ECON 390A

Lecture 1

Part#01

What is R?

R is a free software environment for statistical computing and graphics

Installing R

Carry out the following step

Go to http://cran.r-project.org/ and install the version of R for your operating system.

R Studio:

RStudio is a free and open source integrated development environment (IDE) for R.

To download RStudio please visit: http://rstudio.org/

Please note that you must have R already installed before installing R Studio.

R compiles and runs on a wide variety of UNIX platforms, Windows and Mac OS

R is open-source and free

R is fundamentally a command-driven system

R is an object-oriented programming language

Everything in R is an object (data, functions, etc.)

R is highly extendable

You can write your own custom functions

There are over 13000 free add-on packages (`nrow(available.packages())`)
To make sure this worked, open the program RStudio and go to File > New > R Script. This will open a blank text document.

```
x = 8

x == 5

x==8
```

You've just written your first R script! To save it, go to File > Save As, and choose a name. NOTE: Always save your scripts as .R files so they'll open in RStudio by default.

The Absolute Basics

Here are some of the most fundamental things you can do with R.

Arithmetic

```
#add numbers
5 + 1
## [1] 6
#subtract them
6 - 4
## [1] 2
#divide
13/2
## [1] 6.5
#multiply
4*pi
## [1] 12.56637
#exponentiate
2^10
## [1] 1024
```

Logical Comparison

```
X==y; x is equal to y                 x!=y; x is not equal to y
X<y; x is less than y                 !b; NOT b(i.e. b is FALSE)
x>y; x is greater than y              a|b; Either a or b is TRUE(or both)
x<=y; x is less than or equal to y    a&b; Both a and b are TRUE
x>=y; x is greater than or equal to y

3 < 4
## [1] TRUE
3 > 4
## [1] FALSE
#contrast with 3 = 4; see section about variables below
3 == 4
## [1] FALSE
#!= means "not equal to"
3 != 4
## [1] TRUE
4 >= 5
## [1] FALSE
4 <= 5
## [1] TRUE
2 + 2 == 5
## [1] FALSE
10 - 6 == 4
## [1] TRUE
```

Strings (text)

Numbers are bread and butter for programming language, but text is what will facilitate understanding for us mere mortals.

```
'my AP goin psycho'
## [1] " my AP goin psycho"
#R delimits strings with EITHER double or single quotes.
#  There is only a very minimal difference
"Can't really trust nobody with all this jewelry on you"
## [1] "Can't really trust nobody with all this jewelry on you"
```

Variables

Just like in algebra, variables are a great form of shorthand. Instead of writing 3.1415926… all the time, we can just write pi. Assignment to a variable happens from right to left – the value on the right side gets assigned to the *name* on the left side. You can use nearly anything as a variable name in R. The only rules are:

1. . and _ are OK, but no other symbols.
2. Your variable name must not start with a number or _ (2squared and _one are illegal).

[A note for those of you who have programming experience: while R supports object-oriented programming, periods . do *not* have a special meaning in the language. For historical reasons, R programmers often use periods in place of underscores in variable names, but either works. Just be consistent to keep your code readable.]

```
x = 42
x / 2
## [1] 21
#if we assign something else to x,
#  the old value is deleted
x = "Is it cool that I said all that!"
x
## [1] "Is it cool that I said all that!"
x = 5
x == 5
## [1] TRUE
fonsi = 2
yankee = 2
fonsi.yankee = fonsi + yankee
fonsi.yankee
## [1] 4
bieber=2
despacito = bieber * fonsi.yankee
despacito
## [1] 8
```

Note: In programmer speak, = here is an "assignment operator" – it's the thing used to assign values to a variable name. R also has a second assignment operator that you're bound to see sooner or later, <-. So x <- 42 and x = 42 are *identical*, and both accomplish the task of assigning the value of 42 to the name x. We'll try to stick with using = since it's easier to type and in some ways more intuitive.

Vectors & Types

In R, a vector is just a (ordered) set of related things. You should basically think of it like a column in Excel.

```
x = c(4, 7, 9)
x
## [1] 4 7 9
y = c('a', 'b', 'c')
y
## [1] "a" "b" "c"
```

4, 7, and 9 are "related" because they're all numbers; a, b and c are all letters. Having variables is becoming more convenient – instead of having to write c(4, 7, 9) all the time, we can just write x. What happens when we try and combine things that aren't so obviously related?

```
x = c(1, TRUE, "Baldwin")

x

## [1] "1"     "selina"  "Baldwin"
```

Note the quotation marks. R has converted 1 and TRUE into text representations. That's because 1 and TRUE are different _type_s than "Baldwin". There are four basic types of variables your likely to encounter in this class, listed here in heirarchical order:

1. logical: TRUE or FALSE
2. integer: 0L, -1L, 1L, etc. A (real) number without a decimal part. Technical note: they take up less space in the computer than numbers with decimals.
3. numeric: pi, 0.34, 1.4043, etc. A real number.
4. character: "funky town", "delicate", etc.

Vectors are converted to the highest number on this list present – x above has "three" so the whole vector becomes a character.

Vector Arithmetic and Functions

Vectors make it easy to do many computations all at once – adding one to a list of numbers, dividing all of them by 3, etc. And as long as two vectors are the same length, we can combine them in natural ways:

```
x = c(1, 2, 3)
x + 4
## [1] 5 6 7
x/3
## [1] 0.3333333 0.6666667 1.0000000
-x
```

```
## [1] -1 -2 -3
x^3
## [1]  1  8 27
y = c(3, 2, 1)
x - y
## [1] -2  0  2
x * y
## [1] 3 4 3
x/y
## [1] 0.3333333 1.0000000 3.0000000
x > 2
## [1] FALSE FALSE  TRUE
x >= 2
## [1] FALSE  TRUE  TRUE
```

Just like in math, a function is a way of mapping input to output, and just like in most math classes, you can spot functions since they use parentheses: (). We've already seen the _c_oncatenate function c used (for example) to create vectors.

We can also apply any number of ubiquitous functions to our vector input. Just a small taste:

```
x = c(1, 2, 3)
#sum: add up the elements of a vector
sum(x)
## [1] 6
#Just like you can use the command sum to add up the
#  elements of a numeric vector, you can use
#  prod to take their product:
prod(x)
## [1] 6
sqrt(x)
## [1] 1.000000 1.414214 1.732051
y = c(-1, 2, 4)
#abs: absolute value
abs(y)
## [1] 1 2 4
#exp: exponential. exp(x) is e^x
exp(y)
## [1]  0.3678794  7.3890561 54.5981500
#log: _natural_ logarithm (base e)
log(x)
## [1] 0.0000000 0.6931472 1.0986123
#Note that these functions interpret their input
#  as *radians* rather than degrees.
sin(x) + cos(y)
## [1]  1.3817733  0.4931506 -0.5125236
max(y)
## [1] 4
min(y)
## [1] -1
range(y)
## [1] -1  4
mean(x)
## [1] 2
median(x)
```

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

Another thing that we will do all the time is use regularly-spaced sequences of numbers. These are created in R with : or seq:

```
x = 1:10
x
##  [1]  1  2  3  4  5  6  7  8  9 10
y = 10:1
y
##  [1] 10  9  8  7  6  5  4  3  2  1
#some times the gap is not 1
z = seq(0, 1, by = .02)
z
##  [1] 0.00 0.02 0.04 0.06 0.08 0.10 0.12 0.14 0.16 0.18 0.20 0.22 0.24 0.26
## [15] 0.28 0.30 0.32 0.34 0.36 0.38 0.40 0.42 0.44 0.46 0.48 0.50 0.52 0.54
## [29] 0.56 0.58 0.60 0.62 0.64 0.66 0.68 0.70 0.72 0.74 0.76 0.78 0.80 0.82
## [43] 0.84 0.86 0.88 0.90 0.92 0.94 0.96 0.98 1.00
#other times we care less about the gap and more
#  more about how many points we get out
w = seq(0, 1, length.out = 20)
```

In addition to math/arithmetic functions, there is a litany of basic programming functions that you're likely to use all of the time:

```
x = 99:32
#length: how many elements (items) are there in x?
length(x)
## [1] 68
y = c("hey despacito!", "must dance with you today")
#what TYPE of variable does R think this is?
class(y)
## [1] "character"
#rep: repeat/reproduce
rep(y, 4)
[1] "hey despacito!"            "must dance with you today" "hey despacito!"
"must dance with you today"
[5] "hey despacito!"            "must dance with you today" "hey despacito!"
"must dance with you today"
#head/tail: display only the beginning/end
#  of an object -- very useful for very
#  large objects
x = 1:100000
head(x)
## [1] 1 2 3 4 5 6
tail(x)
## [1]  99995  99996  99997  99998  99999 100000
```

Subsetting Vectors: [

Often we want to examine only part of a vector, most commonly the part of a vector that satisfies some condition, but also looking at the first or last few elements. To do this we *extract* or subset those elements by using [:

```
x = c(5, 4, 1)
x[1]
## [1] 5
x[3]
## [1] 1
x[1:2]
## [1] 5 4
x[2:3]
## [1] 4 1
```

In the syntax x[something], note that *something is itself a vector*! So the above is all short-hand for the more complicated types of subsets:

```
x = 20:30

x

##  [1] 20 21 22 23 24 25 26 27 28 29 30

x[c(1, 3, 5)]

## [1] 20 22 24

x[c(5, 9)]

## [1] 24 28

x[seq(1, 10, by = 2)]

## [1] 20 22 24 26 28
```

Besides being an integer, something can be a *logical vector* of the same length as the vector itself:

```
x = c(5, 6, 7)

x[c(TRUE, TRUE, FALSE)]

## [1] 5 6

x[c(FALSE, TRUE, FALSE)]

## [1] 6

x[c(FALSE, FALSE, TRUE)]

## [1] 7

X=c(7,2,6,9,4,1,3)

Y=X<3|X>=6
```

Most commonly we'll do something that's identical to the above but reads more naturally:

```
x = c(-1, 0, 1)

x > 0
```

```
## [1] FALSE FALSE  TRUE
x[x > 0]
## [1] 1
x[x <= 0]
## [1] -1  0
```

We can also *replace* parts of a vector by subsetting:

```
x = c(-1, 5, 10)
x[3] = 4
x
## [1] -1  5  4
x[x < 0] = 0
```

Named Vectors

It's also often useful to *name* our vectors to help organize the information. Suppose we were keeping track of the ages of the Trumps:

```
trump_ages = c(70, 46, 38, 34, 32, 22, 9)
```

This is nice, but much more useful if we keep track of who each element represents:

```
trump_ages = c(Donald = 70, Melania = 46, Donald_Jr = 38, Ivanka = 34,
               Eric = 32, Tiffany = 22, Barron = 9)
trump_ages
##    Donald   Melania Donald_Jr    Ivanka      Eric   Tiffany    Barron
##        70        46        38        34        32        22         9
```

We can also use the names function to assign names; this is sometimes easier, e.g., if the names have spaces:

```
names(trump_ages) = c("Donald", "Melania", "Donald, Jr.", "Ivanka", "Eric", "
Tiffany", "Barron")
trump_ages
##      Donald     Melania Donald, Jr.      Ivanka        Eric     Tiffany
##          70          46          38          34          32          22
##      Barron
##          9
```

This also makes code for subsetting much easier to read, since we can subset by the names:

```
trump_ages["Donald"]
## Donald
##     70
trump_ages[c("Donald", "Barron")]
## Donald Barron
##     70      9
```

Lists

We saw above that R doesn't like vectors to have different types: c(TRUE, 1, "Drake") becomes c("TRUE", "1", "Drake"). But storing objects with different types is absolutely fundamental to data analysis.

R has a different type of object besides a vector used to store data of different types side-by-side: a list:

```
x = list(TRUE, 1, "Drake")
x
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] "Drake"
```

Note how different the output looks, as compared to using c!! The quotation marks are gone except for the last component. You can ignore the mess of [[and [for now, but as an intimation, consider some more complicated lists:

```
x = list(c(1, 2), c("Selina", "Bieber"), c(TRUE, FALSE), c(5L, 6L))
x
## [[1]]
## [1] 1 2

## [[2]]

## [1] "Selina" "Bieber"

## [[3]]

## [1]  TRUE FALSE

## [[4]]

## [1] 5 6

y = list(list(1, 2, 3), list(4:5), 6)

y

## [[1]]

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

## [1] 1

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

## [1] 2

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

## [1] 3

## [[2]]

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

## [1] 4 5

## [[3]]

## [1] 6

Mylist=list(A=seq(8,36,4), campus=”Huskie”,idm=diag(3))

Mylist

names(Mylist)

Mylist$A
```

x is a list which has 4 components, each of which is a vector with 2 components. This gives the first hint at how R treats a dataset with many variables of different types – at core, R stores a data set in a list!

y is a nested list – it’s a list that has lists for some of its components. This is very useful for more advanced operations, but probably won’t come up for quite some time, so don’t worry if you haven’t wrapped your head around this yet.

Matrices

# Generating matrix A from one vector with all values

```
v <- c(2,-4,-1,5,7,0)

( A <- matrix(v,nrow=2) )

# Generating matrix A from two vectors corresponding to rows:

row1 <- c(2,-1,7); row2 <- c(-4,5,0)

( A <- rbind(row1, row2) )

# Generating matrix A from three vectors corresponding to columns:

 col1 <- c(1,6); col2 <- c(2,3); col3 <- c(7,2)

( AT40 <- cbind(col1, col2, col3) )

# Giving names to rows and columns ( create matrix from AT40 of Ryan Seacrest):

colnames(AT40) <- c(“Drake",“Maroon5",“Selina")

rownames(AT40) <- c(“weekTop",“Peak")

AT40

# Indexing for extracting elements (still using AT40 from above):

AT40[2,1]

AT40[,2]

AT40 [,c(1,3)]

AT40[2,c(1,2)]

AT40 [2,]

# Diaginal and identity matrices:

diag( c(4,2,6) )

diag( 3 )

# Generating matrix A and B
A <- matrix( c(2,-4,-1,5,7,0), nrow=2)
B <- matrix( c(2,1,0,3,-1,5), nrow=2)
```

```
A

B

A*B

# Transpose:

(C <- t(B) )

# Matrix multiplication:

 (D <- A %*% C )

# Inverse:

 solve(D)

# Giving names to rows and columns:
B = matrix(c(2, 4, 3, 1, 5, 7), nrow=3,ncol=2)
C = matrix(c(7, 4, 2),nrow=3,ncol=1)
# combine the columns of B and C with cbind
cbind(B, C)
# combine the rows of two matrices if they have the same number of
columns
D = matrix(c(6, 2),nrow=1,ncol=2)
rbind(B, D)
```

Working with Data Frames from scratch

```
##A data frame is used for storing data tables

topHit = c(1, 3, 5)

s = c(“Drake", “Swift", “Selina")

at40 = c(TRUE, FALSE, TRUE)

df = data.frame(topHit, s, at40)

# Define one x vector for all:

year     <- c(2008,2009,2010,2011,2012,2013)

# Define a matrix of y values:

product1<-c(0,3,6,9,7,8); product2<-c(1,2,3,5,9,6); product3<-
c(2,4,4,2,3,2)

sales_mat <- cbind(product1,product2,product3)

rownames(sales_mat) <- year

# The matrix looks like this:

sales_mat
```

```
# Create a data frame and display it:

sales <- as.data.frame(sales_mat)

sales

# Accessing a single variable:

sales$product2

# Generating a new  variable in the data frame:

sales$totalv1 <- sales$product1 + sales$product2 + sales$product3

# The same but using "with":

sales$totalv2 <- with(sales, product1+product2+product3)

# The same but using "attach":

attach(sales)

sales$totalv3 <- product1+product2+product3

detach(sales)

# Result:

Sales


# Subset: all years in which sales of product 3 were >=3

subset(sales, product3>=3)

# Note: "sales" is defined in Data-frames.R, so it has to be run
first!

# save data frame as RData file (in the current working directory)

save(sales, file = "oursalesdata.RData")

# remove data frame "sales" from memory

rm(sales)

# Does variable "sales" exist?

exists("sales")

# Load data set  (in the current working directory):

load("oursalesdata.RData")

# Does variable "sales" exist?

exists("sales")

sales
```

```
# averages of the variables:
Head, str, colMeans(sales)
```

Build-in Data Frame

```
#For example, here is a built-in data frame in R,called mtcars
Mtcars
nrow(mtcars)
ncol(mtcars)
mtcars["Mazda RX4", "cyl"]
head(mtcars)
```

Getting Help & Packages

- ☐ **R truly exceptional with its vast library of user-contributed packages**
- ☐ **All of the packages available to R users (through CRAN) are completely free of charge**

install.packages("data.table")

library(data.table)

require(data.table)

- ☐ **Google "tutorial data.table"**
- ☐ **CRAN Task Views (https://cran.r-project.org/web/views/)**
- ☐ **https://rseek.org/**
- ☐ **https://www.r-project.org/mail.html**
- ☐ **https://www.rstudio.com/resources/cheatsheets/**
- ☐ **https://www.r-bloggers.com/**
- ☐ **https://stats.idre.ucla.edu/other/dae/**
- ☐ **Edx and Coursera online course on R/probability and statistics/Data science**

Packages

One of the things that makes R truly exceptional is its vast library of user-contributed packages.

R comes pre-loaded with a boat-load of the most common functions / methods of analysis. But in no way is this congenital library complete.

Complementing this core of the most common operations are external *packages*, which are basically sets of functions designed to accomplish specific tasks.

Best of all, unlike some super-expensive programming languages, all of the thousands of packages available to R users (most importantly through CRAN,
the **C**omprehensive **R** **A**rchive **N**etwork) are *completely free of charge*.

The two most important things to know about packages for now is where to find them, how to install them, and how to load them.

We'll work extensively (if we want to work on data science stuffs in addition to our econometrics) with the data.table package, which was built for working with huge data sets.

Where to find packages

Long story short: Google. Got a particular statistical technique in mind? The best R package for this is almost always the top Google result if asked correctly.

How to install packages

Just use install.packages!

```
install.packages("data.table")
```

This will download the code from the package to your computer to a place that R understands.

We do *not* yet have access to the functions in the package. We have to load it first.

How to load packages

Just add it to your library!

```
library(data.table)
```

Et voila! You'll now have access to all of the awesome functions in the data.table package. You can also Google "tutorial data.table" (or in general "tutorial [package name]") and you're very likely to find a trove of sites trying to help you learn the package.

data.tables

Data sets are the lifeblood of a data lover!

As mentioned above, data sets in R basically lists where every element has the same length. In basic R, this is done with a data.frame (just seen before), but it'll also be easier for a beginner to understand the syntax of a data.table, so you can forget about data.frames for now.

We can build a data.table from scratch with the data.table command. This command lets you build up a data.table from several vectors of the same length:

```
##building a data table

fonsi = 1:5

yankee = 2 * fonsi

fonsi.yankee = data.table(fonsi, yankee)

fonsi.yankee

bieber = -4:0
```

```
data.table(fonsi, yankee, bieber)
```

Subsetting data

When you're working with data, you'll often want to look at subsets that satisfy a particular condition. First we'll set up a simple data.table:

```
##set up a simple data.table
location = c("Toronto", "LA", "Texas", "Toronto", "London")
earnings = c(83, 86, 69, 94, 30)
popstar = c("Bieber", "Maroon5", "Selina", "Drake", "DuaLipa")
concert = c(50, 45, 65, 40, 50)
popUSA = data.table(location, earnings, popstar, concert)
popUSA
popUSA[location == 'Toronto']
popUSA[earnings > 30]
popUSA[concert <=40]
```

```
location = c("New York", "Chicago", "Boston", "Boston", "New York")

salary = c(70000, 80000, 60000, 50000, 45000)

title = c("Office Manager", "Research Assistant", "Analyst", "Office Manager"
, "Analyst")

hours = c(50, 56, 65, 40, 50)

jobsearch = data.table(location, salary, title, hours)

jobsearch
```

Now, suppose you wanted to see only the jobs in New York. You could select them as follows:

```
jobsearch[location == 'New York']
##    location salary          title hours

## 1: New York  70000 Office Manager    50

## 2: New York  45000        Analyst    50
```

Notice the use of the *double equal sign*. This command is *testing a logical condition*. If you use a single equals sign, this won't work since = is what is used to name the arguments to a function in R. The preceding command looks at the data.table jobsearch and then the column location and checks which entries satisfy the condition that the location is "New York". Finally, the function returns *only these rows* of the data.table.

Now suppose you wanted to extract only those jobs that pay more than $50,000. The command for this is as follows:

```
jobsearch[salary > 50000]
```

Finally, suppose the most you're willing to work per week is 50 hours. Here are the jobs you should consider:

```
jobsearch[hours <= 50]
```

More on data.table Access

One of the most fundamental R skills you'll need to learn is how to access parts of a data.table or vector. This can be a little confusing at first since there are usually *several different ways to accomplish the same thing*. This section is intended to add more clarity to some of the material on data.tables from above. The *best way to learn this is to play around with different commands and see what happens*. There may be an exercise/quiz at the end of this tutorial in next week.

First I'll build a simple data.table from the following vectors:

```
person = c("Linus", "Snoopy", "Lucy", "Woodstock")

age = c(5, 8, 6, 2)

weight = c(40, 25, 50, 1)

my.data.table = data.table(person, age, weight)

my.data.table
```

Fact #1: You can use the same principles to select subsets of vectors and data.tables *by position*

The only real difference here is that vectors are one-dimensional

```
age[1:2]

## [1] 5 8

age[c(1,3)]

## [1] 5 6
```

whereas data.tables are two-dimensional; the first dimension is rows:

```
my.data.table[1:2]

my.data.table[c(1, 3)]
```

The second dimension is columns; we can specify rows *and* columns by giving two numbers inside [ ]:

```
#what is the first row of the third column?
```

```
my.data.table[1, 3]
##     weight
## 1:      40
#what are the first three rows of the third column?
my.data.table[1:3, 3]
```

If you leave the part *before* the comma blank, you get all the rows:

```
my.data.table[ , 2:3]
```

If you leave the part *after* the comma blank (or don't include it at all), you get all the columns:

```
my.data.table[c(1,3), ]
my.data.table[c(2,4)]
```

Fact #2: There are *three ways* to access the columns of a data.table *by name*.

The first way is to use [["COLUMN NAME GOES HERE"]]

```
my.data.table[["weight"]]
## [1] 40 25 50  1
```

The second is to use $, which is often faster to type since it doesn't require the use of quotation marks:

```
my.data.table$weight
## [1] 40 25 50  1
```

Both of the preceding methods are limited in that they only allow us to reference a single column. We can reference multiple columns as follows:

```
my.data.table[ , c("person", "weight")]
```

Since we left the part before the comma blank, this gave us all the rows. We could get the same thing by accessing these columns *by position* (though this is generally not recommended)

```
my.data.table[ , 2]
my.data.table[ , c(1,2)]
my.data.table[ , 1:2]
```

In some cases it's easier to access columns of a data.table *by name* and in others it's easier to access them *by position*.

```
##How to sort out order of outcomes/data
Movie_ratings=c(3,4,2,2,3,1,1,4,2,2,1,1,4)
#order of movie ratings 1(PG), 2(PG13), 3(R), 4(NC17)
rating_data=factor(Movie_ratings,labels=c("PG","PG13","R","NC17"))
Movie_ratings
rating_data
```

```
# sorting examples using the mtcars dataset
attach(mtcars)

# sort by mpg
newdata <- mtcars[order(mpg),]

# sort by mpg and cyl
newdata <- mtcars[order(mpg, cyl),]

#sort by mpg (ascending) and cyl (descending)
newdata <- mtcars[order(mpg, -cyl),]

detach(mtcars)
```

Loading External Data & Data Summary

The vast majority of the time, you won't be using data that you type in by hand – you'll be importing data from external sources. One of the most common ways to find such data is in comma-separated format – such files are structured such that each row represents a row of data, and columns are separated by a comma (actually, any separating character is possible), e.g., like this:

```
name,age,company

Mike,24,BCG

Rodrigo,25,Uber

Frank,28,FMC

Ethan,22,AirBnB
```

It's very easy to read files like this into R very quickly using fread. The weather site Weather Underground offers lots of historical data in such tabular format. E.g., the data on this page about the weather recorded at Philadelphia International Airport is available as a .csv online here.

We can read this into R like so:

```
Require(data.table)

weather = fread('http://michaelchirico.github.io/philly_weather_data.csv')

weather

summary(weather)

names(weather)

file <- "http://www.ospo.noaa.gov/data/land/bbep2/biomass_burning.txt?filenam
e=LocalFile.txt.gz&dir=C:/R/Data"

Biomass_Burning_Data <- read.csv(file, header=TRUE)

This was an example of how to download the data from .txt file on Internet in
to R. But sometimes we come across tables in HTML format on a website.

library(quantmod)

getSymbols(Symbols = "QQQ", auto_assign = TRUE)

str(QQQ)

head(QQQ)

getSymbols(Symbols = "QQQ", src = "yahoo")

getSymbols(Symbols = "GDP", src = "FRED")

str(GDP)

series_name <- "UNRATE"

# Load the data using getSymbols

getSymbols(Symbols = series_name, src = "FRED")

head(UNRATE)

symbols <- c("AAPL", "MSFT", "IBM")

# Create new environment

data_env <- new.env()

# Load symbols into data_env

getSymbols(Symbols = symbols, env = data_env)

## [1] "AAPL" "MSFT" "IBM"

# Extract the close column from each object and merge into one xts object

close_data <- do.call(merge, eapply(data_env, Cl))

# View the head of close_data

head(close_data)
```

## We will use "Wooldridge" package very often in this course.

#Install Woolridge package

Loading Wooldridge (2016) Data

```
##read a data table using read.dta
require(foreign)
affairs <- read.dta("http://fmwww.bc.edu/ec-p/data/wooldridge/affairs.dta")
write.csv(affairs, "mydat.csv") or
write.xlsx(mydat, "mydat.xlsx")
mydat <- read.csv(file.choose())
head(mddat)
tail(mydat);
names(affairs); colMeans(affairs); sd(affairs$naffairs)
```

Ref: This document was prepared from various sources, and sources that need to be acknowledged are not limited to books and R-tutorials but include other internet materials.