a range of columns, indicate the range separated by a colon.

```
ont14[1,2]
ont14[1,]
ont14[,2]
ont14[1:10,]
ont14[1:10,1]
ont14[1:10,1:3]
```

```
> ont14[1:10,1:3]
          vote1           vote2     votenot
1  Olivia Chow     Don't know    Rob Ford
2    John Tory David Soknacki Olivia Chow
3  Olivia Chow    Olivia Chow    Rob Ford
4  Olivia Chow   Karen Stintz    Rob Ford
5  Olivia Chow     Don't know   John Tory
6  Olivia Chow    Olivia Chow    Rob Ford
7    John Tory    Olivia Chow    Rob Ford
8  Olivia Chow      John Tory    Rob Ford
9  Olivia Chow   Karen Stintz    Rob Ford
10 Olivia Chow   Karen Stintz    Rob Ford
```

Remember to access one variable, use the dollar sign "$". For example, ont14$vote1 returns the vote1 variable. The **attach()** function makes the variables of a data set available in the workspace. Therefore, one can access the variable vote1 by simply writing it instead of using the dollar sign. To release the variables from the workspace, use the **detach()** function.

```
ont14$vote1
attach(ont14)
vote1[1:3]   #The first votes of first three subjects
detach(ont14)   #releases the names
vote1[1:3]   #Variable not available
```

Descriptive Statistics

The **summary()** gives a summary of the object. If the object is a data set, it gives a summary of all the variables in a data set and if it is a variable, it gives a summary of the variable. Other descriptions can be obtained using the **fivenum()**, **min()**, **mean()**, **max()**, **var()**, **quantile()** functions.

Note: always remember to use the **attach()** function at the start of each section to ensure that the variables are in the workspace.

attach(ont14)
summary(toryheard)

```
> summary(toryheard)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  1.0000  1.0000  0.9483  1.0000  1.0000
```

summary(vote1)
torynotheard<-subset(ont14, toryheard==0)
summary(torynotheard$sokheard)

```
> summary(torynotheard$sokheard)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.0000  0.0000  0.1739  0.0000  1.0000
```

fivenum(sokheard)

```
> fivenum(sokheard)
[1] 0 0 1 1 1
```

min(income)
mean(income)
max(income)
var(income)
quantile(income)

```
> quantile(income)
  0%  25%  50%  75% 100%
   1    3    5    7    8
```

Frequency Tables

There are different methods to obtain frequency tables. Remember to install and load a package before using a function from a specific external library. Every table displays the information differently; therefore, each has a different time and place for which it is best suited. Try each out and think about which you liked best and what the positives and negatives are to each table.

<u>Method1</u>
table(vote1)
table(vote1, vote2)

```
> table(vote1, vote2)
                vote2
vote1            David Soknacki Don't know John Tory Karen Stintz Olivia Chow Rob Ford
  David Soknacki             10          4         4            3           9         3
  Don't know                  1         32         5            3           4         4
  John Tory                  33         40        43           70          90        38
  Karen Stintz                3          6        12            7           8         3
  Olivia Chow                43         34       101           55          32         6
  Rob Ford                    8         25        52            9          15        74
```

<u>Method2</u>
install.packages('Hmisc')
library(Hmisc)

describe(vote1)

```
> describe(vote1)
vote1
       n missing  unique
     889       0       6

David Soknacki (33, 4%), Don't know (49, 6%), John Tory (314, 35%)
Karen Stintz (39, 4%), Olivia Chow (271, 30%), Rob Ford (183, 21%)
```

Method3

install.packages('plyr')
library(plyr)
count(vote1, 'vote1')

```
> count(vote1, 'vote1')
           vote1 freq
1 David Soknacki   33
2     Don't know   49
3      John Tory  314
4   Karen Stintz   39
5    Olivia Chow  271
6       Rob Ford  183
```

Method4

install.packages('gmodels')
library(gmodels)
CrossTable(vote1, vote2)

```
> CrossTable(vote1,vote2)

   Cell Contents
|-------------------------|
|                       N |
| Chi-square contribution |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|

Total Observations in Table:  889
```

vote1	vote2 David Soknacki	Don't know	John Tory	Karen Stintz	Olivia Chow	Rob Ford	Row Total
David Soknacki	10	4	4	3	9	3	33
	11.127	0.291	2.041	1.106	1.676	0.646	
	0.303	0.121	0.121	0.091	0.273	0.091	0.037
	0.102	0.028	0.018	0.020	0.057	0.023	
	0.011	0.004	0.004	0.003	0.010	0.003	
Don't know	1	32	5	3	4	4	49
	3.587	75.533	4.051	3.213	2.546	1.323	
	0.020	0.653	0.102	0.061	0.082	0.082	0.055
	0.010	0.227	0.023	0.020	0.025	0.031	
	0.001	0.036	0.006	0.003	0.004	0.004	
John Tory	33	40	43	70	90	38	314
	0.075	1.929	14.770	6.295	20.951	1.150	
	0.105	0.127	0.137	0.223	0.287	0.121	0.353
	0.337	0.284	0.198	0.476	0.570	0.297	
	0.037	0.045	0.048	0.079	0.101	0.043	
Karen Stintz	3	6	12	7	8	3	39
	0.393	0.006	0.646	0.047	0.165	1.218	
	0.077	0.154	0.308	0.179	0.205	0.077	0.044
	0.031	0.043	0.055	0.048	0.051	0.023	
	0.003	0.007	0.013	0.008	0.009	0.003	
Olivia Chow	43	34	101	55	32	6	271
	5.767	1.877	18.361	2.317	5.425	27.942	
	0.159	0.125	0.373	0.203	0.118	0.022	0.305
	0.439	0.241	0.465	0.374	0.203	0.047	
	0.048	0.038	0.114	0.062	0.036	0.007	
Rob Ford	8	25	52	9	15	74	183
	7.346	0.558	1.203	14.937	9.442	86.177	
	0.044	0.137	0.284	0.049	0.082	0.404	0.206
	0.082	0.177	0.240	0.061	0.095	0.578	
	0.009	0.028	0.058	0.010	0.017	0.083	
Column Total	98	141	217	147	158	128	889
	0.110	0.159	0.244	0.165	0.178	0.144	

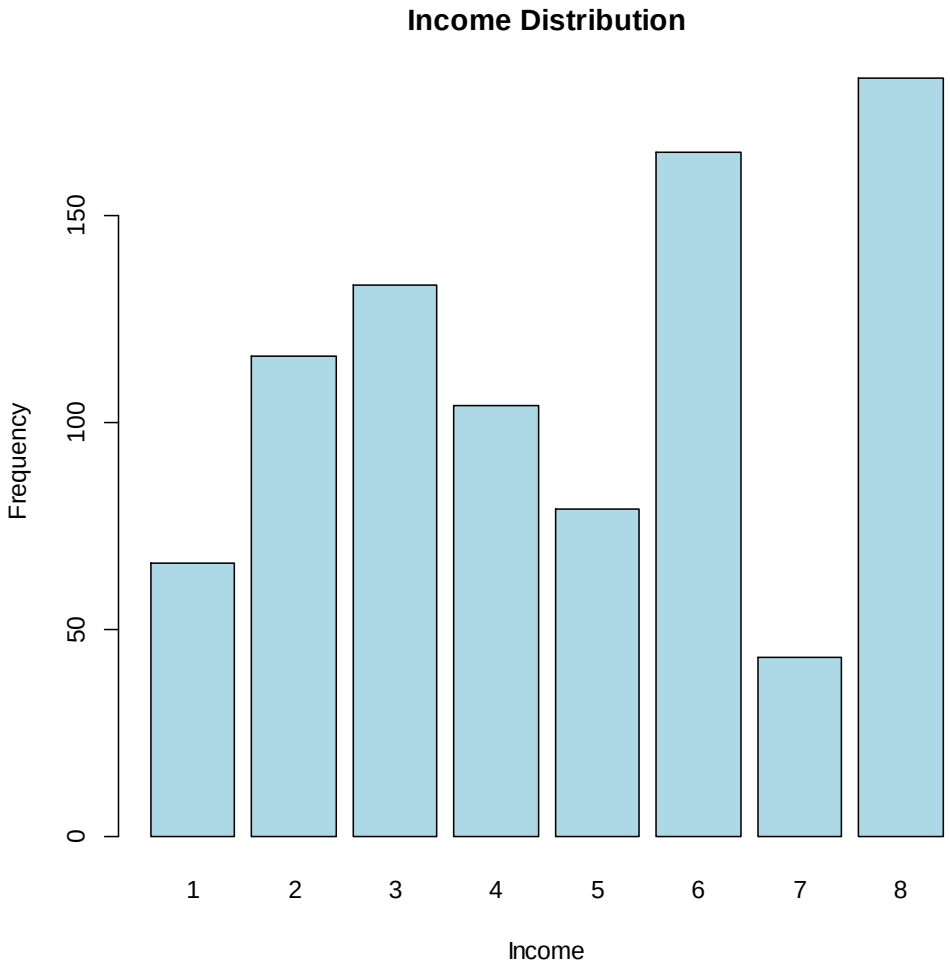
Graphs

The following codes can be used to make bar charts, pie charts, boxplots and scatterplots. These are just a few of the many data visualizations you can produce using R. Each example shows you how to add more information to better develop the data visualization, so the below images are made using the last code in the entry.

Bar Chart

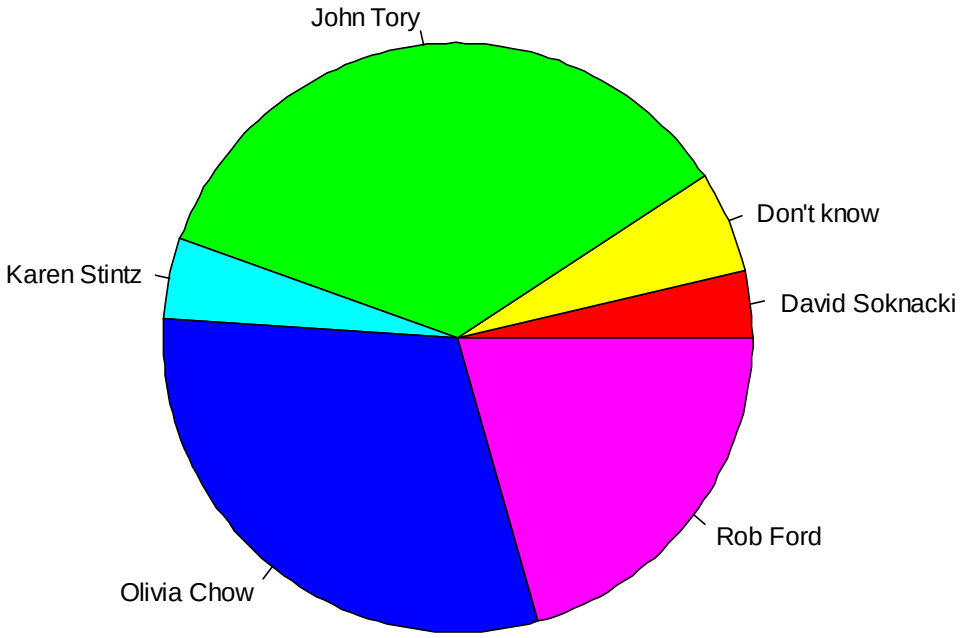
```
barplot(table(income))
barplot(table(income), col="lightblue")
```

```
barplot(table(income), col="lightblue", main="Income Distribution", xlab="Income",
ylab="Frequency")
```



Pie

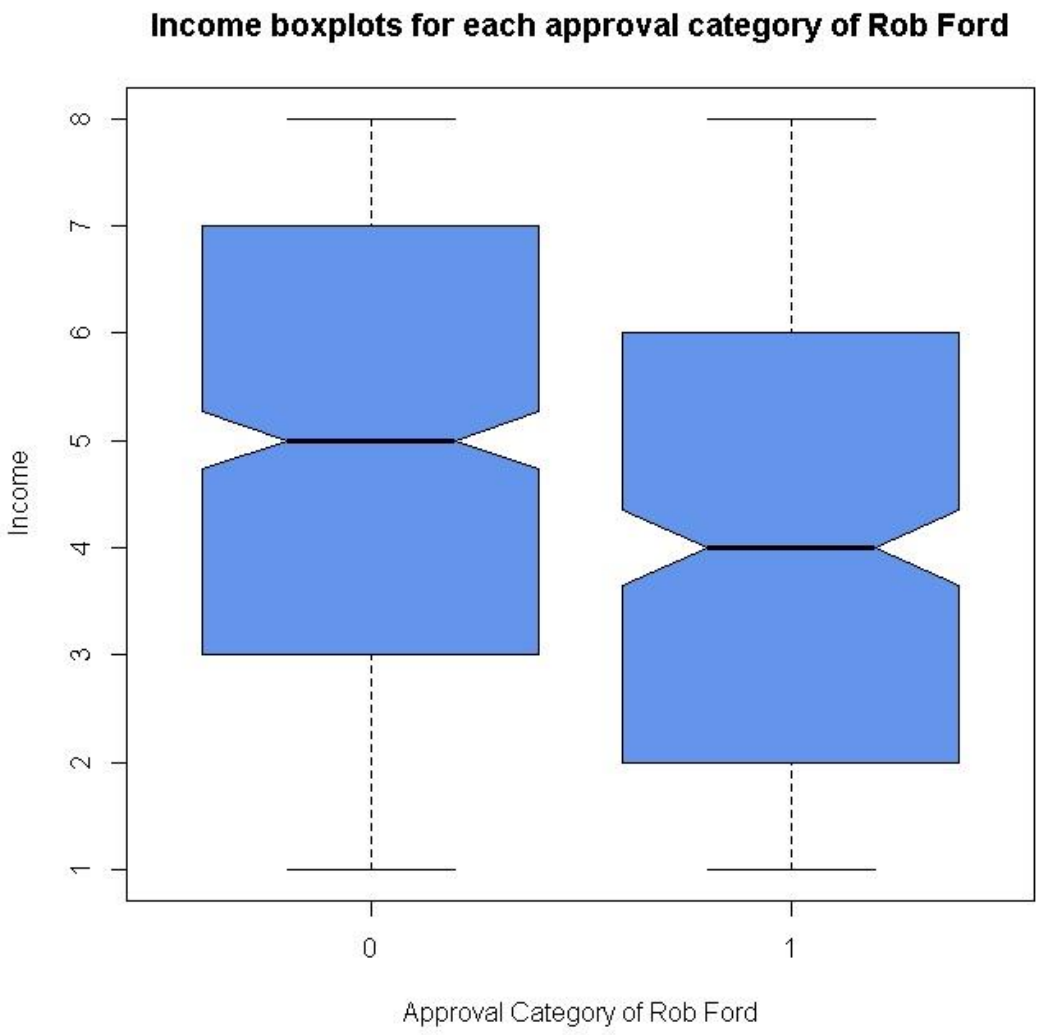
```
pie(table(vote1))
pie(table(vote1), main="First Vote distribution")
pie(table(vote1), main="First Vote distribution", col=rainbow(6))
```



Box Plot

```
boxplot(income)
boxplot(income~fordapp, col="cornflowerblue")
```

boxplot(income~fordapp, col="cornflowerblue", main="Income boxplots for each approval category of Rob Ford", xlab="Approval Category of Rob Ford", ylab="Income", notch=T, label=c("No", "Yes"))



Scatterplot

plot(ageavg, income, bg="lightblue")
plot(ageavg, income, pch=1, cin=0.01)

```
plot(jitter(ageavg, 20), jitter(income, 20))
plot(jitter(ageavg, 20), jitter(income, 20), pch=18, col="red")
plot(jitter(ageavg, 20), jitter(income, 20), pch=18, col="red", main="Scatterplot of Average Age
by Income", xlab="Income", ylab="Average Age")
```

Modifying Data

Dropping & Adding Variables

This ability is useful when the dataset includes information you do not need or if you are trying to reorder the variables. The functions **ls()** and **ls.str()** list the objects that are available in the workspace. To remove a variable from the workspace, use **rm()** or the equivalent **remove()** function.

To remove a variable from the workspace:

```
ls()
```

rm(gender2) #or remove() can also remove datasets
ls()

```
> ls()
[1] "female"      "fordapp2"    "gender2"     "neworder"
[5] "ont14"       "partyfactor" "vote2010a"   "votefordnum"
> rm(gender2)
> ls()
[1] "female"      "fordapp2"    "neworder"    "ont14"
[5] "partyfactor" "vote2010a"   "votefordnum"
```

To remove all objects from the workspace:

rm(list=ls())

To remove a variable from a data frame, use the methods below. Method 1 excludes the variable in question in the creation of a new data set. Method 2 deletes the variable in the same data frame. As you can see, there are now 28 variables in the new data sets instead of 29.

Method 1: Exclude

which(colnames(ont14)=="weightsforallotherqs")

```
> which(colnames(ont14)=="weightsforallotherqs")
[1] 29
```

newont14 <-ont14[c(-29)]
ls.str(newont14)
dim(newont14)

```
> dim(newont14)
[1] 889  28
```

Method 2: Delete

ont14$weightsforallotherqs<-NULL
dim(ont14)

```
> dim(ont14)
[1] 889  28
```

To remove an existing label, use the **factor()** function to replace the label:

vote2010a<-factor(vote2010, levels=c("Yes", "No"), labels=c("Voted in 2010 Municipal Election", "Did not vote in 2010 Municipal Election"))
vote2010b<-factor(vote2010a, levels=c("Voted in 2010 Municipal Election", "Did not vote in 2010 Municipal Election"), labels=c("Yes", "No"))
table(vote2010)

```
> table(vote2010)
vote2010
 No Yes
 93 796
```

table(vote2010a)

```
> table(vote2010a)
vote2010a
       Voted in 2010 Municipal Election
                                    796
Did not vote in 2010 Municipal Election
                                     93
```

table(vote2010b)

```
> table(vote2010b)
vote2010b
Yes  No
796  93
```

To add a variable to a data frame:

Female<-c(1:889)
ont14$female<- female
ls()

```
> female<-c(1:889)
> ont14$female<-female
> ls(ont14)
 [1] "age"                 "ageavg"              "children"
 [4] "chowapp"             "chowheard"           "city"
 [7] "education"           "female"              "fordapp"
[10] "fordleave"           "fordres"             "gender"
[13] "income"              "party2011"           "property"
[16] "sokapp"              "sokheard"            "stinapp"
[19] "stinheard"           "toryapp"             "toryheard"
[22] "transport"           "vote1"               "vote2"
[25] "vote2010"            "vote2011"            "voteford"
[28] "votefordafterrehab"  "votenot"             "weightsforallotherqs"
```

For clarification, you must first assign information to the variable (in this case, the numbers 1 through to 889) and then add that variable to the existing dataset. If you do not create the variable first, then this command script will return an error. There are other methods of adding variables to a data frame, but this is the simplest.

Ordering Data

To order data, create a new data frame with the order of choice. This method can also be used to select variables as well.

neworder <- as.data.frame(cbind(vote1, vote2, votenot, fordapp, voteford,
fordleave, fordres, votefordafterrehab))
ls.str(neworder)

```
> neworder <- as.data.frame(cbind(vote1, vote2, votenot, fordapp, voteford,
+ fordleave, fordres, votefordafterrehab))
> ls.str(neworder)
fordapp :  int [1:889] 0 1 0 0 0 0 0 0 0 0 ...
fordleave :  int [1:889] 1 1 1 1 1 1 1 1 1 1 ...
fordres :  int [1:889] 1 1 1 1 1 1 1 1 1 1 ...
vote1 :  int [1:889] 5 3 5 5 5 5 3 5 5 5 ...
vote2 :  int [1:889] 2 1 5 4 2 5 5 3 4 4 ...
voteford :  int [1:889] 2 2 2 2 2 2 2 2 2 2 ...
votefordafterrehab :  int [1:889] 0 0 0 0 0 0 1 0 0 0 ...
votenot :  int [1:889] 6 5 6 6 2 6 6 6 6 6 ...
```

Labeling Variables

The following shows how to label variable values in examples 1 and 2 and variables in example 3. To label the values of a categorical variable, use the **factor()** function. To label variables or data sets, use the **label()** function from the Hmisc package.

Example 1

```
Library('Hmisc')
fordapp2<-factor(fordapp, levels=c(0,1), labels=c("No", "Yes"))
table(fordapp)
table(fordapp2)
> table(fordapp)
fordapp
  0   1
559 330
> table(fordapp2)
fordapp2
 No Yes
559 330
label(fordapp2) <- "Approval of Rob Ford"
describe(fordapp)
describe(fordapp2)
> describe(fordapp)
fordapp
       n missing  unique    Info     Sum    Mean
     889       0       2     0.7     330  0.3712
> describe(fordapp2)
fordapp2 : Approval of Rob Ford
       n missing  unique
     889       0       2

No (559, 63%), Yes (330, 37%)
```

Example 2

```
vote2010a<-factor(vote2010, levels=c("Yes","No"), labels=c("Voted in 2010 Municipal Election",
"Did not vote in 2010 Municipal Election"))
table(vote2010)
table(vote2010a)
> table(vote2010)
vote2010
 No Yes
 93 796
> table(vote2010a)
vote2010a
       Voted in 2010 Municipal Election Did not vote in 2010 Municipal Election
                                    796                                      93
```

Example 3

```
partyfactor <-factor(party2011, levels=c(1, 2, 3, 4, 5),
labels=c("Progressive Conservative", "Liberal", "NDP", "Green", "Other parties"))
table(party2011)
table(partyfactor)
```

```
> table(party2011)
party2011
  1   2   3   4   5
230 378 147  28  24
> table(partyfactor)
partyfactor
Progressive Conservative                 Liberal                     NDP
                    230                     378                     147
                  Green           Other parties
                     28                      24
```

Recoding

The following examples show how to recode from factor to numeric, from numeric to factor and between numeric variables. To recode, use the **recode()** function from the car package. Now, you may encounter an interesting problem. Because **recode()** is also a function in the Hmisc package previously used in the Labeling Variables section, it may not do what it is intended for in this section, so in this case, **recode()** will accomplish the same function in the car package. This is a good reminder to always look up a new function to learn both the arguments and idiosyncrasies of it.

Example 1

```
votefordnum <- recode(voteford, '"Don\'t know"=2; "No"=0; "Yes"=1;', as.factor.result=FALSE)
#Use backslash to escape the quotation marks
table(voteford)
table(votefordnum)
```

```
> table(voteford)
voteford
Don't know         No        Yes
        62        619        208
> table(votefordnum)
votefordnum
  0   1   2
619 208  62
```

Example 2

```
gender2 <- recode(gender, '1="Male"; 2="Female";', as.factor.result=TRUE)
table(gender)
table(gender2)
```

```
> table(gender)
gender
  1   2
400 489
> table(gender2)
gender2
Female   Male
   489    400
```

Example 3

```
female <- recode(gender,'1=0; 2=1;', as.factor.result=FALSE)
table(gender)
table(female)
```

```
> table(gender)
gender
  1   2
400 489
> table(female)
female
  0   1
400 489
```

Generating New Variables

To generate a new variable that is a combination of other variables, assign the combination to a new variable name. For example, below, totalapp is the average of candidate approval rates. In the code below, the function of average is assigned to "totalapp". This variable has a been recategorized as an ordinal categorical variable "votepnum" and a nominal categorical variable "votep" using the **recode()** function. The standardized values for totalapp can be found using the **scale()** function or by calculating it. Note that the summary of the standardized values using the two methods, **ztotalapp()** and **ztotalapp2()** are the same.

```
totalapp <- (as.numeric(fordapp) + as.numeric(chowapp) + as.numeric(toryapp) +
as.numeric(stinapp) + as.numeric(sokapp))/5
summary(totalapp)
```

```
> totalapp <- (as.numeric(fordapp) + as.numeric(chowapp) + as.numeric(toryapp)
+ + as.numeric(stinapp) + as.numeric(sokapp))/5
> summary(totalapp)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 0.0000  0.4000  0.6000  0.5099  0.6000  1.0000     334
```

```
library('car')
votepnum <- recode(totalapp,'0:0.33=0; 0.331:0.66=1; 0.661:1=2', as.factor.result=TRUE)
label(votepnum) <-"Voting Personality (numeric)"
votep<- recode(totalapp,'0:0.33="Strict"; 0.331:0.66="Average"; 0.661:1="Lenient"',
as.factor.result=TRUE)
label(votep) <-"Voting Personality"
describe(votepnum)
describe(votep)
```

```
> describe(votepnum)
votepnum : Voting Personality (numeric)
      n missing  unique
    555     334       3

0 (103, 19%), 1 (326, 59%), 2 (126, 23%)
> describe(votep)
votep : Voting Personality
      n missing  unique
    555     334       3

Average (326, 59%), Lenient (126, 23%), Strict (103, 19%)
```

```
ztotalapp <- scale(totalapp, center=TRUE, scale=TRUE)
ztotalapp2 <-(totalapp - mean(totalapp, na.rm=TRUE))/sd(totalapp, na.rm=TRUE)
```

```
summary(ztotalapp)
summary(ztotalapp2)
```

```
> summary(ztotalapp)
       V1
 Min.   :-2.4035
 1st Qu.:-0.5181
 Median : 0.4246
 Mean   : 0.0000
 3rd Qu.: 0.4246
 Max.   : 2.3101
 NA's   :334
> summary(ztotalapp2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
-2.4030 -0.5181  0.4246  0.0000  0.4246  2.3100     334
```

The **by()** function can be used to view the average candidate approval rates by another category. For example, below, we can see the average approval rates by income category.

```
ont14$totalapp<-totalapp
by(ont14$totalapp, ont14$income, function(object) mean(x=object, na.rm=TRUE))
```

```
> by(ont14$totalapp, ont14$income, function(object) mean(x=object, na.rm=TRUE))
ont14$income: 1
[1] 0.4470588
------------------------------------------------------------
ont14$income: 2
[1] 0.4923077
------------------------------------------------------------
ont14$income: 3
[1] 0.5064935
------------------------------------------------------------
ont14$income: 4
[1] 0.4925373
------------------------------------------------------------
ont14$income: 5
[1] 0.5137931
------------------------------------------------------------
ont14$income: 6
[1] 0.5421488
------------------------------------------------------------
ont14$income: 7
[1] 0.5
------------------------------------------------------------
ont14$income: 8
[1] 0.5083333
```

```
activity<-(as.numeric(fordapp) + as.numeric(chowapp) + as.numeric(toryapp)
+ as.numeric(stinapp) + as.numeric(sokapp) + as.numeric(chowheard) + as.numeric(toryheard)
+ as.numeric(stinheard) + as.numeric(sokheard))/9
activity<-round(activity, 2)
table(activity, income)
```

```
> activity<-round(activity, 2)
> table(activity, income)
        income
activity  1  2  3  4  5  6  7  8
    0.44  0  1  1  1  0  0  0  0
    0.56  5 15 12 16 15 10  4 23
    0.67  7 20 25 15 10 43 12 39
    0.78  2 13 23 21 18 41  9 28
    0.89  2 14 16 14 15 26  5 30
    1     1  2  0  0  0  1  0  0
```

Managing Data

Generating ID Variable

id<-seq_len(nrow(ont14))
ont14$id<-id

Subsetting Data

To subset, use the **subset()** function. We want to subset, for example, the female voters with the highest income on one hand and the male voters with the lowest education on the other hand. These data sets are saved as subset1 and subset2 respectively.

Example 1

subset1<-subset(ont14, income==7 & gender==2)
dim(subset1)

```
> dim(subset1)
[1] 13 30
```

Example 2

subset2<-subset(ont14, education==4 & gender==1)
dim(subset2)

```
> dim(subset2)
[1] 101  30
```

Appending Data

To append use the **rbind()** function that stacks similar data types. The number and order of variables between the two data sets must match in order for the rbind() function to be effective. Below, we append subset1 and subset2 to create the new data frame appenddata.

appenddata<- rbind(subset1, subset2)

Keeping & Dropping Variables

Managing data can involve keeping or dropping specific variables. For example, in example 3 below, only ford related variables are used to create a new data frame “subset3”. In example 4, we drop the variables in the first columns until column “votefordafterrehab” in order to keep the variables that provide sample information in “subset4”.

Example 1

subset3<- as.data.frame(cbind(id, fordapp, voteford, fordleave, fordres, votefordafterrehab))

dim(subset3)

```
> dim(subset3)
[1] 889   6
```

Example 2

describe(ont14)
which(colnames(ont14)=="votefordafterrehab")
subset4<-ont14[-1:-16]
dim(subset4)

```
> which(colnames(ont14)=="votefordafterrehab")
[1] 16
> subset4<-ont14[-1:-16]
> dim(subset4)
[1] 889  14
```

Merging Data

Because R does not include an undo ability, if you wish to combine data you have previously subsetted or specific data you want in a combined data frame, the **merge()** function will accomplish this.

mergedata<-merge(subset3, subset4, by="id")

Reshaping Data

Read Data

If you need to reshape your data, you can either edit it in your original format (be it SPSS, Stata, Excel, etc.) or, if you have already worked on it in R and do not want to redo that work, then there are ways to reshape datasets using R. The reshape package offers a variety of other ways of massaging data, which are not covered in this section, including the **t()**, **melt()**, and **cast()** functions.

wide <- read.csv("wide.csv", header=T)
wide

```
  id drug1 drug2 drug3 age sleep1 sleep2
1  1     1     1     1  12      5      6
2  2     0     1     0  25      8      9
3  3     1     0     1  32      7      5
4  4     0     0     1  19      9      8
```

Wide to Long

long<-reshape(wide, varying=c("drug1", "drug2", "drug3"), v.names="drug_yn",
+ timevar="drugtype", times=c("1", "2", "3"), new.row.names = 1:12, direction="long")
long

```
   id age score1 score2 drugtype drug_yn
1   1  12      5      6        1       1
2   2  25      8      9        1       0
3   3  32      7      5        1       1
4   4  19      9      8        1       0
5   1  12      5      6        2       1
6   2  25      8      9        2       1
7   3  32      7      5        2       0
8   4  19      9      8        2       0
9   1  12      5      6        3       1
10  2  25      8      9        3       0
11  3  32      7      5        3       1
12  4  19      9      8        3       1
```

```
longsort<-long[order(long$id),]
by(long$drug_yn, long$drugtype, function(object) mean(x=object))
by(long$drug_yn, long$drugtype, function(object) sd(x=object))
cbind(as.vector(by(longsort, list(longsort$drugtype), function(x) mean(longsort$drug_yn))),
as.vector(by(longsort, list(longsort$drugtype), function(x) sd(longsort$drug_yn))))
```

```
          [,1]      [,2]
[1,] 0.5833333 0.5149287
[2,] 0.5833333 0.5149287
[3,] 0.5833333 0.5149287
```

Long to Wide

```
wide2<-reshape(longsort, timevar="drugtype", idvar=c("id", "age", "score1", "score2"),
direction="wide")
wide2
```

```
  id age score1 score2 drug_yn.1 drug_yn.2 drug_yn.3
1  1  12      5      6         1         1         1
2  2  25      8      9         0         1         0
3  3  32      7      5         1         0         1
4  4  19      9      8         0         0         1
```

Counting Data

Highest and Lowest Values in a Group

Here, we want to find the voters with the lowest and highest education level for each city by age group. First, we sort by all three variables (city, age and, education) from lowest to highest. In this case, we also sort by id because some groups with the same city, age and education values have more than one subject in them. Then, we use the **aggregate()** function to find the first individual with the highest and lowest education level in each city by age group.

Generating ID variable

```
id<-seq_len(nrow(ont14))
ont14$id<-id
```

Sorting the groups

```
ont14o<-ont14[order(ont14$city, ont14$age, ont14$education, ont14$id),]
```

Creating variables sorted by id

```
idsort<-seq_len(nrow(ont14o))
ont14o$idsort<-idsort
```

Separating out the lowest and highest list in each group

```
lowest<-aggregate(ont14o, by=list(ont14o$age, ont14o$city), head, 1)
highest<-aggregate(ont14o, by=list(ont14o$age, ont14o$city), tail, 1)
```

Checking for duplicates, this will return TRUE for no duplicates

```
length(unique(ont14$id)) == nrow(ont14)
```

Voilà, here are the relevant variables.

```
lowest[1:10 ,c(1:5, 28,32,33)]
```

```
> lowest[1:10 ,c(1:5, 28,32,33)]
          Group.1                Group.2          vote1        vote2        votenot education  id idsort
1        25 to 34      Etobicoke or York     Don't know    John Tory    Olivia Chow         1 871      1
2        35 to 44      Etobicoke or York       Rob Ford     Rob Ford    Olivia Chow         1 823      7
3        45 to 54      Etobicoke or York       Rob Ford    John Tory    Olivia Chow         1 711     23
4        55 to 64      Etobicoke or York      John Tory     Rob Ford   Karen Stintz         1 273     60
5     65 and over      Etobicoke or York      John Tory  Olivia Chow       Rob Ford         1 500    114
6  Under 25 years      Etobicoke or York       Rob Ford    John Tory    Olivia Chow         2 867    168
7        25 to 34 Former city of Toronto    Olivia Chow Karen Stintz      John Tory         1 848    171
8        35 to 44 Former city of Toronto David Soknacki   Don't know David Soknacki         1 762    181
9        45 to 54 Former city of Toronto David Soknacki     Rob Ford    Olivia Chow         1 595    230
10       55 to 64 Former city of Toronto      John Tory  Olivia Chow       Rob Ford         1   7    288
```

```
highest[1:10 ,c(1:5, 28,32,33)]
```

```
> highest[1:10 ,c(1:5, 28,32,33)]
          Group.1                Group.2        vote1        vote2        votenot education  id idsort
1        25 to 34      Etobicoke or York    John Tory     Rob Ford    Olivia Chow         3 885      6
2        35 to 44      Etobicoke or York     Rob Ford    John Tory    Olivia Chow         4 833     22
3        45 to 54      Etobicoke or York  Olivia Chow Karen Stintz       Rob Ford         4 729     59
4        55 to 64      Etobicoke or York     John Tory   Don't know      Rob Ford         4 570    113
5     65 and over      Etobicoke or York     Rob Ford     Rob Ford    Olivia Chow         4 550    167
6  Under 25 years      Etobicoke or York  Olivia Chow  Olivia Chow       Rob Ford         4 889    170
7        25 to 34 Former city of Toronto Karen Stintz Karen Stintz   Karen Stintz         4 864    180
8        35 to 44 Former city of Toronto  Olivia Chow Karen Stintz       Rob Ford         4 786    229
9        45 to 54 Former city of Toronto     Rob Ford     Rob Ford David Soknacki         4 626    287
10       55 to 64 Former city of Toronto    John Tory Karen Stintz       Rob Ford         4 264    387
```

Dates

Reading Data

In order to work with the dataset for this section, you will need to use the foreign package, which reads and writes data created by some versions of Epi Info, Minitab, S, SAS, SPSS, Stata, Systat and Weka.

```
library(foreign)
dates<- read.dta("dates.dta")
ls.str(dates) #which one are numeric vs char
```

*Data: dates.dta

Date Manipulation

Time in R is counted from January 1, 1970. This means that the number of seconds, hours, days, months, etc. following that date will be expressed as a positive numbers and that the same information preceding that date will be expressed as a negative number. Since R's default is to previous dates as a

character-type number, some modifications will need to be made to modify it to present in the day/month/year format. The following data consists of two dates: one is in string format and the other one is numeric as can be seen with the **ls.str()** function, as done above.

dates

```
> dates
               bdate dmon dday dyear
1          2/24/1968   11    9  1994
2         11-15-1940    4   30  1991
3            1/21/72    9    4  1993
4      July 12, 1983    3   26  1995
5         Dec 2 1944    6   15  1989
6  October 16, 1967     7   12  1990
7    February 1 1940    5    8  1992
8          2/15/1980    4   24  2005
9          7-12-1948   12    3  1991
```

The lubridate package provides functions to identify date-time data, extract components, perform accurate math on date-time variables, handle time zones etc. Below, we use the **mdy()** and **ymd()** functions to convert character-type dates that are in month-date-year and year-month-date formats. Since the variables dmon, dday and dyear are numeric and come separately, to create a date variable, we concatenate the three variable with the **paste()** function before converting it into a date format with the **ymd()** function.

Converting into elapsed days since

```
birthday<-mdy(bdate)
event<-ymd(paste(dyear, "-", dmon, "-", dday, sep=""))
as.data.frame(cbind(birthday, event))
```

```
> birthday<-mdy(bdate)
> event<-ymd(paste(dyear, "-", dmon, "-", dday, sep=""))
> as.data.frame(cbind(birthday, event))
     birthday      event
1    -58492800  784339200
2   -919209600  672969600
3 -59893344000  747100800
4    426816000  796176000
5   -791510400  613872000
6    -69811200  647740800
7   -944092800  705283200
8    319420800 1114300800
9   -677635200  691718400
```

The **format()** function can be used to control the display of elapsed dates.

```
fbirthday<-format(birthday, "%d %b %Y")
fevent<-format(event, "%d %b %Y")
as.data.frame(cbind(fbirthday, fevent))
```

```
> fbirthday<-format(birthday, "%d %b %Y")
> fevent<-format(event, "%d %b %Y")
> as.data.frame(cbind(fbirthday, fevent))
    fbirthday      fevent
1 24 Feb 1968 09 Nov 1994
2 15 Nov 1940 30 Apr 1991
3 21 Jan 0072 04 Sep 1993
4 12 Jul 1983 26 Mar 1995
5 02 Dec 1944 15 Jun 1989
6 16 Oct 1967 12 Jul 1990
7 01 Feb 1940 08 May 1992
8 15 Feb 1980 24 Apr 2005
9 12 Jul 1948 03 Dec 1991
```

Since the third birthday is from year 72CE, we can assume that this is a mistake and requires fixing. This can be done by manually replacing the value of birthday for the 3rd observation.

fbirthday[3]<-"21 Jan 1972"

The number of days elapsed can be converted into weeks, months and years as follows:

diff_days<-event-birthday
weeks <- diff_days/7
months <- diff_days/30.5
years <- diff_days/365.25
as.data.frame(cbind(fbirthday, fevent, diff_days, weeks, months, years))

```
> as.data.frame(cbind(fbirthday, fevent, diff_days, weeks, months, years))
    fbirthday      fevent diff_days            weeks           months            years
1 24 Feb 1968 09 Nov 1994      9755 1393.57142857143  319.83606557377 26.7077344284736
2 15 Nov 1940 30 Apr 1991     18428 2632.57142857143 604.196721311475 50.4531143052704
3 21 Jan 1972 04 Sep 1993    701857 100265.285714286 23011.7049180328 1921.57973990418
4 12 Jul 1983 26 Mar 1995      4275 610.714285714286  140.16393442623 11.7043121149897
5 02 Dec 1944 15 Jun 1989     16266 2323.71428571429 533.311475409836 44.5338809034908
6 16 Oct 1967 12 Jul 1990      8305 1186.42857142857 272.295081967213 22.7378507871321
7 01 Feb 1940 08 May 1992     19090 2727.14285714286 625.901639344262 52.2655715263518
8 15 Feb 1980 24 Apr 2005      9200 1314.28571428571 301.639344262295 25.1882272416153
9 12 Jul 1948 03 Dec 1991     15849 2264.14285714286 519.639344262295 43.3921971252567
```

Note: R has an original function called **as.Date(**[date], [format]**)** that can be used to convert character-type dates to actual dates. However, the lubridate package is more popular because of its ease of usage.