**Foundations of Statistics Using R**

**Module 1: Practice Exercises Working Session**

We will be using R to practice concepts that were learned in the pdf lessons for R. We will build on those fundamental concepts and apply the techniques for sorting, merging and reshaping our data. Next, you

will be introduced to loops, control structures and conditional statements in R.

This will be followed by writing some simple functions to become more efficient and effective in using R. Finally, the session will conclude with some ways to write reports and build dynamic web applications.

You are expected to follow along with the written materials and the activity prompts during the in-class session. This will help guide your work and keep pace with the class exercises. Finally, this document will serve a lesson you can return back to for further study and practice.

This session contains 26 practice exercises (labeled *a* through *z*).

**You’ll be expected to complete all the exercises on your own and submit your R\_practice\_exercies.R file by the end of MODULE 1. Submit on the LMS**

**Session Outline**

1. Sorting vectors

2. Merging data

3. Sorting data frames

4. Reshaping data

5. Iteration using loops

6. Conditional statements

7. Working with text and splitting strings

8. Functions

9. Writing reports

10. Web-Based Applications using Shiny

11. Summary

12. Resources

|  |  |
| --- | --- |
| You will need the following files: | You will need to have the following R packages installed: |
| * R\_in\_class\_session.R * sidewalk\_example\_01.R * sidewalk\_example\_02.Rmd * [world\_internet\_usage.csv](https://newclasses.nyu.edu/access/content/group/a7210e25-c499-4829-90cf-46e7b788ff77/world_internet_usage.csv) * [march\_madness.csv](https://newclasses.nyu.edu/access/content/group/a7210e25-c499-4829-90cf-46e7b788ff77/march_madness.csv) * [Sidewalk\_Cafes.csv](https://newclasses.nyu.edu/access/content/group/a7210e25-c499-4829-90cf-46e7b788ff77/Sidewalk_Cafes.csv) | * reshape2 * ggplot2 * ggthemes * lattice * stringr * knitr * shiny * rsconnect |

**We’ll begin by opening RStudio and opening the R\_practice\_exercies.R file.**

#MODULE 1: iDAF - Essential of Statistics Using R####

#YOUR NAME####

#1. SORTING VECTORS####

#a#### input data

#90,87,69,89,58,93,99,98,76,88

datamininggrades=c(90,87,69,89,58,93,99,98,76,88)

#b####

#c#### input data

#"A-", "B+", "F", "B+", "F", "A", "A", "A", "C+", "B+"

#d####

#e#### input data

#"Strongly Disagree", "Disagree", "Somewhat Disagree","Neither agree or disagree", "Somewhat agree", "Agree", "Strongly Agree")

#2. MERGING DATA TOGETHER####

#f####

#3. SORTING DATA FRAMES####

#g####

#4. RESHAPHING A DATA FRAME####

#h#### input data

#"country", "2000", "2001", "2002", "2003", "2004", "2005", "2006", "2007", "2008", "2009", "2010", "2011", "2012"

#i####

#j####

#starter script

#h1 =histogram(internet$"2000", breaks=8, xlab="Internet Usage", main="2000")

#print(h1, split=c(1,1,4,4), more=T);

#k####

#l####

#m####

#5. ITERATION USING LOOPS####

#n####

#o####

#p####

#6. CONDITIONAL STATEMENTS####

#q####

#r####

#s####

#t####

#7. WORKING WITH TEXT AND SPLITTING STRINGS####

#u####

#8. WRITING FUNCTIONS####

#v####

#w####

#9. WRITING REPORTS####

#x####create a separate .R script

#you do not need to submit your script for this exercise.

#y####

#you do not need to submit your script for this exercise.

#10. SHINY ####

#you do not need to submit your script for this exercise.

#z####

**1. Sorting numeric, character, and factor vectors**

In this section we will discuss the ways to sort a vector or data frame, and combine two vectors together in a matrix and a data frame.

**Sorting a numeric or character vector**

A common data analysis task is to sort a variable. For numeric variables you might want to sort it in ascending or descending order. That is, you may want to sort it from the highest to the lowest values or vice-versa. For character variables you might want to sort the data alphabetically.

You can sort variables using the sort() function.   
  
The usage is: sort(x, decreasing = FALSE, ...)

**Let’s create a simple numeric vector called datamininggrades using the c() function that contains the final numeric grades of sample of 10 students in their data mining course. Sort it from lowest to highest.**

a

**The grades are** 90,87,69,89,58,93,99,98,76,88

> datamininggrades = c(90,87,69,89,58,93,99,98,76,88)

> sort(x=datamininggrades)

[1] 58 69 76 87 88 89 90 93 98 99

By default, you’ll notice that R will sort a numeric vector in ascending order (from lowest to highest).

**Sort the vector in descending order by adding an additional parameter *decreasing* and assign it to TRUE.**

b

> sort(x=datamininggrades, decreasing=TRUE)

[1] 99 98 93 90 89 88 87 76 69 58

**Next, create a simple character vector called lettergrade with the ten following values:** A-", "B+", "F", "B+", "F", "A", "A", "A", "C+", "B+"

c

> lettergrade = c("A-", "B+", "F", "B+", "F", "A", "A", "A", "C+", "B+" )

> sort(x=lettergrade)

[1] "A" "A" "A" "A-" "B+" "B+" "B+" "C+" "F" "F"

> class(lettergrade)

[1] "character

With characters, the sort function sorts alphabetically. Things get more complicated with characters such as + and -.

For example, what if the grades were:   
  
> newgrades=c("A-", "B", "F", "B+", "B", "A", "A", "A", "C+", "B-")

> sort(newgrades)

[1] "A" "A" "A" "A-" "B" "B" "B-" "B+" "C+" "F"

**Sorting a factor variable**

In addition to sorting variables of type numeric and character, we can sort factor variables. There are two ways you can sort a factor variable: by level or alphabetically.

**To demonstrate how to sort factors, we’ll create a factor variable called lettergradefactor using the lettergrade vector we created previously. We use the factor() function to cast lettergrade as a factor.**

d

**>** lettergradefactor = factor(lettergrade)  
> lettergradefactor

[1] A- B+ F B+ F A A A C+ B+

Levels: A A- B+ C+ F

**Next, sort the lettergradefactor variable using the sort() function, just as we did earlier.**

> sort(lettergradefactor)  
[1] A A A A- B+ B+ B+ C+ F F

Levels: A A- B+ C+ F

By default the factor levels are alphabetically ordered. There are cases when they are not. For example, let's return to lesson 6. If we have data that contains Likert-scale responses from Disagree to Strongly Agree, alphabetically sorting is not useful for ordinal variables. That is, variables that have a clear ordering sequence.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Strongly disagree | Disagree | Somewhat disagree | Neither agree or disagree | Somewhat agree | Agree | Strongly agree |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

**Create a vector called agreement that contains the following values: Disagree", "Neither agree or disagree", "Somewhat agree", "Agree", "Strongly Agree".**

e

> #create the variable

> > agreement = c("Strongly Disagree", "Disagree", "Somewhat Disagree","Neither agree or disagree", "Somewhat agree", "Agree", "Strongly Agree")

> #view the variable

> agreement

[1] "Strongly Disagree" "Disagree" "Somewhat Disagree" "Neither agree or disagree"

[5] "Somewhat agree" "Agree" "Strongly Agree"

**Next, convert the vector to a factor variable called agreementfactor.**

>#convert to factor

> agreementfactor = factor(agreement)

> #view factor variable with levels

> agreementfactor

[1] Strongly Disagree Disagree Somewhat Disagree Neither agree or disagree

[5] Somewhat agree Agree Strongly Agree

Levels: Agree Disagree Neither agree or disagree Somewhat agree Somewhat Disagree Strongly Agree Strongly Disagree

**Then, sort the factor variable and note how it sorts the values.**

> sort(agreementfactor)

[1] Agree Disagree Neither agree or disagree Somewhat agree

[5] Somewhat Disagree Strongly Agree Strongly Disagree

Levels: Agree Disagree Neither agree or disagree Somewhat agree Somewhat Disagree Strongly Agree Strongly Disagree

**Instead, let’s define the order of the factor levels using the levels parameter of the factor function.**

The usage is:factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x))

># define levels

> agreementfactor = factor(agreement, levels = c("Strongly Disagree", "Disagree", "Somewhat Disagree","Neither agree or disagree", "Somewhat agree", "Agree", "Strongly Agree"))

> # sort

> sort(agreementfactor)

[1] Strongly Disagree Disagree Somewhat Disagree Neither agree or disagree

[5] Somewhat agree Agree Strongly Agree

Levels: Strongly Disagree Disagree Somewhat Disagree Neither agree or disagree Somewhat agree Agree Strongly Agree

While it initially looked as those R was sorting factor variables alphabetically; this example proves that it sorts factors based on the level order.

**2. Merging data together**

It is quite common to combine several vectors together into one data structure. We can either merge our data together as a matrix or as a data frame.  
  
Let's merge the two vectors datamining grades and lettergradefactor together in one data structure.

**Use the data.frame() function to create a data frame instead of a matrix.**

f

The simplified usage is:   
data.frame(..., row.names = NULL, …

> finalgrades = data.frame(datamininggrades, lettergradefactor)

As you can see from the output below, the data.frame function preserves the native data type of each variable in the data structure.

> finalgrades

datamininggrades lettergradefactor

1 90 A-

2 87 B+

3 69 F

4 89 B+

5 58 F

6 93 A

7 99 A

8 98 A

9 76 C+

10 88 B+

**3. Sorting a data frame**

The sort() function doesn't work on a data frame. Instead, we'll use a function called *order(*) along with a function called *with()*.  
  
The order () usage is: order(..., na.last = TRUE, decreasing = FALSE)

### Arguments

|  |  |
| --- | --- |
| ... | a sequence of numeric, complex, character or logical vectors, all of the same length, or a classed **R** object. |
| X | an atomic vector. |
| partial | vector of indices for partial sorting. (Non-NULL values are not implemented.) |
| decreasing | logical. Should the sort order be increasing or decreasing? |
| na.last | for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed |

The with() usage is**:** with(data, expr, ...) within(data, expr, ...)

### Arguments

|  |  |
| --- | --- |
| data | data to use for constructing an environment. For the default withmethod this may be an environment, a list, a data frame, or an integer as in sys.call. For within, it can be a list or a data frame. |
| expr | expression to evaluate. |
| ... | arguments to be passed to future methods. |

g

**Here’s an example for you to practice that sorts a data frame.**

> finalgrades[with(finalgrades, order(-datamininggrades)), ] #for descending use -

datamininggrades lettergradefactor

7 99 A

8 98 A

6 93 A

1 90 A-

4 89 B+

10 88 B+

2 87 B+

9 76 C+

3 69 F

5 58 F

**For ascending order remove the –**  
> finalgrades[with(finalgrades, order(datamininggrades)), ] #for ascending

datamininggrades lettergradefactor

5 58 F

3 69 F

9 76 C+

2 87 B+

10 88 B+

4 89 B+

1 90 A-

6 93 A

8 98 A

7 99 A

**4. Reshaping or flipping a data frame**

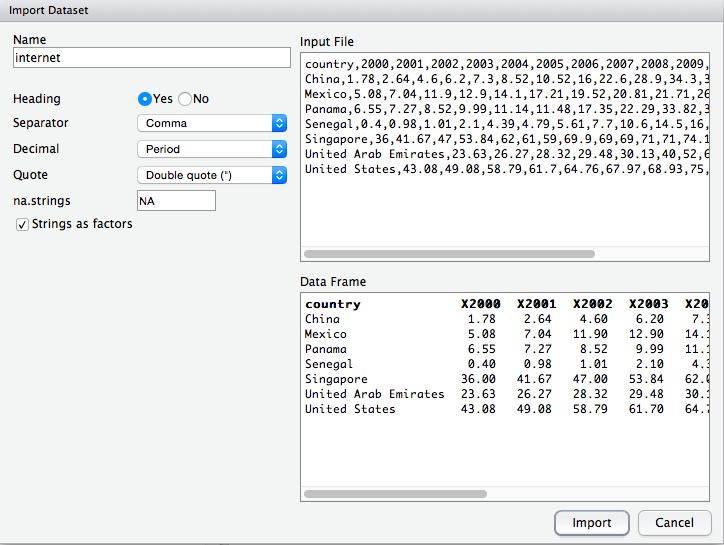
As we learned in session 6, data is messy. A task that you’ll have to undertake is reshaping your data. This means taking the data that you have in whatever format it is given to you, and converting it to the format that you need.

Let’s look at a simple example of World Internet Usage over time for a few countries. Specifically, we’ll look Internet Users per 100 people in the United States, Panama, China, Senegal, Mexico, United Arab Emirates, and Singapore from 2000 to 2012. These are people with access to the worldwide network, according to The World Bank.

I’ve provided you with a version of this file called world\_internet\_usage.csv.

h

**Import the csv file into R and name it *internet*.**

****

As we examine this data file we notice there’s a column for each year that corresponds to each country. The columns all have X’s in front of them. R does not accept numbers as column names. We need to rename our columns with each year in quotes; R treats numbers within quotes as a characters.

**Change the column names using the names() function.**

The usage is:

names(x) <- value

### Arguments

|  |  |
| --- | --- |
| x | an **R** object. |
| value | a character vector of up to the same length as x, or NULL. |

> #view internet

> View(internet)

> #change column names

> names(internet) <-c("country", "2000", "2001", "2002", "2003", "2004", "2005", "2006", "2007", "2008", "2009", "2010", "2011", "2012")

> str(internet)

'data.frame': 7 obs. of 14 variables:

$ country: Factor w/ 7 levels "China","Mexico",..: 1 2 3 4 5 6 7

$ 2000 : num 1.78 5.08 6.55 0.4 36 ...

$ 2001 : num 2.64 7.04 7.27 0.98 41.67 ...

$ 2002 : num 4.6 11.9 8.52 1.01 47 ...

$ 2003 : num 6.2 12.9 9.99 2.1 53.84 ...

$ 2004 : num 7.3 14.1 11.14 4.39 62 ...

$ 2005 : num 8.52 17.21 11.48 4.79 61 ...

$ 2006 : num 10.52 19.52 17.35 5.61 59 ...

$ 2007 : num 16 20.8 22.3 7.7 69.9 ...

$ 2008 : num 22.6 21.7 33.8 10.6 69 ...

$ 2009 : num 28.9 26.3 39.1 14.5 69 ...

$ 2010 : num 34.3 31.1 40.1 16 71 ...

$ 2011 : num 38.3 35 42.7 17.5 71 ...

$ 2012 : num 42.3 38.4 45.2 19.2 74.2 ...

Let's assume we want to quickly plot our data. We want to see how all the countries compare overtime. Our assumption is that there is a general upward trend.

What will be the X variable?

What will be the Y variable?

Right now our data is in the following format:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| country | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Given the format above, we would only be able to select a single year as our x variable. In this format, we can only graph one year at a time using a histogram or to create a histogram matrix to plot a histogram for each year from 2000 through 2012.

i

**Create a single histogram for a single year. Select the year 2000. Use the hist() function.**

> # single histogram for a single year

> hist(internet$"2000", breaks=8, freq=TRUE, main="Histogram of Internet Usage in 2000", xlab = "Internet Usage")

**To plot a matrix of histograms we would use the lattice package. An example is provided below.**

j

> require(lattice)

h1 =histogram(internet$"2000", breaks=8, xlab="Internet Usage", main="2000")

h2 =histogram(internet$"2001", breaks=8, xlab="Internet Usage", main="2001")

h3 =histogram(internet$"2002", breaks=8, xlab="Internet Usage", main="2002")

h4 =histogram(internet$"2003", breaks=8, xlab="Internet Usage", main="2003")

h5 =histogram(internet$"2004", breaks=8, xlab="Internet Usage", main="2004")

h6 =histogram(internet$"2005", breaks=8, xlab="Internet Usage", main="2005")

h7 =histogram(internet$"2006", breaks=8, xlab="Internet Usage", main="2006")

h8 =histogram(internet$"2007", breaks=8, xlab="Internet Usage", main="2007")

h9 =histogram(internet$"2008", breaks=8, xlab="Internet Usage", main="2008")

h10 =histogram(internet$"2009", breaks=8, xlab="Internet Usage", main="2009")

h11 =histogram(internet$"2010", breaks=8, xlab="Internet Usage", main="2010")

h12 =histogram(internet$"2011", breaks=8, xlab="Internet Usage", main="2011")

h13 =histogram(internet$"2012", breaks=8, xlab="Internet Usage", main="2012")

print(h1, split=c(1,1,4,4), more=T);

print(h2, split=c(2,1,4,4), more=T);

print(h3, split=c(3,1,4,4), more=T);

print(h4, split=c(4,1,4,4), more=T);

print(h5, split=c(1,2,4,4), more=T);

print(h6, split=c(2,2,4,4), more=T);

print(h7, split=c(3,2,4,4), more=T);

print(h8, split=c(4,2,4,4), more=T);

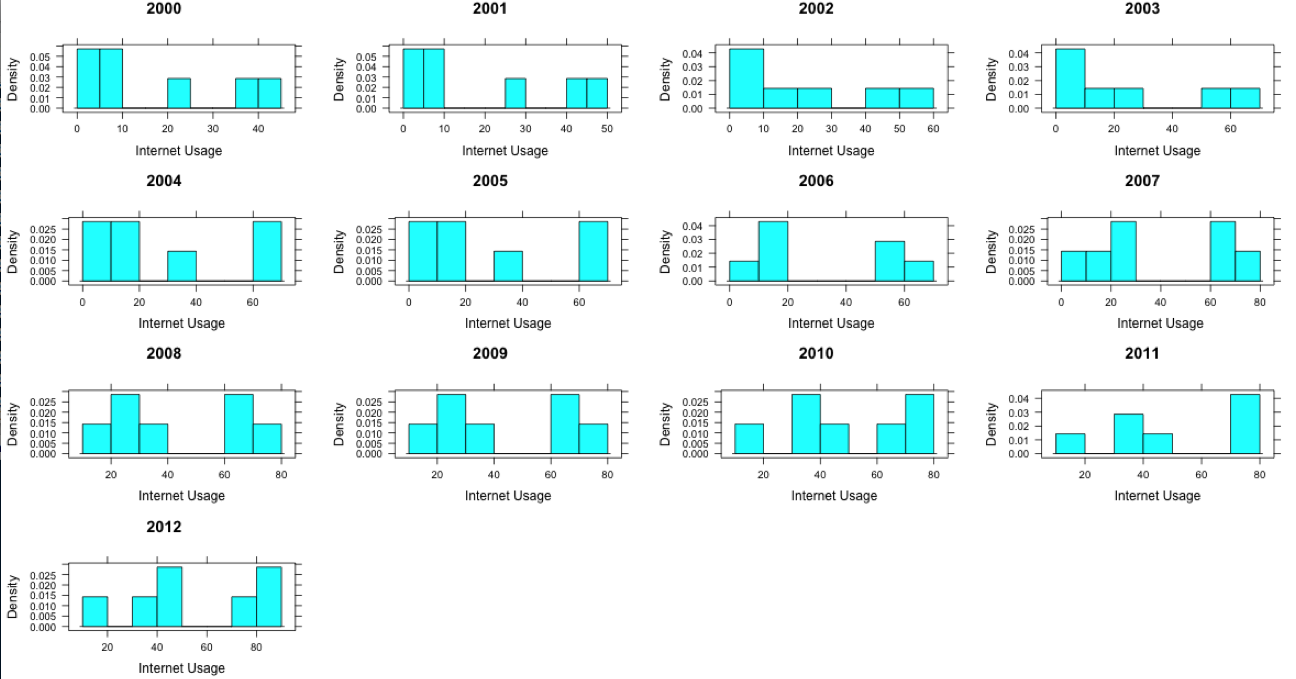
print(h9, split=c(1,3,4,4), more=T);

print(h10, split=c(2,3,4,4), more=T);

print(h11, split=c(3,3,4,4), more=T);

print(h12, split=c(4,3,4,4), more=T);

print(h13, split=c(1,4,4,4), more=F);



By reviewing the matrix we can see the changes in the distributions. We can see the countries represented in the lower frequency bins are moving towards increased internet usage from 2009 onward.

What if we wanted to compare countries? Unfortunately, the way that the data is formatted doesn’t allow us to compare countries. We can only compare years. To compare countries, ideally, you’d want to have your data in a format where you would have a column for country and a column for year. This will allow you to plot a simple x and y.

|  |  |  |
| --- | --- | --- |
| Country | year | usage |
|  |  |  |

The melt() function from the reshape2 package enables us to easily transform our data from wide to long format. To use the melt function you will need to install the reshape2 package. Go to packages > install packages > reshape2

k

**Next, let’s transform our internet data frame from wide to long format using the melt() function.**

The usage for melt() is:

melt(data, ..., na.rm = FALSE, value.name = "value")

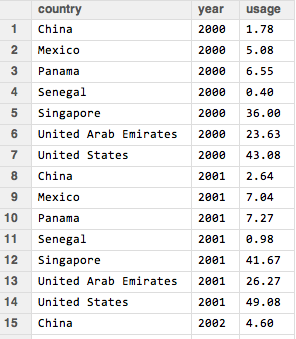
Arguments

|  |  |
| --- | --- |
| data | Data set to melt |
| na.rm | Should NA values be removed from the data set? This will convert explicit missings to implicit missings. |
| ... | further arguments passed to or from other methods. |
| value.name | name of variable used to store values |

> library(reshape2)

> iw\_melt = melt(internet,id.vars="country", variable.name="year", value.name="usage")

> View(iw\_melt)

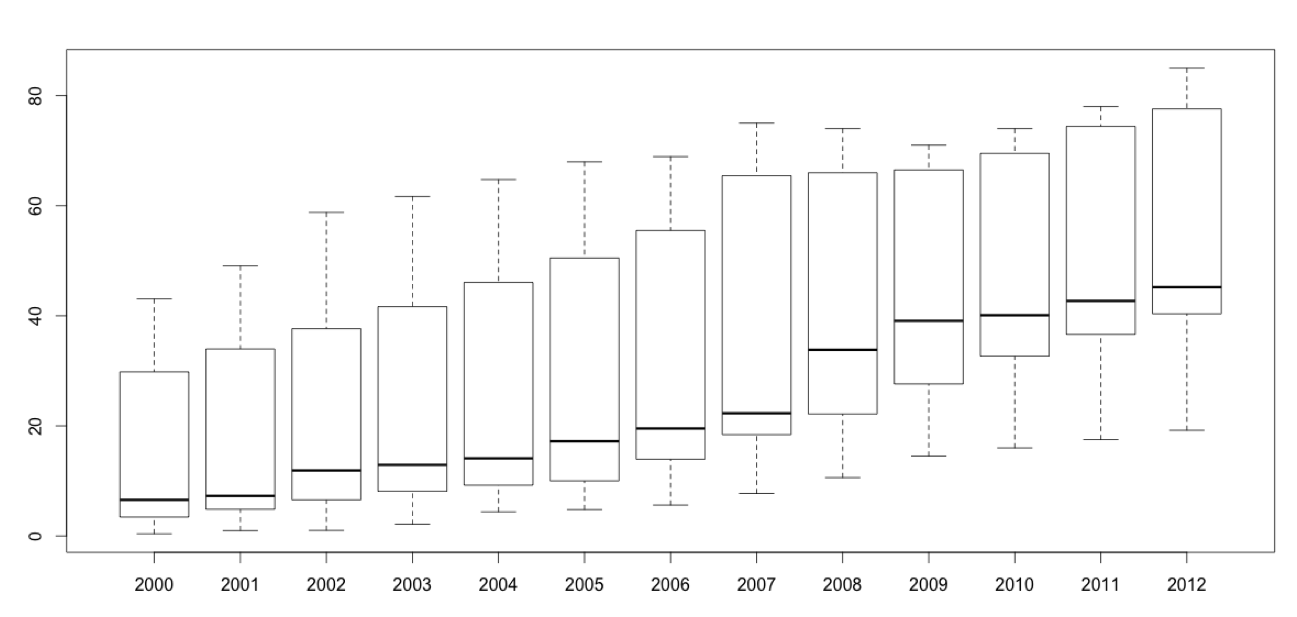
****

Now we can create a quick plot of our data.

l

> plot(iw\_melt$year,iw\_melt$usage)

What can you conclude from this plot? There is certainly a big range, but there is a general upward trend showing that usage is increasing over time.



Perhaps you wanted to look at the changes over time by country? This could provide some more details as to how countries compare to one another and how fast each country is adopting to the changes in technology.

A line chart is the best way to see this data. To create a multiple time series line chart, we need to use a more powerful graphics package called **ggplot2**.

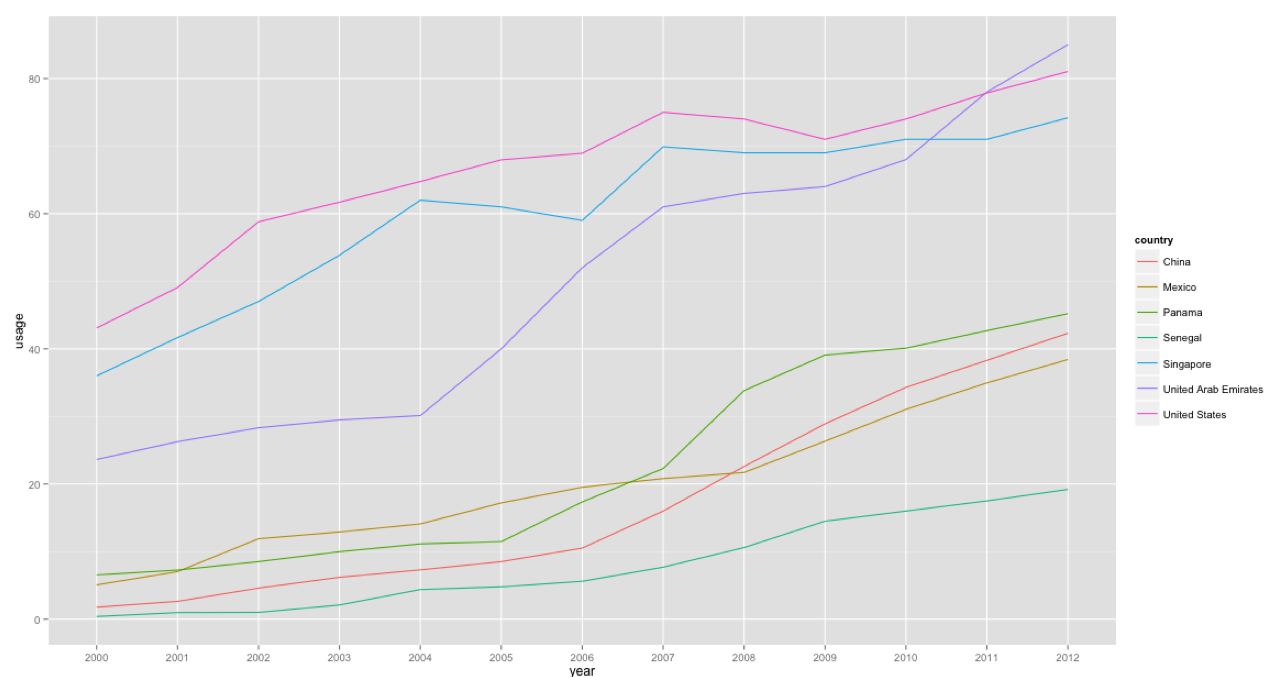
You will need to install this package, if you haven’t already. To install go to packages > install packages > ggplot2.

**Using the ggplot function we plot the year on the x-axis and the usage on the y-axis with a line plotted for each country.**

m

> library(ggplot2)

> ggplot(iw\_melt,aes(x=year,y=usage,colour=country,group=country)) + geom\_line()



By viewing the data as a time series line chart, we can clearly see those countries from our data set that are leading the world in internet usage per 100 people: United States, Singapore, and United Arab Emirates. This data was masked in the previous charts.

**5. Iteration using loops**

Iteration enables you to execute a block of code over and over again until some criterion or condition is met. In most programming languages, loops are used to perform iterative tasks. A loop is a block of code comprised of a sequence of commands that R executes over and over again until some termination criterion is met. In R we can construct a loop in several ways. We will study two functions that will enable us to create loops: The *while* loop and the *for* loop.

**The while loop**

The template for the while function is the following:

while (condition)

{

Statement1

Statement2

…

}

The *condition* in *while (condition)* must correspond to a logical value, either TRUE or FALSE. When R reads a while statement, it first evaluates to see if the *condition* is TRUE or FALSE. The condition can be something simple such as x==1. If the statement x==1 is TRUE, then the code within the curly brackets will execute and R will return back to the while (condition) statement. If the condition is still TRUE, R will then execute the code within the curly brackets until x==1 is FALSE. Now, if the condition x ==1 is FALSE, R will not execute the code within the curly brackets and the loop will be exited.

n

**Let’s suppose you want to print a number on the screen until you reach 9 printouts.**

counter = 0

while (counter <9) {

print(counter)

counter = counter + 1

}

[1] 0

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

[1] 6

[1] 7

[1] 8

**What’s happening here? (see the example on the whiteboard).**

|  |  |  |
| --- | --- | --- |
| **Condition** | **Output** | **Counter Update** |
| 0 | 0 | 1 |
| 1 | 1 | 2 |
| 2 | 2 | 3 |
| 3 | 3 | 4 |
| 4 | 4 | 5 |
| 5 | 5 | 6 |
| 6 | 6 | 7 |
| 7 | 7 | 8 |
| 8 | 8 | 9 |
| 9 | LOOP EXITS |  |

Since we started the counter at 0, we printed numbers 0 through 8 on the screen. Also, we used the < than instead of the less than or equal to <= operator. Consider changing the counter initialization value or the operator when you get unexpected results.

**The for Loop**

The for loop runs a fixed number of iterations based on the number of elements in a vector. The template for the *for loop* is pretty straightforward:

for (var in vector) {

Statement1  
Statement2

Etc.

}

For a given vector, each element within the vector corresponds to a possible value of the variable *var.* In the first iteration of the loop, the var is assigned the value of the first element of the vector. In the second iteration of the loop, the var is assigned the value of the second element of the vector, and it continues until all values of the vector have been exhausted. Then, the *for* loop terminates.

o (value in 1:10)

{

print(value)

}

**To illustrate how a *for* loop functions try the following code:**

for (value in 1:10)

{

print(value)

}

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

[1] 6

[1] 7

[1] 8

[1] 9

[1] 10

**Let’s return back to our *internet* data frame. Let’s grab the name of each country and make it upper case. Use the function toupper() and the for() function.**The usage for toupper() is: toupper(x) where x is the data you want to convert.

p

{

print(value)

}

for (i in internet$country)

{

caps = toupper(i)

print (caps)

}

[1] "CHINA"

[1] "MEXICO"

[1] "PANAMA"

[1] "SENEGAL"

[1] "SINGAPORE"

[1] "UNITED ARAB EMIRATES"

[1] "UNITED STATES"

**6. Conditional statements**

Unlike loops that execute a block of code over and over again, conditional statements execute only once based on the logic of set of one or more conditions. The *if* and *if else* statements are used to evaluate conditions for truth. That is, for logical or Boolean values. For example, the logic of the if statement works the following way:

q (value in 1:10)

{

print(value)

}

weather = “rain”

if (weather == “rain”) {

#bring umbrella

print(“Don’t forget your umbrella”)

}

In the example above we are evaluating the weather to see if it is indeed raining. If it is, that is if the resulting value of the evaluated statement weather == rain returns a value of TRUE, then and only then will the statement “Don’t forget your umbrella” be printed to the screen.  
  
The basic format of an if statement is:

if (condition) {

statement1  
 statement2

etc.  
}

If the *condition* is evaluated to TRUE, then R will execute the statements in between the curly brackets. If the condition is FALSE, the statements are not executed.

r (value in 1:10)

{

print(value)

}

**Let’s try to take the mean of the 2012 variable from the internet data frame and test to see if the first value of the vector is less than or equal to the mean. If it is, then print out the country name.**

meaninternet = mean(internet$"2012")

if (internet$"2012"[1] <= meaninternet) {

print (as.character(internet$country[1]))

}

**You can extend the *if* statement to include an alterative condition called *else.* The following structure is used:**

if (condition) {

statement1  
 statement2

etc.

} else {

statement3  
statement4  
etc.

}

If the condition is true, then statement1, statement2, etc. are executed. If the condition is false, then the second block of code enclosed within the curly brackets of the else statement are executed (i.e. statement3, statement4, etc.).

**Let’s try to take the mean of the internet$”2012” variable from the internet data frame and test to see if the first value of the vector is greater than or equal to the mean. If it is, print out the name of the country (i.e internet$”2012”[1]) as illustrated in the following sentence:**

s (value in 1:10)

{

print(value)

}

*Country name* has a lower internet usage than the mean.

**If it is not, then print out:***Country name* has a lower internet usage than the mean.

*Hint: to print out variables together with characters use the cat() function.*

meaninternet = mean(internet$"2012")

if (internet$"2012"[1] >= meaninternet) {

print (as.character(internet$country[1]))

} else {

cat (as.character(internet$country[1]), "has a lower internet usage than the mean.")

}

**Standard escape characters that are evaluated in print(), cat() and other text processing commands.**

|  |  |
| --- | --- |
| **Escape sequence** | **Meaning** |
| \n | Newline |
| \t | Horizontal tab |
| \v | Vertical tab |
| \b | Backspace |
| \r | Carriage return |
| \\ | Backslash |
| \’ | Single quote |
| **\”** | **Double quote** |

**Bringing together loops and conditions. Write a program that generates the following output from the internet$”2012” vector.**

t

{

print(value)

}

Singapore is greater than the mean of 55.05 with a value of 74.18

United Arab Emirates is greater than the mean of 55.05 with a value of 85

United States is greater than the mean of 55.05 with a value of 81.03

meaninternet = round(mean(internet$"2012"),2)

x = 0

for (i in internet$"2012"){

x = x + 1

#print(x)

if (internet$"2012"[x] > meaninternet){

country = as.character(internet$country[x])

cat(country, "is greater than the mean of", meaninternet,

"with a value of", internet$"2012"[x],"\n")

}

}

**7. Working with text and splitting strings**

Back to messy data. Working with text or strings can be difficult. Sometimes our data is stored as a long string of characters and we want to only the first few characters, such as the month, day or year in 10-12-1977. To parse a string we can simple use the strsplit() function.

The usage is:

strsplit(x, split, fixed = FALSE, …)

u (value in 1:10)

{

print(value)

}

**Parse 10-12-1977 based on the – delimiter.**

s = "10-12-1977"

s1 = strsplit(s, split='-', fixed=TRUE)

s1

[1] "10" "12" "1977"

The strsplit() function returns both pieces of the string parsed on the split parameter as a list. If you do not want that result, wrap the call in unlist, then index that array so that only the third of the three elements in the vector are returned. Finally, the fixed parameter should be set to TRUE to indicate that the split parameter is not a regular expression, but a literal matching character. See below.

s = "10-12-1977"

s1 = unlist(strsplit(s, split='-', fixed=TRUE))[3]

s1

[1] "1977"

**8. Writing functions**

In Lesson 1 the concept of a function was introduced, for example the Square Root function, sqrt(). There are many built in function in the baseline R distribution, and thousands more available in packages, but there may still be times when you want to go beyond what already exists, or you want to save time by customizing a function.

All functions in R have are called with the same basic syntax, function(), the name of the function followed by parenthesis. The parentheses contain any arguments that get passed to the function. The process for defining a function is shown below:

function\_name <- function(argument1, argument2,....){  
 The code to evaluate the function goes here.  
 return(object) #this object can be any variable type, array, or dataframe  
 }

The function is given a name, and the arguments to be passed are given names that will be used inside the function, or the local environment. Then the code to evaluate the function is placed between the curly braces. The return statement ends the function, and returns a variable to the global environment.   
  
“The ability to read, understand, modify and write simple pieces of code is an essential skill for modern data analysis. Lots of high-quality software already exists for specific purposes, which you can and should use, but statisticians need to grasp how such software works, tweak it to suit their needs, recombine existing pieces of code, and when needed create their own tools. Someone who just knows how to run canned routines is not a data analyst but a technician who tends a machine they do not understand.[[1]](#footnote-1)”

Let's look a simple example, a function that will return the [Net Present Value](http://en.wikipedia.org/wiki/Net_present_value) of a payment made one year from now. The formula for NPV is:

Where is the payment, is the risk free rate of return, and is the number time of the cash flow. Let's create a simple function that will calculate the NPV of a cash flow received one year from now. In that case the NPV formula can be reduced to this:

npv <-function (Pt,r,t){  
 pv=Pt/(1+r)^t  
 return (pv)  
}

Now see what happens when the function is evaluated.

Pt<-50  
r<-.05  
t<-1  
npv(Pt,r,t)

## [1] 47.61905

The function has returned the NPV of the payment one year from today.

**Let’s write a simple function together. If you remember back to one of our earlier exercises you were asked to compute various statistical measures for the *attitude* data set.**

v (value in 1:10)

{

print(value)

}

Rather than typing 9 functions for 7 variables = 63 lines of code, we can write a short function called descriptives.

#write function

descriptives=function(x){

c(mean = mean(x),

sd = sd(x),

median = median(x),

min = min(x),

max = max(x),

range = max(x)-min(x),

quantile = quantile(x),

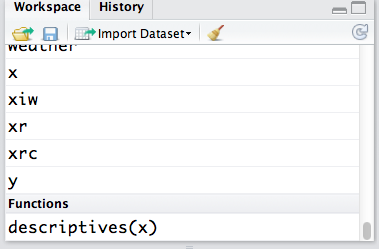
IQR = IQR(x),

var = var(x)

)

}

After we write the function, we can see that it is available for us to use at any point (just like a variable) in our workspace).



To execute or use our function we need to use the sapply() function. The function means that you are applying a function to a list or vector.

The usage for the sapply function is below:

sapply(X, FUN,…)

X is the data and FUN is the function. In our case, the data is attitude and the function is descriptives.

#execute function below  
sapply(attitude, descriptives)

**Return back to the march madness data set. Let’s say you want to add a new column to the data set called wins. Note: Wins is the first part of the string in the record column. Hint: use str\_split from the stringr library.**

w (value in 1:10)

{

print(value)

}

#import march madness data

#change columns to lowercase

names(march) = tolower(names(march))

library(stringr)

wins=unlist(str\_split(march$record, '-'))

wins

march\_wins = data.frame(march,wins)

View(march\_wins)

**Now, write a function for exercise w.**

library(stringr)

recordscore = function(x)

{

c(unlist(str\_split(x, '-'))[1])

}

sapply(march$record, recordscore)

march\_wins =sapply(march$record, recordscore)

march\_wins = as.numeric(march\_wins)

class(march\_wins)

max(march\_wins)

march\_new = data.frame(march$school, march\_wins)

View(march\_new)

names(march\_new) =c("school","wins")

View(march\_new)

**9. Writing reports**

**9a. An introduction and a note on LaTeX**

Up until now, we haven’t focused very much on sharing our work with others, or updating our visualizations in the light of new data. Say you have a dataset that comes in the same format every week, such as a sales report. Wouldn’t it be nice to simply produce a report with every week with a single click, updated with all new visualizations? **knitr** is an R package that makes it easy to just that! It also makes it easy to show others what you are doing in R by making it really easy to publish your code and results to a web page, Word Document, or PDF file.

**LaTeX,** An aside before we get started. We won’t be covering it here, but [LaTeX](http://en.wikipedia.org/wiki/LaTeX) is a very powerful, open source typesetting system. Its real strength is that it is capable of consistently formatting very complex text, such as equations for professional publication. If you want to use knitr to go directly to PDF, you will need to have a version of LaTeX installed, but you should know that you also have the option to go directly to LaTeX, and working with your output there. Learning LaTeX will likely require a significant amount time, but you find it to be a worthwhile investment.

**9b. Creating dynamic reports**

x(value in 1:10)

{

print(value)

}

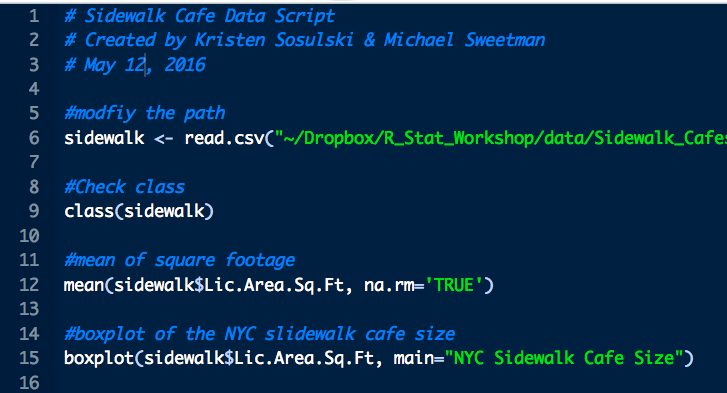
**Install knitr**

Begin by installing the knitr package along with any identified dependencies in RStudio. For a video demo of knitr go to: <https://www.screenr.com/qcv8>

**Open or create an R script**

Let’s start off with a fairly simple R script, for example, one that will import the Sidewalk Café data and create a simple box plot, as shown in the figure below.

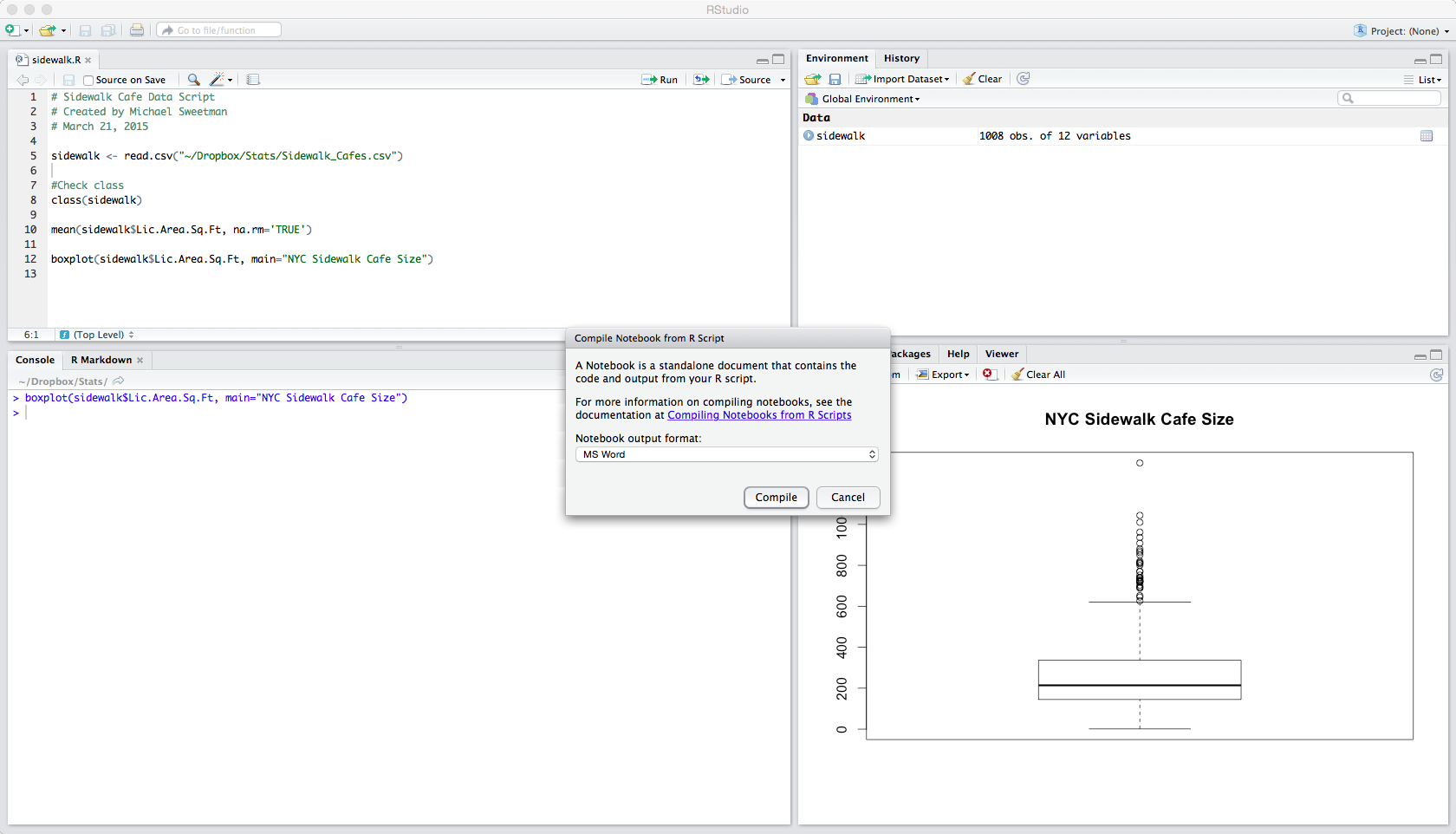
Data: Sidewalk\_Cafes.csv

Code: sidewalk\_example\_01.R (open this file and change the path in read.csv())  
****

One way to share your work would be simply share the R script file. This method is easy, but has drawbacks; particularly that it isn’t formatted for easy reading or style. It also isn’t easy to share your results at the same time. This is where knitr can be very useful. It can easily format your script file for embedding in an HTML webpage, Word Document, or PDF file.

Let’s try it: Just click on the “Compile Notebook” button   
(You can also chose the “Compile Notebook” command from the File menu).

You will be presented with a menu to chose your output format, either HTML, Word, or PDF, as shown below. If you chose to go directly PDF, you will need to have a specific version of LaTeX installed depending on your operating system, the error message in the console will tell you which version you would need. For now, let’s choose MS Word.

****

Shown below is the compilation of that script. This format makes it easy to add explanatory notes or commentary. The plot is also created and included. Notice that the output of each command is denoted with ##, and interspersed with the R code. That is helpful since any line that begins with # is treated as a comment. So anyone who wants to run the code can copy and paste then entire block of text, including the output into the console, and it the code will be executed, but the old output will be ignored.

slidewalk\_example\_01.R

ksosulsk

Thu May 12 11:13:47 2016

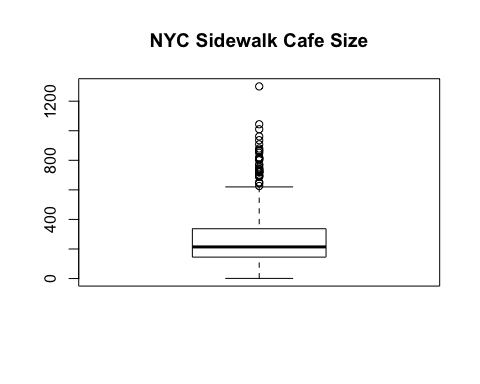
# Sidewalk Cafe Data Script  
# Created by Kristen Sosulski & Michael Sweetman  
# May 12, 2016  
  
#modfiy the path  
sidewalk <- read.csv("~/Dropbox/R\_Stat\_Workshop/data/Sidewalk\_Cafes.csv")  
  
#Check class  
class(sidewalk)

## [1] "data.frame"

#mean of square footage  
mean(sidewalk$Lic.Area.Sq.Ft, na.rm='TRUE')

## [1] 258.6475

#boxplot of the NYC slidewalk cafe size  
boxplot(sidewalk$Lic.Area.Sq.Ft, main="NYC Sidewalk Cafe Size")



**Creating an R Markdown File**

The above example demonstrated how to publish your R scripts and output, but it still has some drawbacks. Suppose you want to create a report that updates automatically? The above method will still be easier than manually creating the report each time, but formatting and adding comments will still have to be done manually every time the report is run. Creating an R Markdown file will allow us to automate the process of data manipulation, report creation, and formatting.

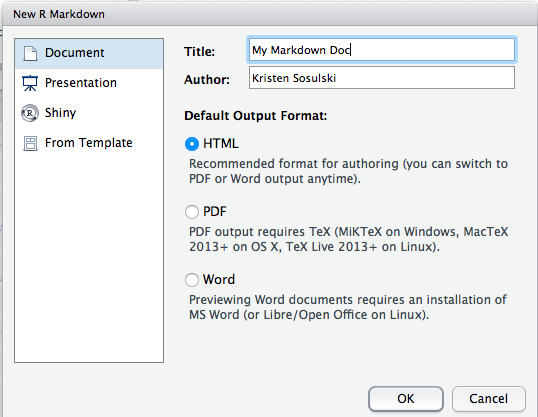
y (value in 1:10)

{

print(value)

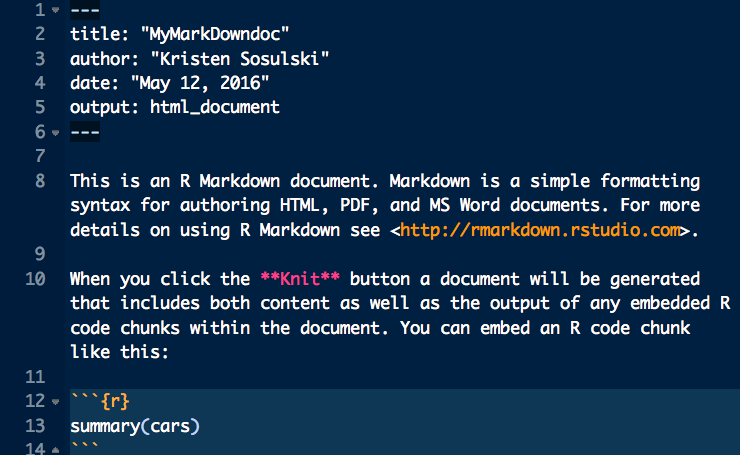
}

To get started go to the File menu, click New File > R Markdown.  
(Note: The first time you do this you will be prompted to install several packages. Allow the install.) The following dialog box will appear:

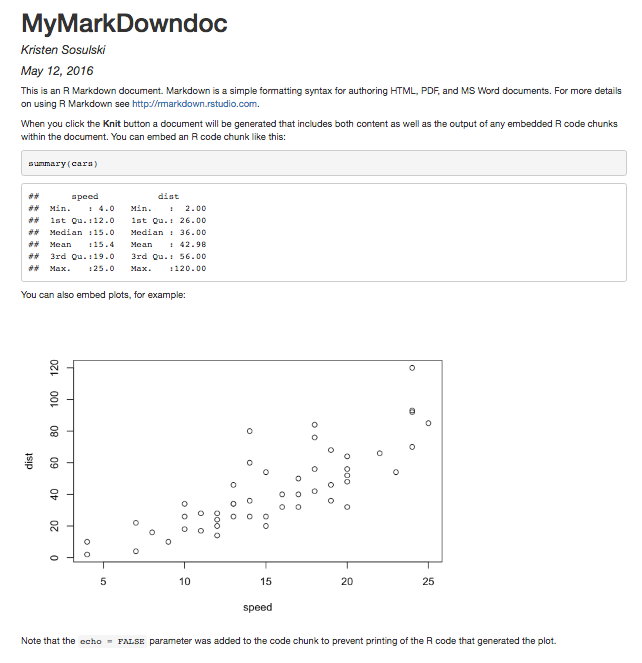


**Title and Author**: Enter a name for the file, and your name. Select HTML as the default output format. Click OK.

R Studio will create a Markdown file for you to begin editing. This document will allow you to create a file using simple plain text, which can then be easily converted into HTML, Word, or PDF. You can easily embed your R code and output. The file that RStudio creates will also be pre-populated with example Markdown syntax file using one of the sample data sets embedded in R. Following is the example Markdown file:



To see the HTML file, select 



**Some Basic R Markdown Syntax**

Let’s first introduce the syntax used by R Markdown. As stated before, the goal of [Markdown](http://en.wikipedia.org/wiki/Markdown) is to allow creation of webpages automatically from a plaintext base. The R Markdown implementation is based on the Markdown standard, but unfortunately the Markdown language is somewhat fragmented, so the syntax used in R Markdown will likely differ from any other implementation of Markdown that you come across.

The purpose of the specialized syntax in the R Markdown file is to allow the translator to identify the intended use of the text, be it a heading, bold, R code, etc. R code is embedded in the markdown file in “chunks.” The default is that these chunks will be formatted and included in the report, but that option can be turned off with the “echo=off” option. Below is a list of several useful commands.

**Document Heading**

---

heading text

---

**Bold**

\*\*text\*\*

**Italic**

\*text\*

**List**

\* Item 1

\* Item 2

+ Item 2a

+ Item 2b

**R Code**

```{r}

place the R Code here

```

**R Code With Options**

```{r, echo=FALSE}

echo=FALSE will output the result of the code without outputting the code itself. Another common option is eval=FALSE, which puts the code in the report without the output of the code.

```

**Inline R Code**

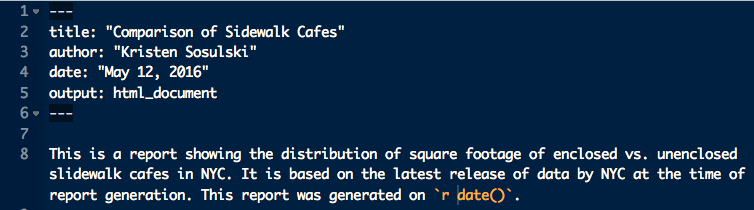
You can place R code in the middle of the line like this. `r place code here`

That’s just a bit to get you started, there are many other options, and they are detailed at the [Rstudio Markdown](http://rmarkdown.rstudio.com) page. There is also an excellent “[cheat sheet](http://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf)”, which shows many different syntax commands.

**An R Markdown Example**

Let’s say that NYC wants to track the square footage trend in enclosed vs. unenclosed sidewalk cafes. They want a webpage that will show everyone the current distribution of square footage of enclosed vs. unenclosed sidewalk cafés, as well as some of the code that was used to perform the calculations. We’ll create the entire report using an R Markdown file.

Start off with the Markdown file we opened earlier. Delete everything except the title block, which you can modify to your liking. Below the three dashes that end the title block we can begin to add our text. Let’s put in a quick line of introduction, for that we can use plaintext. It may be a good idea to show when the report was generated, so we can use the inline R code call discussed above to add the date. The function r date(), will return the current date as a character string, so we can add that to our introductory text. See below:

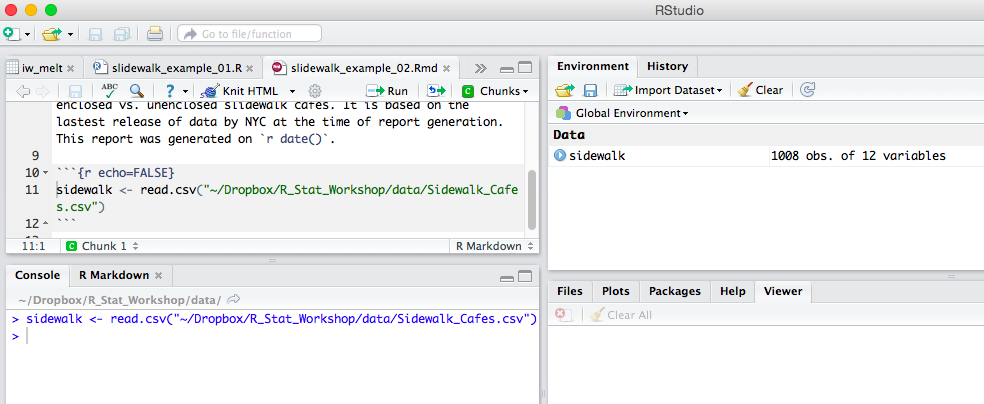


To generate your HTML, Word, or PDF file, you just click on the Knit button that you see in the above image. Below is the generated report. Notice the date of the execution has been placed in the report. The inline code execution option can be particularly useful for placing summary statistics into the narrative description.

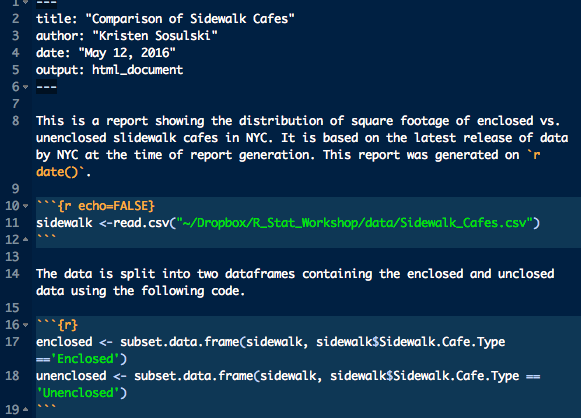


Now the next step is to import the sidewalk café data. Since this is a perhaps an uninteresting step for this report we will not “echo” this code into the report. Start by inserting an R code block or chunk into the Markdown file, using the syntax given in the previous section.

You could also use the Chunks button in RStudio  to save some typing.  
  
Inside the code chunk, type the command to import the data. You can test your R code execution from the drop down menu on the Chunks button. You can see the results of executing the code chunk in the picture below. The import command was executed, and the sidewalk data frame now exists.



Now we have to process the data, specifically we want to split the sidewalk café data set into two sets, representing the enclosed and unenclosed data. This code is a little more complex, so perhaps we want to add a line or two of explanation before this step. You can see the narrative for the report, and the second code chunk in the screenshot of the Markdown file below.

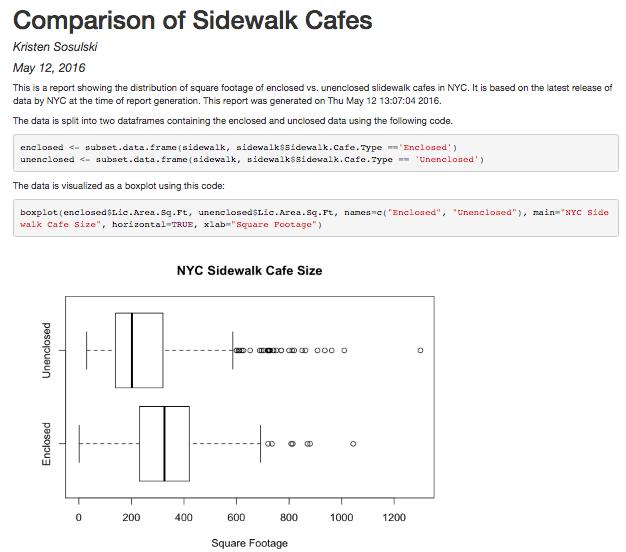


And below is the additional portion of the report. Remember, we echoed back the code this time. Since subsetting the data frame doesn’t produce any visual output, there is no evaluation of the code to add to the report, but the code is executed nonetheless.

The data was split into two data frames containing the enclosed and unenclosed data using the following code.

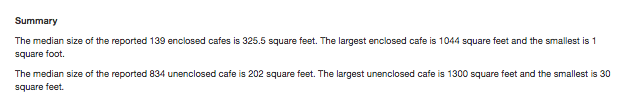
enclosed <- subset.data.frame(sidewalk, sidewalk$Sidewalk.Cafe.Type == 'Enclosed')  
  
unenclosed <- subset.data.frame(sidewalk, sidewalk$Sidewalk.Cafe.Type == 'Unenclosed')

The final step for this report is to produce the boxplot. Again, we will insert some descriptive narrative, and show the code used to produce the plot. However, since this time there is output from the execution of the R code, the visualization will also be added to the report. The final report, encompassing the entire Markdown file, is shown on the next page.

****

**Take it one step further.**

Try to replicate the additional report “Summary” output produced below.

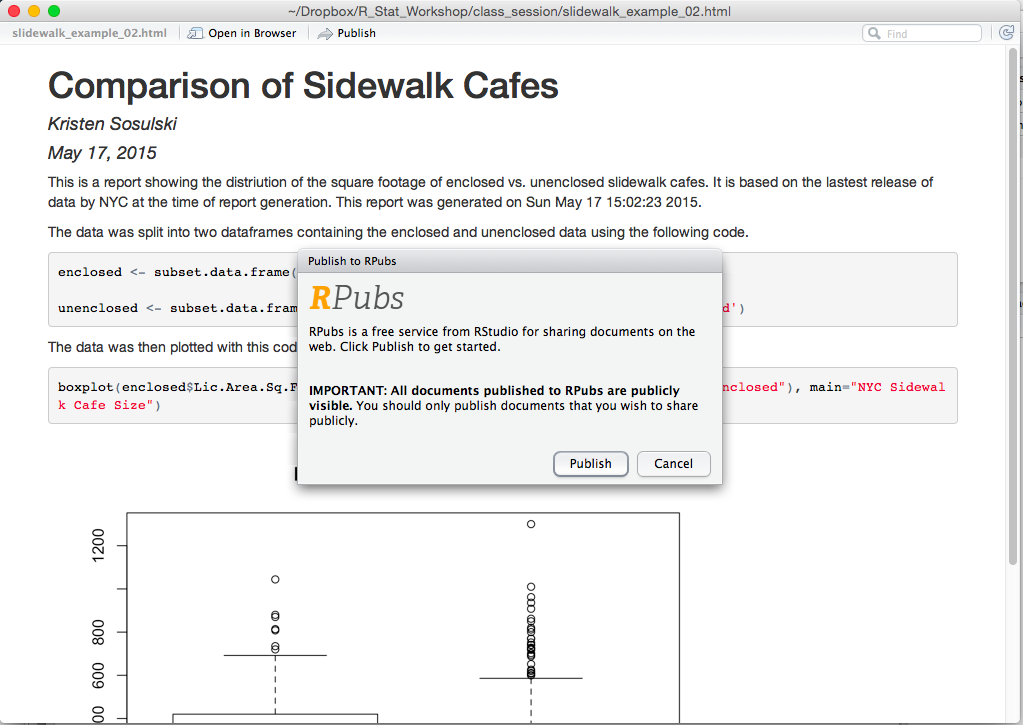


**Publishing to RPubs**

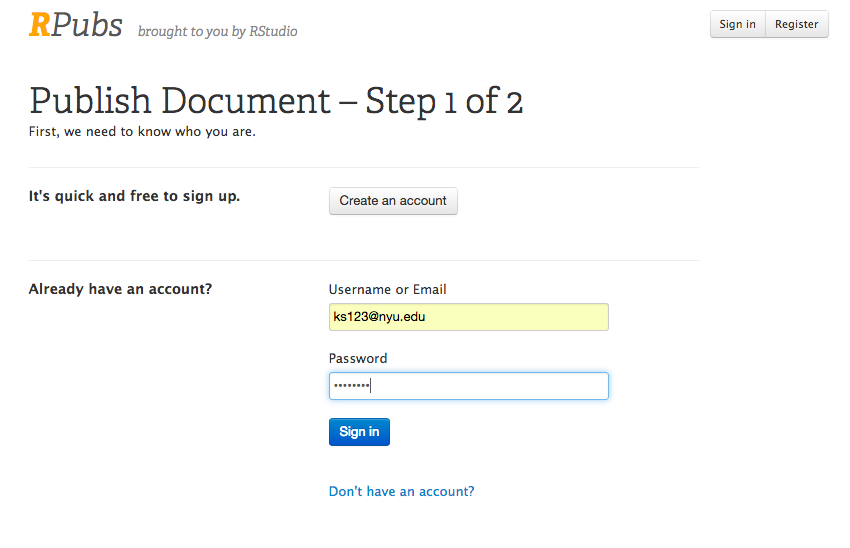
You can publish your .R and .Rmd documents to RPubs.

After you run Knit HTML, you will see an option to publish.

Select, Publish and the RPubs dialog box will appear. Select Publish.



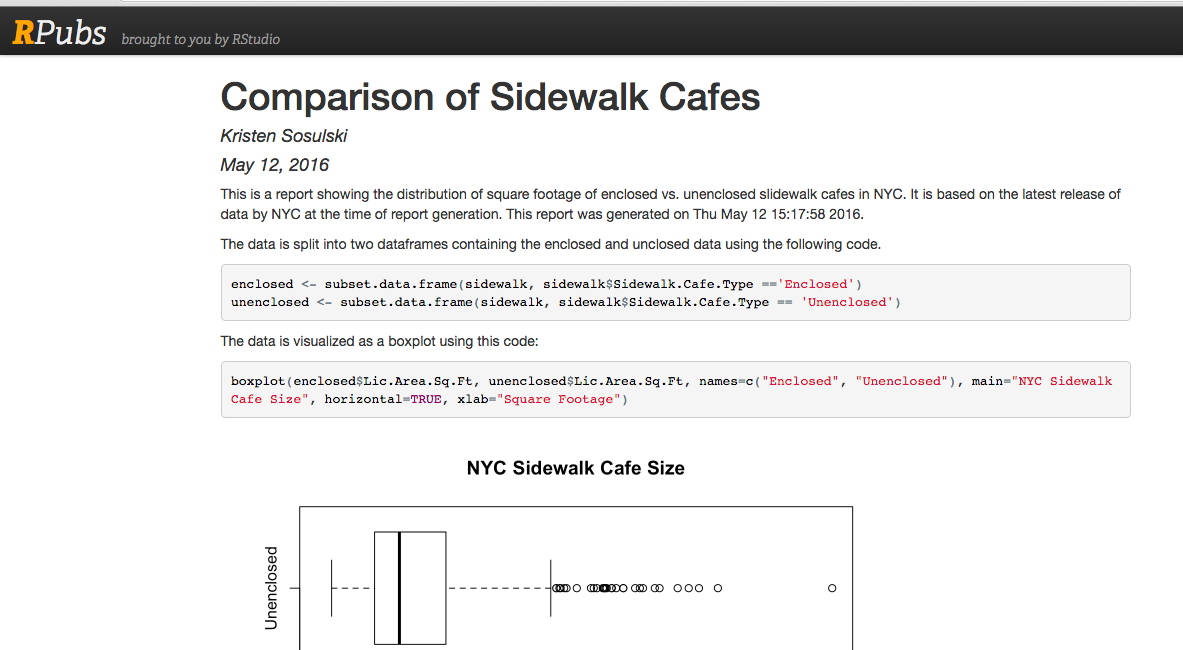
Then you will be presented with an option to login or create an account on RPubs.



After you create an account, provide your document with a title.



Then you will be provided with a URL you can share and update. The published document is below and available at: <http://rpubs.com/sosulski/180484>



This section introduced the KnitR and R Markdown modules in R, and demonstrated some of the basic functionality. There is much more functionality than was demonstrated here, for example Markdown can also easily add tables to reports and there are many additional formatting options. There are other modules, Shiny, for instance, that can be used to add interactivity.

**10. Shiny**

[Shiny](http://shiny.rstudio.com) is an application for R that allows for publishing of your work to the web in an interactive format without detailed knowledge of the underlying web code (HTML, JavaScript, or CSS).

Here’s an example of a capstone project using shiny:

https://team-endurance.shinyapps.io/TE\_PM/

This tutorial is based off the more detailed tutorial at [R Studio](http://shiny.rstudio.com/tutorial/), and will demonstrate a few of the capabilities of Shiny, using the NYC Sidewalk Café data from earlier in the class.

#### 10.1. Basic Structure of a Shiny App

The first thing you need to do is to make sure you have the Shiny package installed and enabled. Next, we will look at the basic operation of a Shiny app.

The basic structure of a Shiny app consists of a folder in the working directory of R, for example: app\_1. That folder then contains two R script files, server.R and ui.R. Server.R contains the R commands that govern the server in performing calculations, analyzing data, and creating visualizations. ui.R contains the instructions for layout of the user interface and controls the interaction with the user. The app is then launched with the command runApp(“app name here”). When you run an app, you can no longer interact with the command line interface of R, as the runApp command is continuously running to be able to respond to commands from the user interface.

#### 10.2 Quick Demo Example (Try it)

z(value in 1:10)

{

print(value)

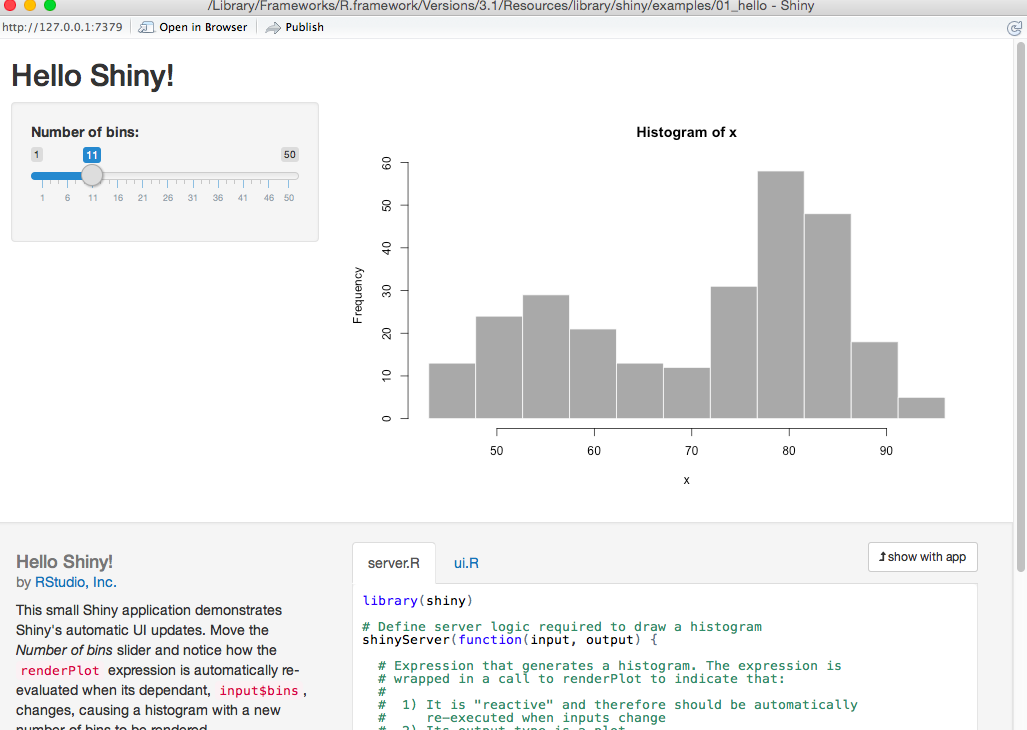
}

There is a set of built in examples included with Shiny. Let's look at the first one, a basic histogram. To run it, first make sure the Shiny package is installed and enabled, then run:

> require(shiny)

> runExample**("01\_hello")**

This will bring up a sample histogram, and a slider to control the bin size. You can also see the code being used to generate the histogram and slider below the histogram. Your view should be similar to the image below.

****

If you look at the console window of R Studio, you will see a small STOP sign icon. If you click that button, it will stop the execution of the sever code, and allow you to interact with R Studio again.

#### 10.3 Modifying the Example to use the Sidewalk Cafe Data

Now try making your own histogram using this example as a base, but substituting in the sidewalk cafe dataset. First create a new directory in your working directory called histogram. Next, copy and paste the code below into new R script files.   
  
**First, the server.R file:**

library(shiny)  
  
# Define server logic required to draw a histogram  
shinyServer(function(input, output) {  
   
 # Expression that generates a histogram. The expression is  
 # wrapped in a call to renderPlot to indicate that:  
 #  
 # 1) It is "reactive" and therefore should re-execute automatically  
 # when inputs change  
 # 2) Its output type is a plot  
   
 output$distPlot <- renderPlot({  
 x <- faithful[, 2] # Old Faithful Geyser data  
 bins <- seq(min(x), max(x), length.out = input$bins + 1)  
   
 # draw the histogram with the specified number of bins  
 hist(x, breaks = bins, col = 'darkgray', border = 'white')  
 })  
})

**And now ui.R**

library(shiny)  
  
# Define UI for application that draws a histogram  
shinyUI(fluidPage(  
   
 # Application title  
 titlePanel("Hello Shiny!"),  
   
 # Sidebar with a slider input for the number of bins  
 sidebarLayout(  
 sidebarPanel(  
 sliderInput("bins",  
 "Number of bins:",  
 min = 1,  
 max = 50,  
 value = 30)  
 ),  
   
 # Show a plot of the generated distribution  
 mainPanel(  
 plotOutput("distPlot")  
 )  
 )  
))

You can read the comments in the code to get a basic idea of what each part does. The ui.R file looks pretty good, except we want to change the title. We'll edit "Hello Shiny" to say "NYC Sidewalk Cafe Data".   
  
Now we'll pull in the sidewalk cafe data, creating a histogram of the square footage of the sidewalk cafes. The line of the code that pulls in the data is in the server.R file, and is below:

x <- faithful[, 2] # Old Faithful Geyser data

This needs to be changed to pull in the sidewalk cafe square footage data instead. Also, if you notice, the next line calculates the bin length. That calculation will fail if there are any "NA" values in the data, so the NA values need to be dropped. **Copy the Sidewalk\_Cafes.csv file to your histogram directory**. The new code will look like this:

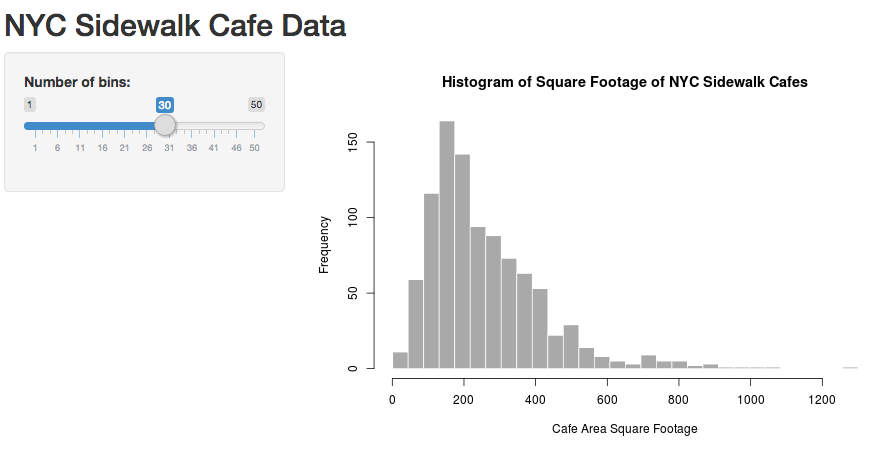
sidewalk <- read.csv("Sidewalk\_Cafes.csv")

x <- na.omit(sidewalk$Lic.Area.Sq.Ft)

Now if you try running the histogram code you should see a histogram formed with sidewalk cafe data. You can run the code by selecting the RunApp button in RStudio.



You should also add a new title and axis labels to the chart, finishing with something like the output below:



### 10.4 Elements of a Shiny App

As previously stated, the code that runs a Shiny app resides in two files, server.R for the server commands, and ui.R for the User Interface. We'll now explore the elements that are used for building the apps are deployed and interact with one another.

#### The User Interface

The user interface begins with the fluidPage function. This function creates a blank webpage that is automatically sized to the browser window. Next, panels are embedded in the webpage using the fluidPage function. It is common to use a title panel, and the sidebarLayout function. The sidebarLayout function requires a sidebarPanel and mainPanel. Each of the above functions takes arguments, which can be in the form of non-interactive text, or much more advanced functions.   
  
Here is a simple example using only non-interactive text:

shinyUI(fluidPage(  
 titlePanel("Hello Shiny!"),  
 sidebarLayout(  
 sidebarPanel("Hello from sidebarPanel"),  
 mainPanel("Hello from mainPanel")  
)  
))

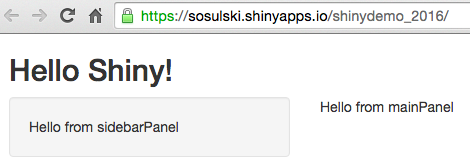
Create a directory called "shinydemo\_2016" in your working directory and save the above code to ui.R

#### The Basic Server File

A basic server file consists of the shinyServer function, which serves to receive input from and deliver output to the User Interface. The code is shown below.

shinyServer(function(input,output) {  
}  
)

Copy and paste that code to a file called server.R in the shinydemo\_2016 directory. You now have a Shiny app that will display static text in a title panel, a side panel, and a main panel. The server will also be listening for input from the UI. If you run shinydemo\_2016, you should see the webpage shown below.



#### Reviewing the Histogram Example

The code in the histogram example should make more sense now, and it is also a good example of some simple interaction between the UI and the server. Let's review it. (All the comments have been removed for a more compact layout.)

shinyUI(fluidPage(  
   
 titlePanel("NYC Sidewalk Cafe Data"),  
  
 sidebarLayout(  
 sidebarPanel(

Everything above is the same as our simple "shinydemo\_2016 " example. However, the chunk of code defines the slider widget.

sliderInput("bins",  
 "Number of bins:",  
 min = 1,  
 max = 50,  
 value = 30)  
 ),

There are a few new things going on in the above code. First, the sliderInput function is a UI widget. There are many predefined widgets you can use to build your page. There is a specific tutorial on building widgets at [R Studio,](http://shiny.rstudio.com/tutorial/lesson3/) and a gallery of available widgets at: http://shiny.rstudio.com/gallery/widget-gallery.html

The next thing to notice is in the first line of the code, the sliderInput function is making the value of the slider available to the server through the variable named "bins." This is how the UI interacts with the server. The remaining lines of code are the values and text that are displayed on the slider.

mainPanel(  
 plotOutput("distPlot")  
 )  
 )  
))

The next lines of code illustrate how output from the server is displayed in the UI. The mainPanel is displaying the plot, which is defined in the server code. The final lines of code in the UI file are quite straightforward, and are just closing braces for the various functions in the UI file.

Now let's take another look at the server.R file.

shinyServer(function(input, output) {  
  
 output$distPlot <- renderPlot({  
 x <- na.omit(sidewalk$Lic.Area.Sq.Ft)  
 bins <- seq(min(x), max(x), length.out = input$bins + 1)

The server file opens with the shinyServer function that is the basic function to set the server listening for input and output. The next line defines the output as the distPlot function, which is referenced in the UI file, in the mainPanel. distPlot is defined by the renderPlot function, and as noted in the comments(which were removed), the renderPlot function will cause the plot to be redrawn automatically when one of the inputs changes. These are the functions that add interactivity to your R code, and they are explained in much more detail at: <http://shiny.rstudio.com/tutorial/lesson4/>

The next line contains our data, but the following line is a little more complicated. It defines the break points for the histogram bins, using a sequence from the min x to the max x, with the number of break points dictated by the input from the slider. Now the entire sequence should be clear. Moving the slider will cause the UI to update the bins variable to the server. The server will take that input, recalculate the bin size, update the plot, and return the plot to the UI for the display to be redrawn.

hist(x, breaks = bins, col = 'darkgray', border = 'white', main='Histogram of Sidewalk Cafe Size', ylab="Count", xlab='Cafe Area Square Footage')  
 })  
})

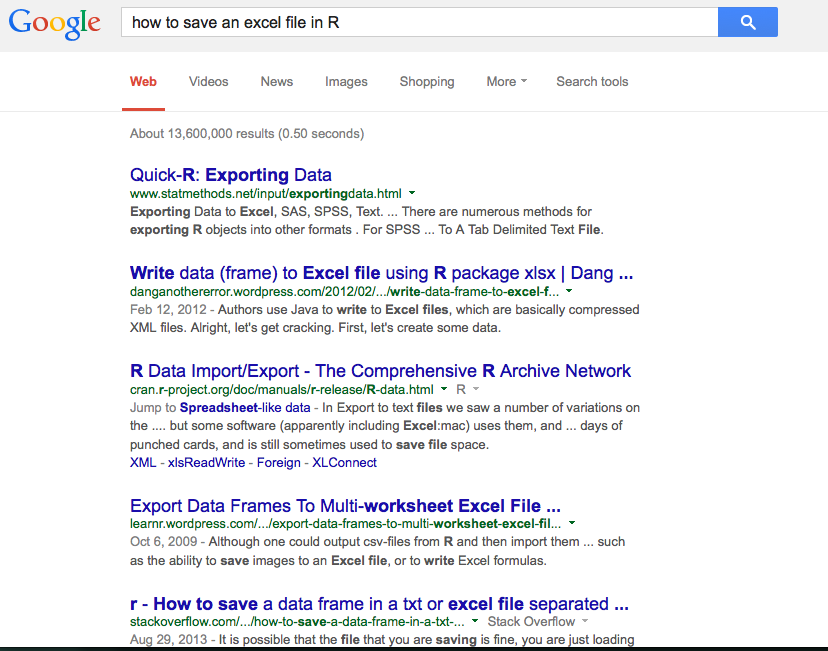
The remaining code is simply standard R for drawing a histogram, and the closing braces indicating the end of the various functions.

**11. Summary**

**As with any programming language (or language in general) there’s always more to learn.**

R is powerful. It can read and write JSON objects through rjson package. It can open webpages via the RCurl package. You can print your modified data frames to different format files. You can execute regular expressions such as grep and sed for textual pattern matching of strings. You will never be done learning R. It is a language that is evolving with technological change. Keep up to date with CRAN (the R source for everything). Also, use Google to search for your R questions.

StackoverFlow, statmethods, cran.r are your major sources of information of any regarding R. For example, suppose you wanted to export your data as an excel (.xls file). Simple search Google:



**Overview of command used in this lesson:**

* Vectors can be sorted using the sort() function
* The order() combined with the *with* function is best used when sorting multiple columns of a data frame.
* melt() from the reshape2 package and used to convert wide to long data sets
* ggplot() from the ggplot2 package is used to create sophisticated plots such as colorful line charts.
* while and for are two different types of loops for iterative processes
* if, else, ifelse are the logical control structures in R.
* print() prints a string or variable to the screen
* cat() concatenates and prints strings with other data to the screen
* The strsplit() function returns both pieces of the string parsed on the split parameter as a list
* Use unlist() with strsplit and an index to return a specific delimited element in the string
* Escape sequences enable printing of reserved characters in R such as single or double quotation marks.
* Functions: One of the great strengths of R is the user's ability to add functions. In fact, many of the functions in R are actually functions of functions. The structure of a function is given below.
* sapply() applies a function over a vector or a list
* Reports: use the knitr package to create reports in R. Reports can be published as a Microsoft Word document, PDF, or HTML file. HTML files can be shared and published to RPubs.
* Some of the main things to keep in mind are with regard to reports:
* An R script can be compiled for export in HTML, Word, or PDF simply with the KnitR function just by clicking the Notebook button in RStudio.
* R Markdown allows for conversion of near plaintext to well-formatted webpages and other formats.
* R Markdown also allows for the interspersing code blocks with narrative or the output from the code execution.
* The R code can be hidden from the report and just the output shown, or visa versa.
* The R code can also be executed and the result displayed inline, a useful feature for inserting results into a narrative description.
* Some of the common formatting was shown, bold, italics, lists, and headings. There are many more options available, many of which were not demonstrated.
* R markdown files enables you to chunk you code and create dynamic reports with code chunks and the output. Markdown files can also be published on RPubs.
* The shiny package enables you to create interactive web applications.

**10. Lesson Resources**

**Support**

Google

Stack Overflow

**Tutorials**

<http://tryr.codeschool.com/>

<http://www.r-tutor.com/>

**Examples**

Google

Stack Overflow  
Github

**General R syntax and programming guide**

<http://cran.r-project.org/doc/manuals/r-release/R-lang.html>

<http://statmethods.net>

[Software for Data Analysis: Programming with R (Statistics and Computing)](http://chggtrx.com/click.track?CID=267582&AFID=301076&ADID=1088031&SID=compdata&isbn_ean=9781441926128) http://www.assoc-amazon.com/e/ir?t=coursera-20&l=as2&o=1&a=1441926127by John M. Chambers (Springer)

**Writing functions and loops**

* <http://www.statmethods.net/management/userfunctions.html>
* <http://www.stat.cmu.edu/~cshalizi/402/programming/writing-functions.pdf>
* [Writing a for-loop in R](http://paleocave.sciencesortof.com/2013/03/writing-a-for-loop-in-r/)  
  http://paleocave.sciencesortof.com/2013/03/writing-a-for-loop-in-r/
* [How to write and debug an R function](http://www.r-bloggers.com/how-to-write-and-debug-an-r-function/)  
  http://www.r-bloggers.com/how-to-write-and-debug-an-r-function/
* [User-written Functions](http://www.statmethods.net/management/userfunctions.html)  
  http://www.statmethods.net/management/userfunctions.html
* [R Library: Introduction to functions](http://www.ats.ucla.edu/stat/r/library/intro_function.htm)
* http://www.ats.ucla.edu/stat/r/library/intro\_function.htm

**R Markdown**

* Overview: http://rmarkdown.rstudio.com/
* Cheat sheet:<http://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf>

**Shiny**

* Tutorial:<http://shiny.rstudio.com/tutorial/>
* Publishing Shiny Apps: <https://www.shinyapps.io/admin/#/dashboard>

1. http://www.stat.cmu.edu/~cshalizi/402/programming/writing-functions.pdf [↑](#footnote-ref-1)