



COMPUTATIONAL FINANCE & RISK MANAGEMENT

UNIVERSITY *of* WASHINGTON

Department of Applied Mathematics

R Fundamentals

CFRM 425 (003)

R Programming for Quantitative Finance

References/Reading/Topics

- Reading: Jeet and Vats, Ch 1 [JV] (through conclusion)
- Reference: Zuur, Ieno, and Masters [ZIM], Ch's 1, 6
- [JV] Publisher website information:
 - <https://www.packtpub.com/big-data-and-business-intelligence/learning-quantitative-finance-r> (general information)
 - <https://account.packtpub.com/getfile/9781786462411/code> (sample code)
- Topics:
 - R Types, Objects and Functions
 - Importing and exporting data
 - Writing R Code Expressions
 - A word from our sponsor

R Types, Objects and Functions



- `character`
- `numeric (double)`
- `integer`
- `logical`

- In general programming, we typically have container classes that contain one or more elements
 - Homogeneous containers
 - vector
 - matrix
 - array (arbitrary dimensions)
 - Heterogeneous containers
 - dataframe
 - list

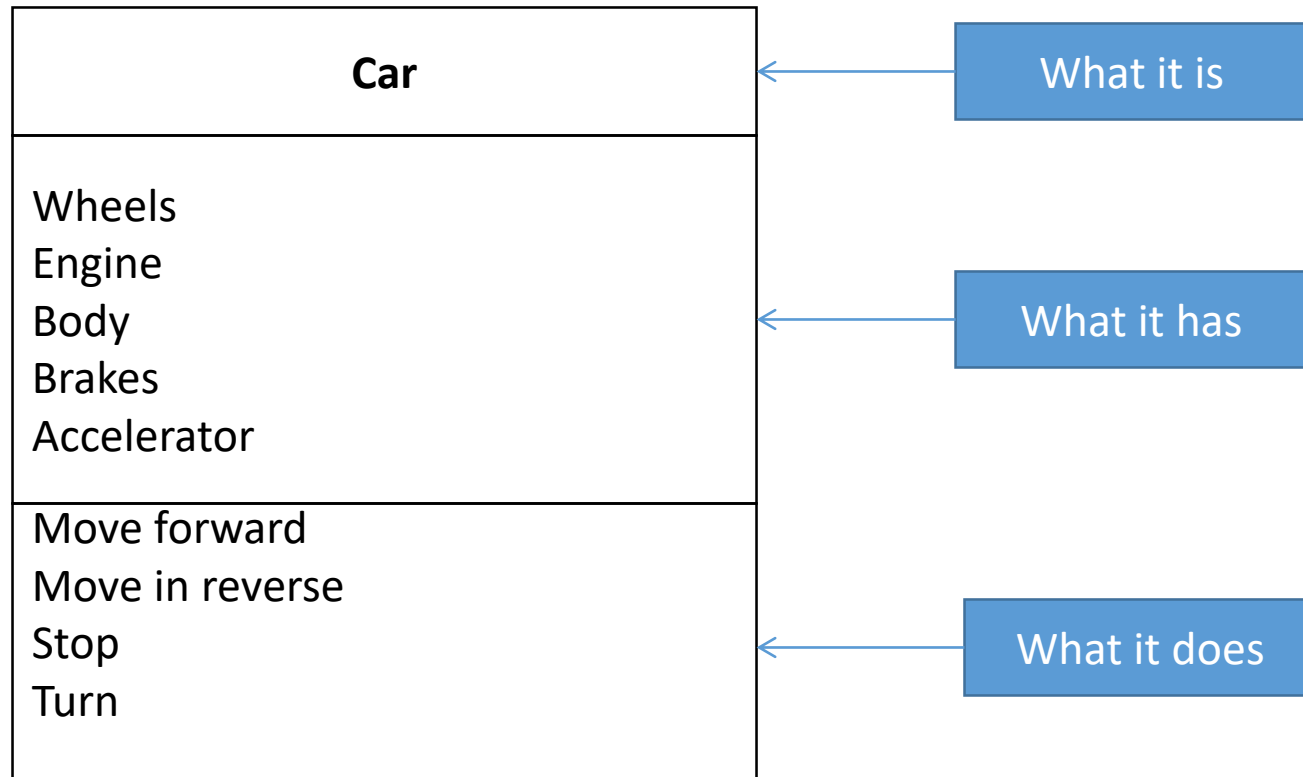
What is a Class? What is an Object?

- A Car (the class)
 - Has wheels, an engine, a steering wheel, a chassis a body, brakes, and an accelerator (member variables)
 - Moves forward, stops, turns (member functions)
- An object is an instance of the class
 - A Dino Ferrari 308 GT4 is an instance of the Car class
 - A Honda Accord is an instance of the Car class



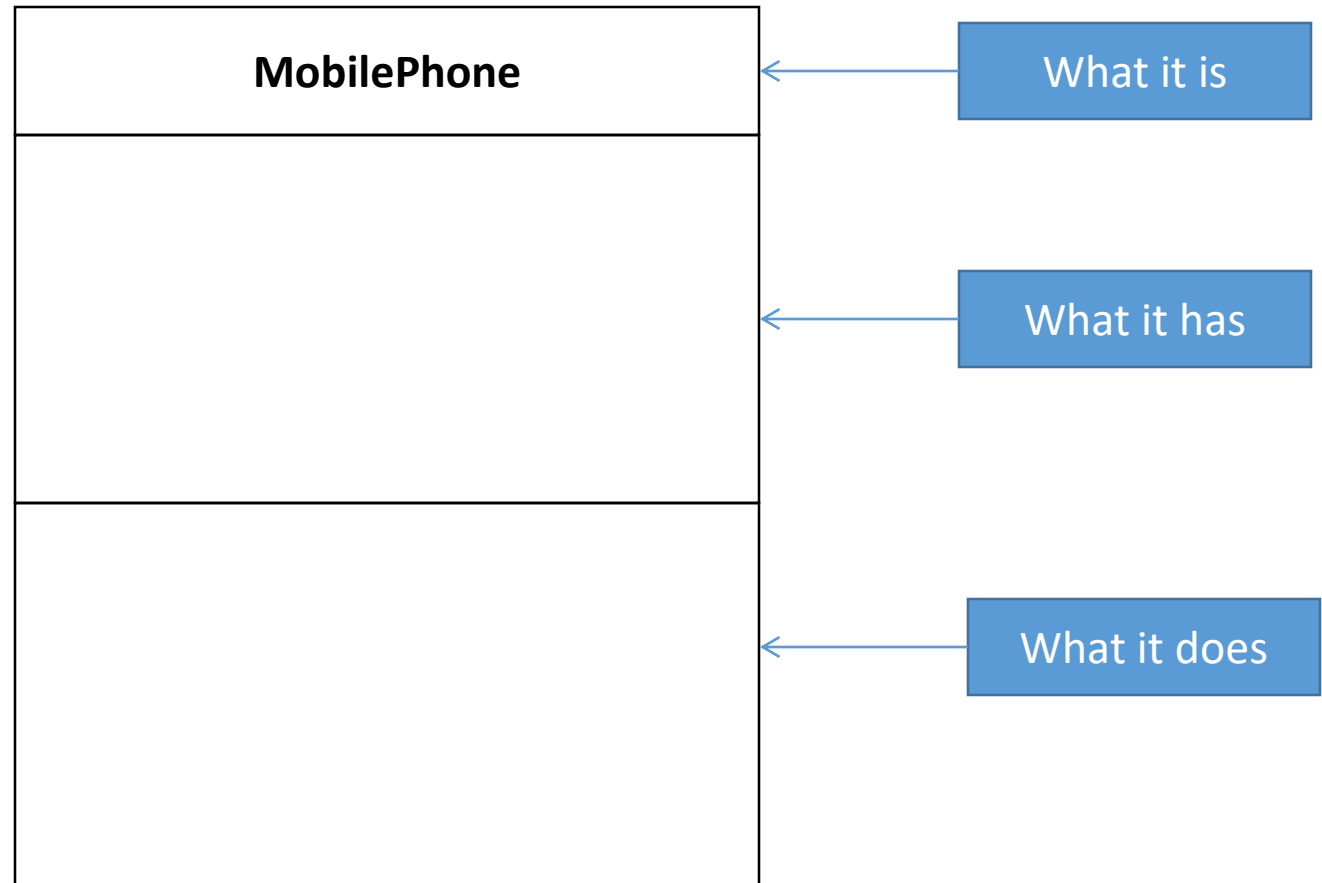
UML =

Unified Markup Language. Used heavily in object-oriented programming



Exercise: Mobile Phone

- This is a typical interview question...



R Container Classes/Objects

- A vector is initialized using the `c(.)` function in R, where *c* means to *combine* elements of like types into a vector container:

```
a <- "Quantitative"  
b <- "Finance"  
cv <- c(a,b)
```

- The individual elements are then accessed with the `[.]` operator and 1-based indexing:

```
cv[1]  
cv[2]
```

- Output:

```
> cv[1]  
[1] "Quantitative"  
> cv[2]  
[1] "Finance"
```

- A vector can similarly contain numeric types:

```
Var <- c(1.06, 2, 3.6)
```

- Then, use the individual elements in computations:

```
s <- v[2] + v[3]  
s
```

- Not surprisingly:

```
> s <- var[2] + var[3]  
> s  
[1] 5.6
```

- A matrix can be defined without data and then populated later using the `[i, j]` operator
- A matrix can also be defined with the data in a single vector, along with the following settings:
 - Column major or row major order (**byrow** = {TRUE, FALSE})
 - Number of rows or number of columns (**nrow** or **ncol**)
- Example:

```
M <- matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)
```

- Note the result is in **column major** order by default:

	[,1]	[,2]	[,3]
[1,]	1	3	5
[2,]	2	4	6

- A matrix can have data entered in row major order by putting the

byrow = TRUE

parameter setting (**FALSE** by default); viz,

```
M2 <- matrix(c(1,2,3,4,5,6), byrow = TRUE, nrow = 2)
```

	[,1]	[,2]	[,3]
[1,]	1	2	3
[2,]	4	5	6

- Matrix objects must be of homogeneous type; eg, all numeric, all character, etc
- Look at two examples of character type matrices, and random access by element, row, column, subsets:

```
msgData <- c("I", "love", "R", "more", "than", "Matlab")
(msgMtx <- matrix(msgData, byrow = TRUE, nrow = 2))
#      [,1] [,2] [,3]
# [1,] "I"  "love" "R"
# [2,] "more" "than" "Matlab"
```

```
(msgMtx2 <- matrix(msgData, byrow = TRUE, nrow = 3))
#      [,1] [,2]
# [1,] "I"  "love"
# [2,] "R"  "more"
# [3,] "than" "Matlab"
```

- **msgMtx:**

```
      [,1]      [,2]      [,3]  
[1,] "I"       "love"    "R"  
[2,] "more"    "than"    "Matlab"
```

- Results from previous slide:

```
> msgMtx[1, 2]  
[1] "love"  
> msgMtx[1, ]  
[1] "I"       "love"    "R"  
> msgMtx[, 2]  
[1] "love"    "than"
```

- **msgMtx2:**

```
      [,1]      [,2]  
[1,] "I"       "love"  
[2,] "R"       "more"  
[3,] "than"    "Matlab"
```

- Results from previous slide:

```
> msgMtx2[c(2, 3), ]  
      [,1]      [,2]  
[1,] "R"       "more"  
[2,] "than"    "Matlab"
```

R Container Classes/Objects

- **Dataframes:** Like an R matrix, but columns may have heterogeneous data types
- Have other properties that we will discuss later on in the course

- Example:

```
Name = c("Alexandra", "John", "Bob")
Age = c(18, 20, 23)
Gender = c("F", "M", "M")
```

```
(data <- data.frame(Name, Age, Gender))
#      Name Age Gender
# 1 Alexandra 18      F
# 2      John 20      M
# 3       Bob 23      M

data[, c(1, 3)]
```

- 1st and 3rd columns:

	Name	Gender
1	Alexandra	F
2	John	M
3	Bob	M

R Container Classes/Objects

- Individual columns on a dataframe are member variables on the dataframe object
- These member variables are vector types
- They can be accessed with the \$ operator
- The square bracket operator may be used just as that on a vector to access individual elements or a subset of column elements

```
data$Name  
data$Age[c(2,3)]  
data$Gender[1]
```

- Results:

```
> data$Name  
[1] Alexandra John      Bob  
Levels: Alexandra Bob John  
> data$Age[c(2,3)]  
[1] 20 23  
> data$Gender[1]  
[1] F  
Levels: F M
```

R Container Classes/Objects

- Lists: A generalized dataframe
 - Heterogeneous data
 - “Columns” need not be of equal length
 - Sometimes called a “jagged array” in other programming languages
- Example: Create a list with the data shown here:

```
List1 = list(c(4L,5L,6L), "Hello", c(24.5, 53.63))
```

- Accessing each “column” (vector) of data, instead of the \$ operator, we use double square brackets, as shown:

```
List1[[1]]  
List1[[2]]  
List1[[3]]
```

- Results:

```
> List1[[1]]  
[1] 4 5 6  
> List1[[2]]  
[1] "Hello"  
> List1[[3]]  
[1] 24.50 53.63
```

- Recall from our earlier dataframe example:

```
> (data <- data.frame(Name, Age, Gender))  
      Name Age Gender  
1 Alexandra 18      F  
2      John 20      M  
3       Bob 23      M  
> data$Gender[1]  
[1] F  
Levels: F M
```

- The character data F and M are automatically stored as *factor* or *categorical* data
- This is not the case, however, for:
 - Character data in arrays
 - vectors
 - matrices
 - Integer data in dataframes or arrays (where we want integers to represent factor data)

- Enter the **factor(.)** function
- Along with its friends
 - **levels(.)** Returns the unique level in a “column”
 - **nlevels(.)** Returns the numbers of levels in a “column”

```
a <-c(2, 3, 4, 2, 3)
fact <-factor(a)
```

- Integers as factor variables are now stored in the vector **fact**

```
[1] 2 3 4 2 3
Levels: 2 3 4
```

- Then, verify:

```
levels(fact)
nlevels(fact)
```

- Output:

```
> levels(fact)
[1] "2" "3" "4"
> nlevels(fact)
[1] 3
```

Factors

- Similar for character data – eg chicken wing spices:

```
# wing sauces:
```

```
spiceLevels <- c("low", "medium", "hot", "insanity")
```

```
levels(spiceLevels) # NULL
```

```
spiceLevels <- factor(spiceLevels)
```

```
> spiceLevels
```

```
[1] low      medium    hot       insanity
```

```
Levels: hot insanity low medium
```



```
levels(spiceLevels)
```

```
nlevels(spiceLevels)
```

```
> levels(spiceLevels)
```

```
[1] "hot"      "insanity" "low"      "medium"
```

```
> nlevels(spiceLevels)
```

```
[1] 4
```



Importing and exporting data

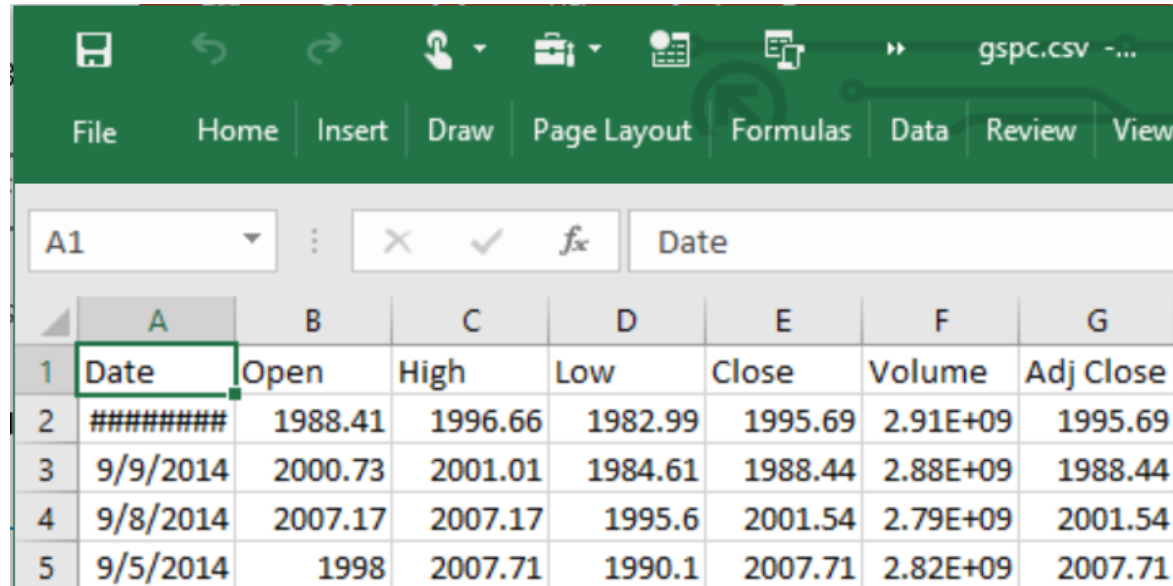


.csv and Excel file data I/O

- The text covers Importing and exporting different data types
- We will concentrate on two formats related to Excel
 - .csv (comma delimited text) files
 - .xlsx Excel files
- Data transfer functions for .csv files are included in base R
- For Excel files, we need to install a separate R package, xlsx, plus Java if it is not yet installed on your machine
- Time permitting, we will return to database I/O later

Base R Functions

- Given a file of S&P 500 market data, GSPC.CSV



	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Volume	Adj Close
2	#####	1988.41	1996.66	1982.99	1995.69	2.91E+09	1995.69
3	9/9/2014	2000.73	2001.01	1984.61	1988.44	2.88E+09	1988.44
4	9/8/2014	2007.17	2007.17	1995.6	2001.54	2.79E+09	2001.54
5	9/5/2014	1998	2007.71	1990.1	2007.71	2.82E+09	2007.71

- `read.csv(.)` and `write.csv(.)`
- Import to an R dataframe
- Export subset to another .csv file

Convert closing prices to log returns:

```
dir <- "C:/temp/data"
```

```
# read.csv and write.csv examples
```

```
gspc <- read.csv(file.path(dir, "gspc.csv"))
```

```
gspc.close <- gspc[, "Close"] # implicit conversion to vector
```

```
log.gspc <- log(gspc.close[-1]/gspc.close[-length(gspc.close)])
```

```
dates <- as.vector(gspc[, "Date"])
```

```
dates <- dates[-1]
```

```
rtnData <- as.data.frame(cbind(dates, as.numeric(log.gspc)))
```

```
colnames(rtnData) <- c("Date", "Return")
```

```
write.csv(rtnData, file.path(dir, "gspcRtns.csv"), row.names = FALSE)
```

The xlsx R package

- The xlsx package allows us to do similar things as we can with `read.csv(.)` and `write.csv(.)`, but with the convenience of staying in “Excel space” and not having to convert to .csv files first
- One can also access data from, and set results in, particular worksheets in a file.
- It is a robust and proven package, widely used in practice and well-supported => **reliable**
- Unlike the typical R package, however, one needs to do a little extra work to install it, although this is relatively painless compared to other R/Excel utilities (see next page)
- An excellent online tutorial may be found here:
<http://www.sthda.com/english/wiki/r-xlsx-package-a-quick-start-guide-to-manipulate-excel-files-in-r>

The xlsx R package

- Installation of xlsx requires that you have Java installed on your Windows machine
- It is freely available from Oracle (non-commercial): <https://www.java.com/en/download/>
- ***HOWEVER***, if you are running a 64-bit version of R, you will need to make sure to install the 64-bit version of Java; the link above defaults to the 32-bit version (and the resulting error gives you no clue as to what the problem is!)
- Therefore, *make sure to drill down to the All Java Downloads page, and download the 64-bit version*: <https://www.java.com/en/download/manual.jsp#win>
- After installing Java, install and load the xlsx package as you would for any other R package.

JSJ | <https://www.java.com/en/download/manual.jsp#win>

Java™ Download Help

Available Operating Systems

- » [Windows](#)
- » [Mac](#)
- » [Linux](#)
- » [Solaris](#)

Help Resources

- » [Troubleshoot Java](#)

Java 7

- » [Where can I get Java 7?](#)

JDK

- » [Looking for the JDK?](#)

Java Downloads for All Operating Systems

Recommended Version 8 Update 111
Release date October 18, 2016

Select the file according to your operating system from the list below to get the computer.

> [Remove Older Versions](#) > [What is Java](#)

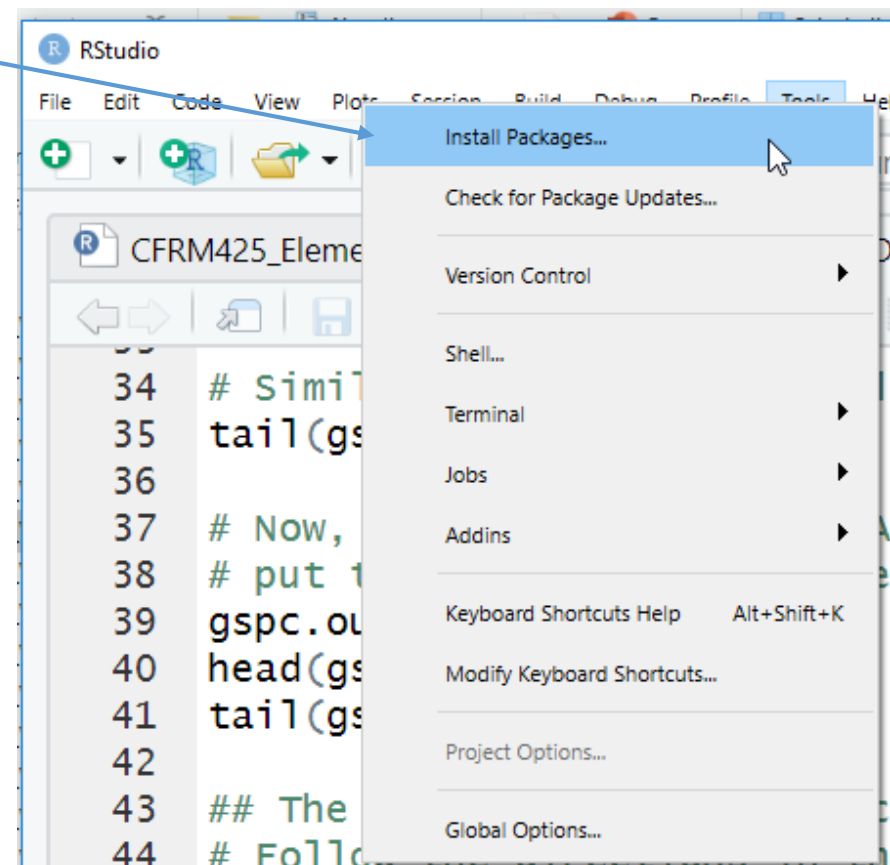
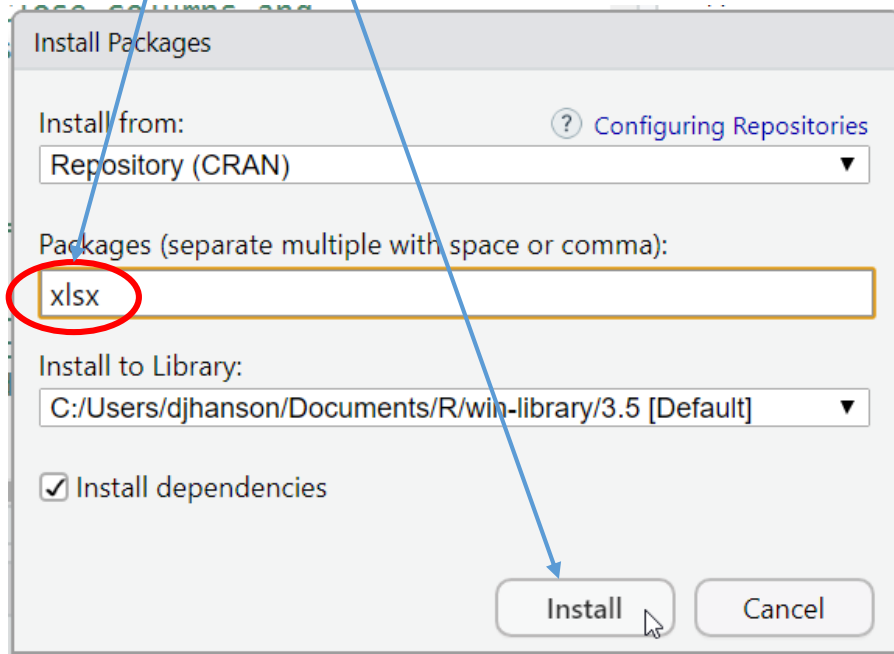
By downloading Java you acknowledge that you have read and accepted [license agreement](#)

Windows	Which should I choose?
Windows Online filesize: 721 KB	Instructions
Windows Offline filesize: 53.53 MB	Instructions
Windows Offline (64-bit) filesize: 60.31 MB	Instructions

The xlsx R package

- To install the xlsx package (or in general, any other package) from CRAN using Rstudio:

1. Choose Tools/Install Packages...
2. Type in the xlsx package name
3. Click on Install



The xlsx R package


```
# Set working directory to C:/temp/data
setwd("C:/temp/data")

library(xlsx)
file <- "GSPC.xlsx"

gspc.xl <- read.xlsx(file, 1)           # read first sheet

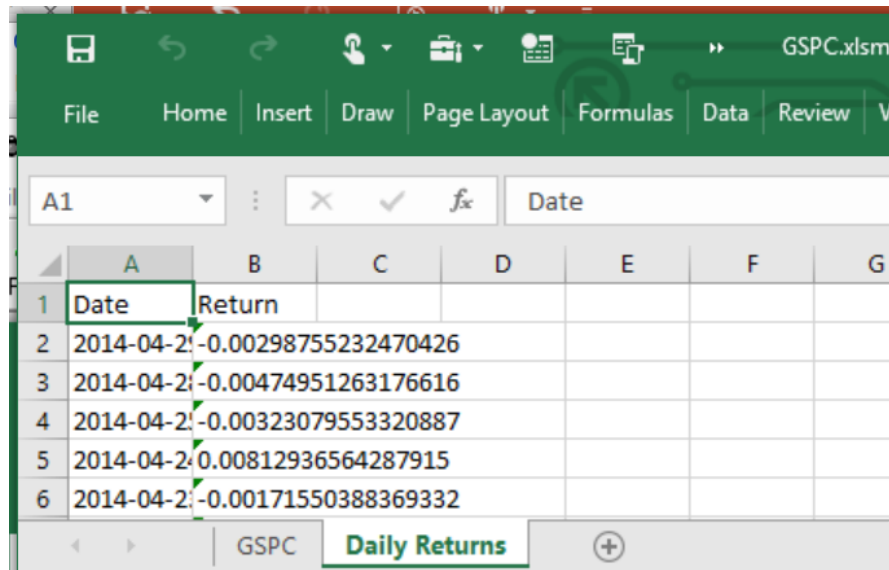
# Note we can use the column names or index numbers
# to access dataframe columns:
gspc.xl.close <- gspc.xl[, c("Date", "Close")]

# *** Be sure to put append = TRUE, lest your previous data be wiped out! ***
write.xlsx(gspc.xl.close, file = file, sheetName = "CloseData",
           col.names = TRUE, row.names = FALSE, append = TRUE)
```



The xlsx R package

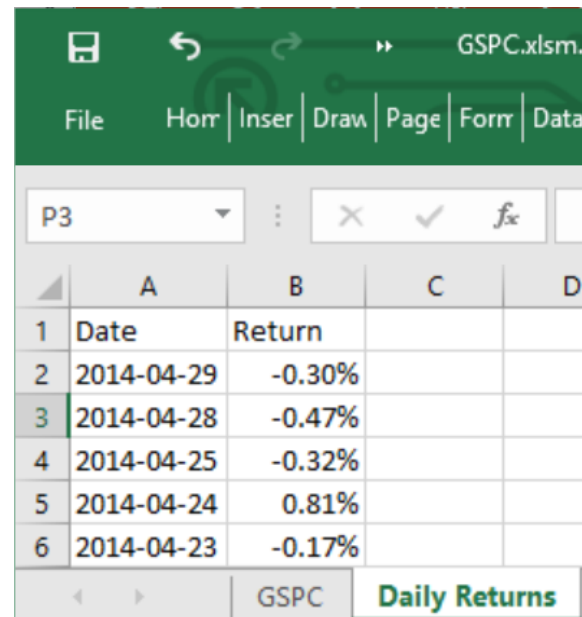
- Result:



This screenshot shows an Excel spreadsheet with the 'Daily Returns' worksheet selected. The data is organized in two columns: 'Date' and 'Return'. The 'Return' column contains long decimal values representing the daily returns. The spreadsheet is titled 'GSPC.xlsxm'.

	A	B	C	D	E	F	G
1	Date	Return					
2	2014-04-29	-0.00298755232470426					
3	2014-04-28	-0.00474951263176616					
4	2014-04-25	-0.00323079553320887					
5	2014-04-24	0.00812936564287915					
6	2014-04-23	-0.00171550388369332					

- Clean it up a bit and format:



This screenshot shows the same Excel spreadsheet as the previous one, but with the data formatted. The 'Date' column is formatted as dates, and the 'Return' column is formatted as percentages. The spreadsheet is titled 'GSPC.xlsxm'.

	A	B	C	D
1	Date	Return		
2	2014-04-29	-0.30%		
3	2014-04-28	-0.47%		
4	2014-04-25	-0.32%		
5	2014-04-24	0.81%		
6	2014-04-23	-0.17%		

- See CFRM425_DataImportExport_003.R
- Put **row.names = FALSE** inside the arguments of **write.xlsx(.);** otherwise, will get row numbers

Writing R Code Expressions



- Operations
 - Arithmetic
 - Conditional/Logical
- Keywords
- User-Defined Functions
- Conditional Branching
- Loops and Iteration
- Vectorized Functions
 - **apply(.)**
 - **sapply(.)**

Arithmetic Operations

- R has the following arithmetic operators:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
%	Modulus
<- or =	Assignment

- The usual rules of precedence apply, but round brackets are recommended for clarity

Conditional Operators (Real Numbers)

- R has the following conditional operators defined for scalars
- Each returns a Boolean type: TRUE or FALSE

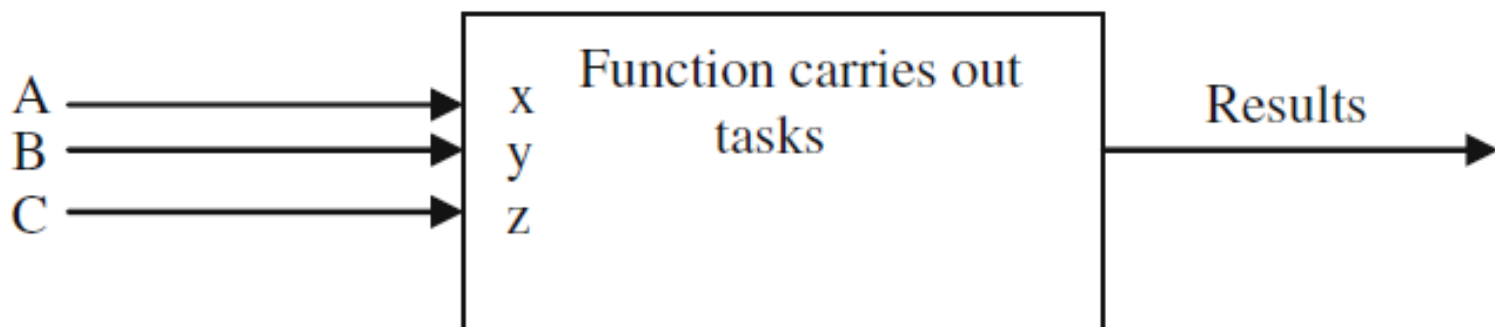
Operator	Description
==	Equality
> (>=)	Greater than (or equal)
< (<=)	Less than (or equal)
!=	Inequality
&&	Logical AND
	Logical OR
!	Logical NOT

- Operators **&** and **|** are also defined, but for vector operations (we will cover these soon)
- Make sure to distinguish between assignment (=) and equality (==)

- Some symbols are used to represent special values and cannot be reassigned:
 - **NA**: This is used to define missing or unknown values
 - **Inf**: This is used to represent infinity. For example, $1/0$ produces the result infinity
 - **NaN**: This is used to define the result of arithmetic expression which is undefined. For example, $0/0$ produces NaN
 - **NULL**: This is used to represent empty result
 - **TRUE** and **FALSE**: These are logical values and are generally generated when values are compared

Functions

- We are math people and we know what functions are; eg,
 - $f(x) = e^x$
 - $f(x) = \log(x + 15)$
 - $f(x) = \sin x$
 - $f(x) = \sqrt{x^2 + x + 1}$
- Presumably, you have programmed functions in other languages (Fig 6.3 from [ZIM]):



Functions

- The syntax for writing a function in R is as follows:

```
fcf <- function(function args) {  
  Your code goes here  
  # You can also put in comments  
  More code  
  Result to be returned is on the last line  
  Alternatively, you can put a return statement  
}
```

- We will look at the sample code in **CFRM425_Functions_Conditional_Statements_003.R**
- We will depart from the book here and just look at some real-valued functions, which are typically more relevant for traditional quant finance modeling
- Categorical data models are, however, can be relevant for areas such as
 - Credit risk models
 - Commercial/mortgage loan analyses
 - Regime switching (eg low interest to higher interest rate environment change)
 - Up/down market price movement prediction in trading models
 - Newer models related to predictive analytics (a growing area in quant finance)
- For a first look at functions in R, though, we'll keep it simple and defer categorical examples until later (time permitting)

Functions

- Some (contrived) examples:

```
trigFcn <- function(x)
{
  y <- 1.06 * x
  sin(y) + cos(y)
}

# Alternatively:
trigFcnAlt <- function(x)
{
  w <- x + 58.63
  return(sin(w) + cos(w))
}
```

- Source the function definitions (highlight and ctrl + enter)
- Now, call them (highlight and ctrl + enter):

```
trigFcn(-5.6)
u <- trigFcnAlt(-54.3)
u
```

- Results in console:

```
> trigFcn(-5.6)
[1] 1.280587
> u <- trigFcnAlt(-54.3)
> u
[1] -1.300914
```


Functions

- A function with more than one variable:

```
multiTrig <- function(x, y, z)
{
  return(sin(x) + cos(y) + tan(z))
}
```

- Source the function definition (highlight and ctrl + enter)
- Now, call it (highlight and ctrl + enter, or place cursor on same line and ctrl + enter):

```
multiTrig(0.1, 0.2, 0.5)
```

- Result in console:

```
> multiTrig(0.1, 0.2, 0.5)
[1] 1.626202
```

- Note that it is legal to put the arguments in a different order, as long as the variable names are included:

```
multiTrig(y = 0.2, z = 0.5, x = 0.1)
```

- Result in console is the same:

```
> multiTrig(y = 0.2, z = 0.5, x = 0.1)
[1] 1.626202
```

Functions

- Some (contrived) examples with no input arguments:

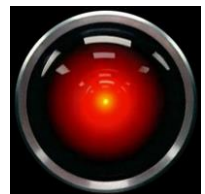
```
Dave <- function()  
{  
  print("Open the pod bay door HAL!")  
}  
  
HAL <- function()  
{  
  print("I'm sorry Dave; I'm afraid I can't do that.")  
}
```

- Again, source the function definitions (highlight and ctrl + enter)
- Now, call them (highlight and ctrl + enter):

```
Dave()  
HAL()
```

- Results in console:

```
> Dave()  
[1] "Open the pod bay door HAL!"  
> HAL()  
[1] "I'm sorry Dave; I'm afraid I can't do that."
```



Conditional Statements

- Basically three types of conditional statements:
 - if
 - if/else
 - if/else if/else
- A more advanced method is the **case_when** function for handling multiple else if conditions
 - Cleaner and more easily maintainable
 - Similar to a switch/case statement in languages such as C++
 - Will defer for now
 - See the following if you wish:
https://www.rdocumentation.org/packages/dplyr/versions/0.7.6/topics/case_when

- Syntax

- if statement

```
if(condition) {  
    do stuff ...  
}
```

- if/else statement

```
if(condition) {  
    do stuff if condition is TRUE...  
} else {  
    do something else ...  
}
```

- if/else if/else

```
if(condition1) {  
    do stuff if condition1 is TRUE...  
} else if(condition2) {  
    do stuff if condition2 is TRUE...  
} else {  
    do stuff if none of the above conditions holds  
}
```

Conditional Statements - Examples

- In these examples, inputs may be real numbers or vectors
 - New stuff:
 - The length of a vector **v** is obtained by **length(v)**
 - The **stop(.)** function will terminate a program, with the message in the argument that is displayed at termination (an error-handling example)

```
x.vec <- c(1:10)          # 10 elements
y.vec <- c(11:20)         # 10 elements
y.bogus <- c(12:20)       # 9 elements

g <- function(x, y) {
  if(length(x) != length(y)) {
    stop("ERROR: vector inputs are not of same length")
  }
  x + y
}
```

```
g(x.vec, y.vec)
g(x.vec, y.bogus)  # Function stops -- as desired
```

```
> g(x.vec, y.vec)
[1] 12 14 16 18 20 22 24 26 28 30
> g(x.vec, y.bogus)  # Function stops -- as desired
Error in g(x.vec, y.bogus) : ERROR: vector inputs are not of same length
```

- if/else:

```
g.else <- function(x, y) {  
  if(length(x) != length(y)) {  
    stop("ERROR: vector inputs are not of same length")  
  } else {  
    x + y  
  }  
}
```

```
g.else(x.vec, y.vec)      # OK!  
g.else(x.vec, y.bogus)    # Get error, by design
```

```
> g.else(x.vec, y.vec)      # OK!  
[1] 12 14 16 18 20 22 24 26 28 30  
> g.else(x.vec, y.bogus)    # Get error, by design  
Error in g.else(x.vec, y.bogus) :  
  ERROR: vector inputs are not of same length
```

Conditional Statements - Examples

- **if/else if/else:**
- Remark: if we call `g.else(3.2, y.vec)`, we will get an error
- However, scalar + vector addition is defined in R

`1.0 + c(10.1, 11.1, 12.1) = 11.1 12.1 13.1`

- We can allow it in our code as follows:

```
g.else.if <- function(x, y) {  
  if(length(x) == 1 || length(y) == 1) {  
    x + y  
  } else if(length(x) != length(y)) {  
    stop("ERROR: vector inputs are not of same length")  
  } else {  
    x + y  
  }  
}
```

```
g.else.if(x.vec, y.vec)      # OK!  
g.else.if(3.2, y.vec)       # OK!  
g.else.if(x.vec, y.bogus)   # Get error, by design
```

```
> g.else.if(x.vec, y.vec)    # OK!  
[1] 12 14 16 18 20 22 24 26 28 30  
> g.else.if(3.2, y.vec)     # OK!  
[1] 14.2 15.2 16.2 17.2 18.2 19.2 20.2 21.2 22.2 23.2  
> g.else.if(x.vec, y.bogus) # Get error, by design  
Error in g.else.if(x.vec, y.bogus) :  
  ERROR: vector inputs are not of same length
```

Conditional Statements - Examples

- if/else if/else (results):

```
g.else.if <- function(x, y) {  
  if(length(x) == 1 || length(y) == 1) {  
    x + y  
  } else if(length(x) != length(y)) {  
    stop("ERROR: vector inputs are not of same length")  
  } else {  
    x + y  
  }  
}
```

```
g.else.if(x.vec, y.vec)      # OK!  
g.else.if(3.2, y.vec)        # OK!  
g.else.if(x.vec, y.bogus)    # Get error, by design
```

```
> g.else.if(x.vec, y.vec)      # OK!  
[1] 12 14 16 18 20 22 24 26 28 30  
> g.else.if(3.2, y.vec)        # OK!  
[1] 14.2 15.2 16.2 17.2 18.2 19.2 20.2 21.2 22.2 23.2  
> g.else.if(x.vec, y.bogus)    # Get error, by design  
Error in g.else.if(x.vec, y.bogus) :  
  ERROR: vector inputs are not of same length
```


Iterative Statement – For Loops

- These statements are executed for a defined number of times and are controlled by a counter or index and incremented at each cycle
- The syntax is

```
for(some indexed condition) {  
    do stuff  
}
```

or

```
for(some indexed condition)  
{  
    do stuff  
}
```

Iterative Statement – For Loops

- Some examples, using the vectors **x.vec** and **y.vec** from the previous conditional examples
- The **for** loops will depend on the number of elements in at least one of the vectors
- Simple example:

```
summ = 0.0
for(i in 1:length(x.vec))
{
  summ = summ + x.vec[i]
}
```

summ

- Result:

```
> summ
[1] 110
```

Iterative Statement – For Loops

- Calculate inner (dot) product of two vectors, as a function:

```
innerProd <- function(x, y) {  
  if(length(x) != length(y)) {  
    stop("ERROR: vector inputs are not of same length")  
  } else {  
    dotprod = 0.0  
    for(i in 1:length(x)) {  
      dotprod = dotprod + (x[i]*y[i])  
    }  
    return(dotprod)  
  }  
}
```

- Run function and check with R's `%*%` operator for inner product and matrix multiplication to check (this is preferred):

```
(innerProd(x.vec, y.vec))  
x.vec %*% y.vec
```

- Results:

```
> (innerProd(x.vec, y.vec))  
[1] 935  
> x.vec %*% y.vec  
      [,1]  
[1,] 935
```

Iterative Statement – For Loops

- Calculate inner (dot) product of two vectors again, but put in a limit and break out of loop if limit exceeded:

```
innerProdLim <- function(x, y, limit) {  
  if(length(x) != length(y)) {  
    stop("ERROR: vector inputs are not of same length")  
  } else {  
    summ = 0.0  
    for(i in 1:length(x)) {  
      summ = summ + (x[i]*y[i])  
      if(summ > limit) {  
        break      # terminates loop and advances to  
                  # first command outside of loop.  
      }  
    }  
    return(summ)  
  }  
}
```

- Call function: `(innerProdLim(x.vec, y.vec, 400.0))`

- Results:

```
> (innerProdLim(x.vec, y.vec, 400.0))  
[1] 420
```

Iterative Statement – While Loops

- Calculate inner (dot) product of two vectors again, but use a **while(.)** condition instead

```
whileInnerProd <- function(x, y) {  
  if(length(x) != length(y)) {  
    stop("ERROR: vector inputs are not of same length")  
  } else {  
    dotprod = 0.0  
    i = 1L # L for "long integer"  
    while(i <= length(x)) {  
      dotprod = dotprod + (x[i]*y[i])  
      i = i + 1  
    }  
    return(dotprod)  
  }  
}
```

- Call function: **whileInnerProd(x.vec, y.vec)**
- Result (matches previous result):

```
> (whileInnerProd(x.vec, y.vec))  
[1] 935
```

The **apply(.)** and **sapply(.)** functions

- These are two of the most powerful functions in R
 - Vectorized operations that are far more efficient than interpreted (not compiled) loops
 - ALWAYS prefer a ***apply(.)** function over loops when possible
 - Function to be applied may be an R function or a user-defined function (UDF)
- **apply(.)** will operate on rows or columns in a **matrix** or **dataframe** object, and returns a vector type with the results
 - **MARGIN** argument indicates whether over rows (1) or columns (2)
- **sapply(.)** will operate across columns on a **dataframe**, but not **matrix** objects; returns a vector type
- Other ***apply(.)** functions exist in R, depending on the return type desired
 - **lapply(.)**, for example, returns a list type rather than a vector
 - We will be more concerned, at least for now, with **apply(.)** and **sapply(.)**

The `apply(.)` and `sapply(.)` functions

- Some examples: Supposed we have a matrix (**appMtx**) and dataframe (**appDf**) that hold the same data:

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

- First, look at the matrix case:

```
(mtxColMeans <- apply(X = appMtx, MARGIN = 2, FUN = mean))  
(mtxRowVols <- apply(appMtx, 1, sd))  
(dfColMeans <- apply(appDf, 2, mean))  
(dfRowVols <- apply(appDf, 1, sd))
```

- Results:

```
> (mtxColMeans <- apply(X = appMtx, MARGIN = 2, FUN = mean))  
[1] 1 2 3 4  
> (mtxRowVols <- apply(appMtx, 1, sd))  
[1] 1.290994 1.290994 1.290994 1.290994 1.290994  
> (dfColMeans <- apply(appDf, 2, mean))  
v1 v2 v3 v4  
 1  2  3  4  
> (dfRowVols <- apply(appDf, 1, sd))  
[1] 1.290994 1.290994 1.290994 1.290994 1.290994
```

The `apply(.)` and `sapply(.)` functions

- Check the return types (objects):

```
is(mtxColMeans)  
is(dfRowVols)
```

- Each is a vector of double (numeric) types:

```
> is(mtxColMeans)  # vector of doubles  
[1] "numeric"      "vector"          "numericORNULL"  
> is(dfRowVols)    # vector of doubles  
[1] "numeric"      "vector"          "numericORNULL"
```


The `apply(.)` and `sapply(.)` functions

- Next, let's look at `sapply(.)`
- Note that it is only applied to columns in the dataframe:

```
(dfColVols <- sapply(X = appDf, FUN = sd))  
(dfColMeans <- sapply(appDf, mean))
```

- Results:

```
> (dfColVols <- sapply(X = appDf, FUN = sd))  
v1 v2 v3 v4  
0  0  0  0  
> (dfColMeans <- sapply(appDf, mean))  
v1 v2 v3 v4  
1  2  3  4
```

- Each is a vector of double (numeric) types:

```
is(dfRowVols)
```

```
> is(dfRowVols)
```

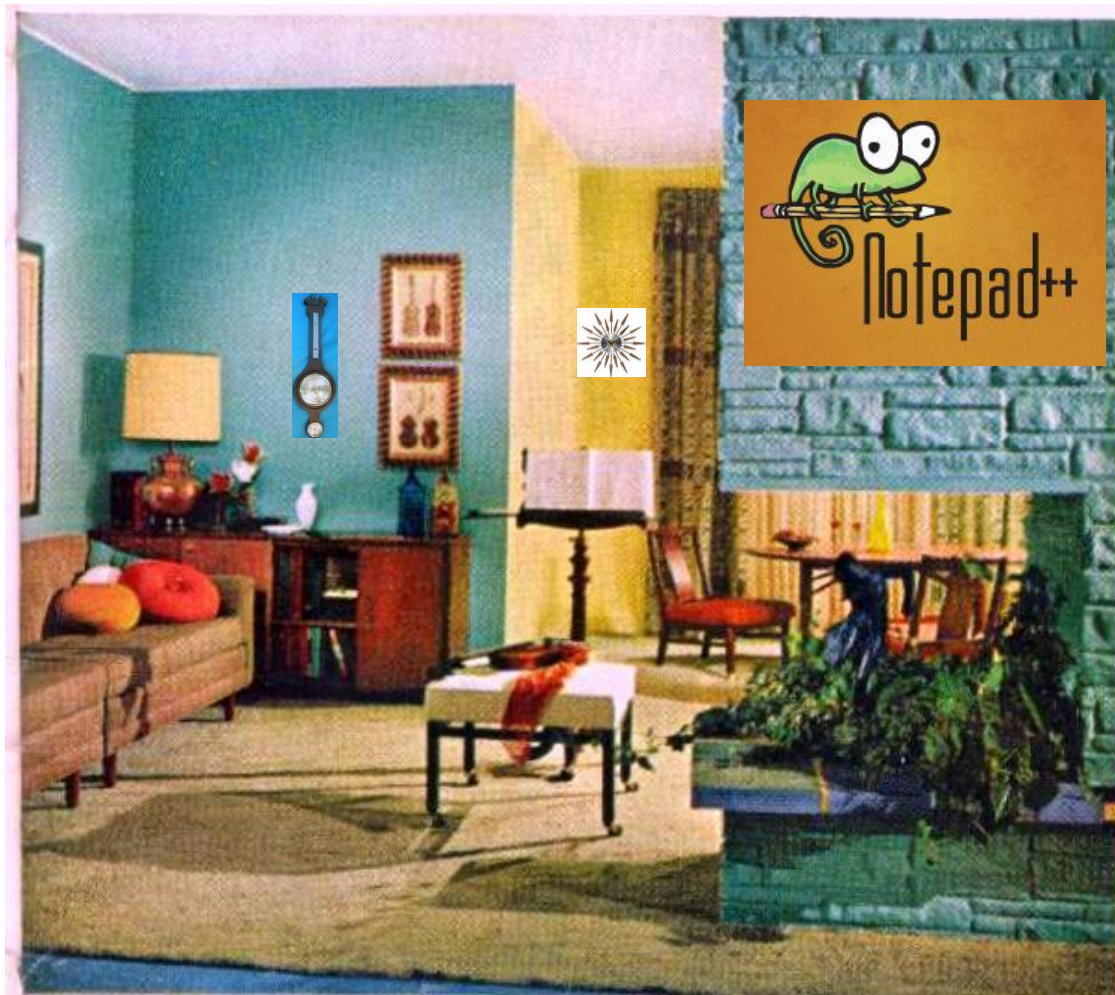
```
[1] "numeric"
```

```
"vector"
```

```
"numericORNULL"
```

And now, a word from our sponsor

- For working with data files in text/csv format (ie, ASCII), the free utility **Notepad++** is highly recommended
- Makes it easy to get a quick view of your data without the overhead of Excel or other software
- Also useful for viewing source code files in R (and other programming languages) without the overhead of RStudio or other IDE's
- Download here:
- <https://notepad-plus-plus.org/download>



Jupiter and Beyond



THE END