Introduction To Data Science Using R Programming

Dr. Shahid Mahmood Awan
Assistant Professor
School of Systems and Technology, University of Management and Technology
shahid.awan@umt.edu.pk

Umer Saeed(MS Data Science, BSc Telecommunication Engineering)
Sr. RF Optimization & Planning Engineer
f2017313017@umt.edu.pk

Introduction To Data Science Using R Programming

Background and Overview

Background and Overview

- **What is R:**
- R is a dialect of the S language
- **What is S?**
- S is a language that was developed by John Chambers and others at Bell Labs.
- S was initiated in 1976 as an internal statistical analysis environment-originally implemented as Fortran libraries.
- Early versions of the language did not contain functions for statistical modeling.
- In 1998 the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language). The book statistical Models in S by Chambers and Hastie (the white book) documents the statistical analysis functionality.
- Version 4 of the language was release in 1998 and is the version we use day. The book programming with Data by Johan Chambers (the green book) documents this version of the language.

Background and Overview

- **Historical Notes**
- In 1993 Bell Labs gave StatSci (now Insightful Corp.) an exclusive license to develop and sell the S language.
- In 2004 Insightful purchased the S language from Lucent for $2 million and is the current owner.
- In 2006, Alcatel purchased Lucent Technologies and is no called Alcatel-Lucent.
- Insightful sells its implementation of the S language under the product name S-PLUS and has built a number of fancy features (GUIs, mostly) on top of it-hence the “PLUS”
- In 2008 Insightful is acquired by TIBCO for $25 million.
- The fundamentals of the S language itself has not changed dramatically since 1998.
- In 1998, S won the Association for Computing Machinery’s Software System Award.

Background and Overview

- **S Philosophy**
- In " Stages in the Evolution of S", Johan Chambers writes:
- *"[W]e wanted users to be able to begin in a interactive environment, where they did consciously think of themselves as programming. Then as their needs became clearer and their sophistical increased, they should be able to slide gradually into programming, when the language and system aspects would become more important."*

Background and Overview

- **Back to R**
- 1991: Created in New Zealand by Ross lhaka and Robert Gentleman. Their experience developing R is documented in a 1996 JCGS paper.
- 1993: First announcement of R to the public.
- 1995: Marin Mächler convince Ross and Robert to use the GNU General Public License to make R free software.
- 1996: A public mailing list is created (R-help and R-devel)
- 1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the sources code for R.
- 2000: R version 1.0.0 is released.
- 2012: R version 2.15.1 is released on June 22,2012.

Background and Overview

- **Features of R**
- Syntax is very similar to S, making it easy for S-PLUS users to switch over.
- Semantics are superficially similar to S, but in reality are quit different (more on that later)
- Runs on almost any standard computing platform/OS (even on the PlayStation3)
- Frequent release (annual+bgfix releases); active development.
- Quite lean, as far as software goes; functionality is divided into modular packages.
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (usr→ programmer)

Background and Overview

- **What is R?**
- A software environment for data analysis, statistical computing and graphics.

- A programming language;
- 1- Natural to use
- 2- Complete data analyses in just a few lines.

- **Why Use R?**
- There are many choices for data analysis software;
  - SAS, Stata, SPSS, Excel, MATLAB, Minitab, Pandas.
  - So why are we using R?

Background and Overview

- ✓ Free (Open-Source Project) - (Both in the sense of beer and in the sense of speech.)

With free software, you are granted;

1- The freedom to run the program, for any purpose (freedom 0)

2- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.

3- The freedom to redistribute copies so you can help your neighbor (freedom 2).

4- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

Background and Overview

- ✓ Widely used;
- More than 2 million very active and vibrant users community around the world
- New features are being developed all the time
- A lot of community resources; R-help and R-devel mailing lists and Stack Overflow.
- Easy to re-run previous work and make adjustments
- Nice graphics and visualizations

Background and Overview

- **Drawbacks of R**
- Essentially based on 40 year old technology.
- Little built in support for dynamic or 3-D graphics (but things have improved greatly since the “old days”).
- Functionality is based on consumer demand and user contributions. If no one feels like implementing your favorite method, then it’s your job!
- (Or you need to pay someone to do it)
- Objects must generally be stored in physical memory; but there have been advancements to deal with this too
- Not ideal for all possible situations (but this is a drawback of all software packages).

Background and Overview

- **Design of the R System**
- The R system is divided into 2 conceptual parts:
- 1- The “base” R system that you download from CRAN

- 2- Everything else.

- R functionality is divided into a number of packages.
- 1- The “base” R system contains, among other things, the base package which is required to run R and contains the most fundamental functions.

- 2-The other packages contained in the “base” system include utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4

- 3- There are also “Recommend” packages: boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix

Background and Overview

- And there are many other packages available:
- 1- There are about 4000 packages on CRAN that have been developed by users and programmers around the world.
- 2- There are also many packages associated with the Bioconductor project (http://bioconductor.org)
- 3- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion

R Resources

- We will just use the R command line interface
- If you want to try a graphical user interface (GUI), here are some popular choices:
  - RStudio (http://www.rstudio.com)
  - Rattle (http://rattle.togaware.com)
  - Official page: (http://www.r-project.org)
  - Download page:(http://www.cran.r-project.org)

  - Some helpful websites:
  - http://www.statmethods.net
  - www.rseek.org
  - http://www.ats.ucla.edu/stat/r/
  - http://finzi.psych.upenn.edu/search.html
- Best way to learn R is through trial and error

Introduction To Data Science Using R Programming

Installing R and R Studio on Windows

Installing R and R Studio on Windows

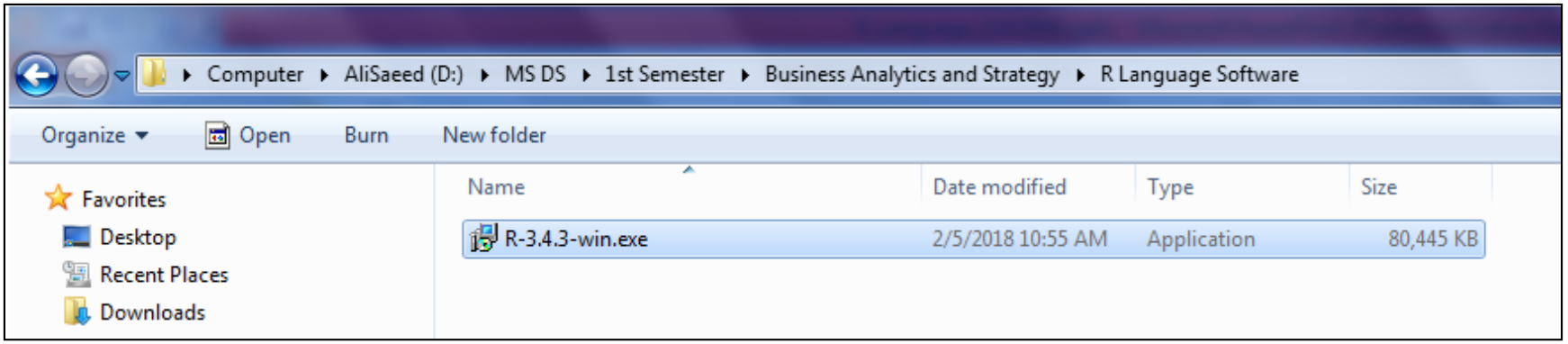
- Windows doesn't come with R installed.
- You can get a basic copy of R from the Comprehensive R Archive Network (CRAN) site at https://cran.r-project.org/.
- The site provides both source code versions and compiled versions of the R distribution for various platforms.
- Unless you plan to make your own changes to the basic R support or want to delve into how R works, getting the compiled version is always better.
- Before you can install RStudio, you must install a copy of R on your system.
- RStudio comes with a graphical installation application for Windows, so getting a good install means using a wizard, much as you would for any other installation.
- Desktop version of RStudio (https://www.rstudio.com/ products/rstudio/#Desktop) to make the task of working with R even easier.

Installing R on Windows

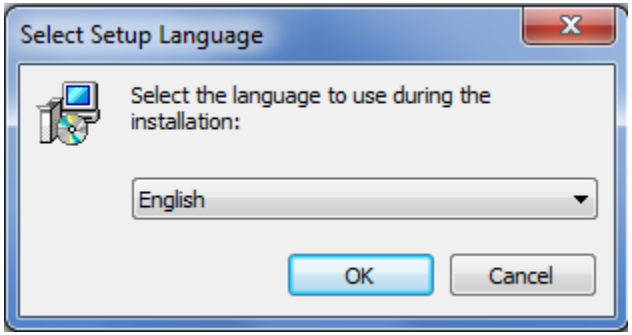
- **Locate the downloaded copy of R on your system**
- The name of this file varies, but it normally appears as R-3.4.3-win.exe.
- The version number appears as part of the filename.
- In this case, the filename tells you that you've downloaded version 3.4.3, which is the version of R used for Umer Saeed R Language notes.
- Using a different version of R could cause problems with the downloadable source code.

Installing R on Windows

- Double-click the installation file

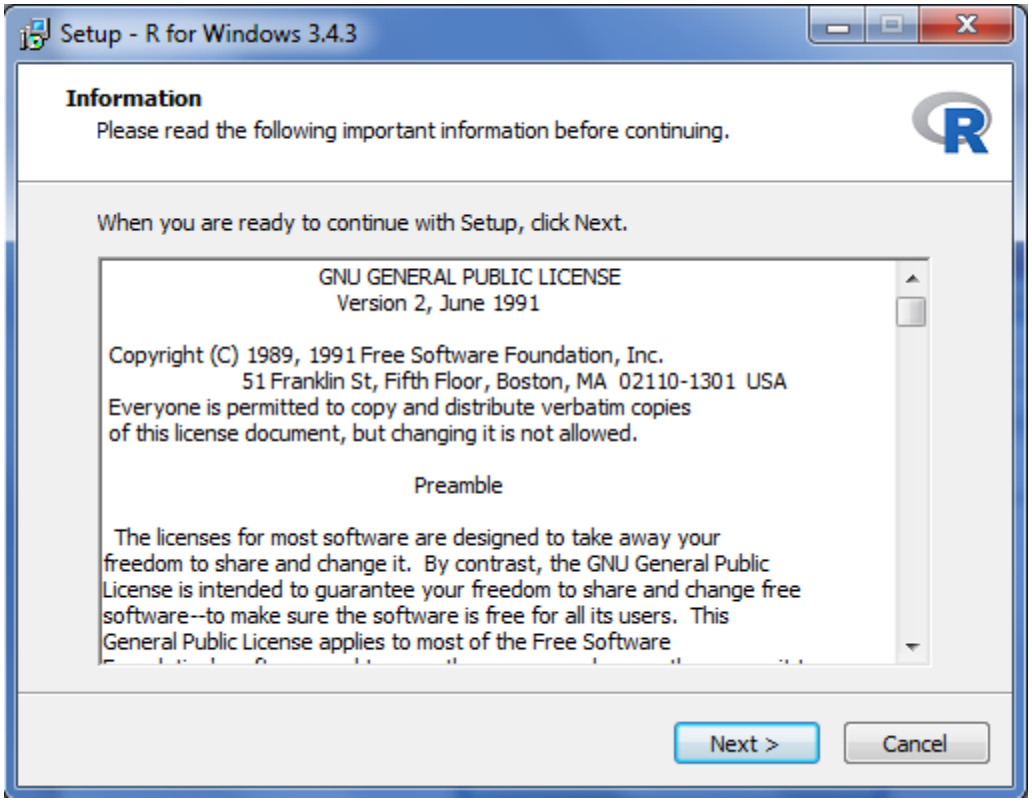


- You see the Select Setup Language dialog box



Choose a language from the list and then click OK

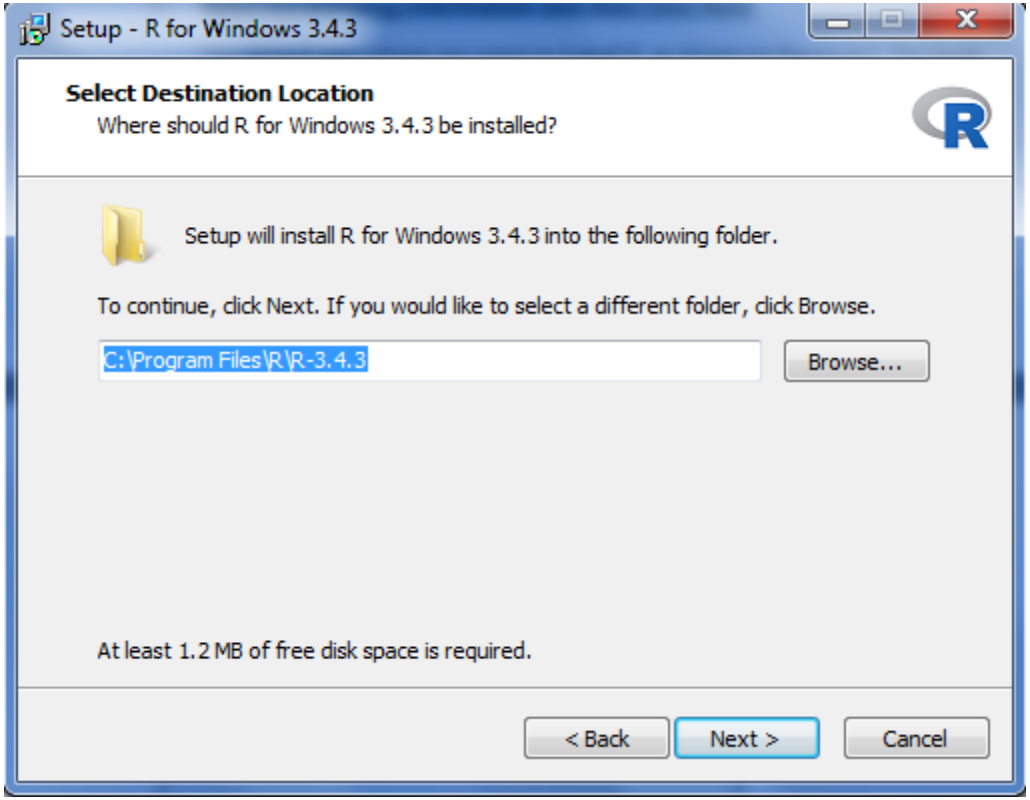
Installing R on Windows



Read the licensing information and then click Next.

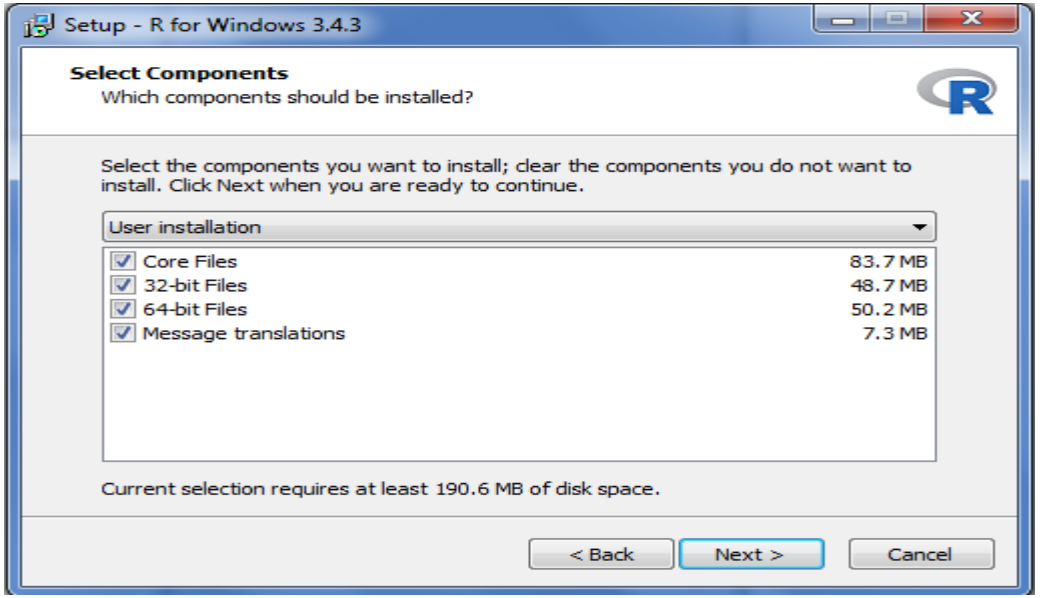
You see the Setup dialog box, shown in Figure . The content of this dialog box differs with the version of R you install. However, the main consideration is that you install version 3.4.3, as shown in the figure.

Installing R on Windows



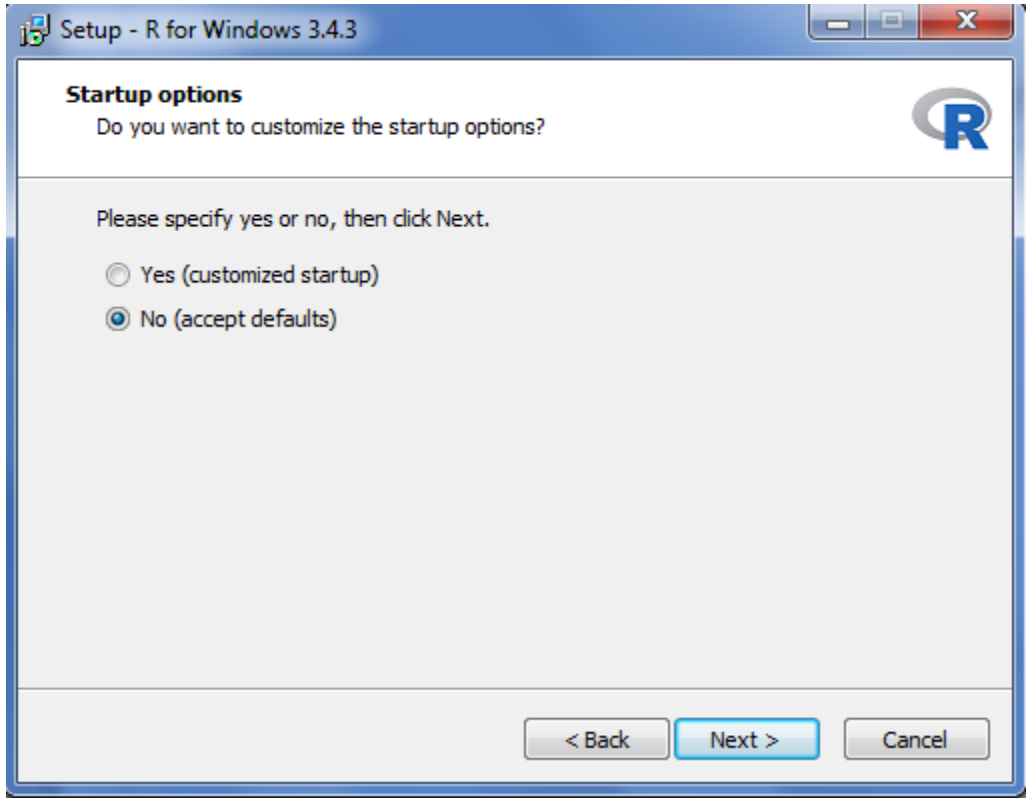
The wizard asks where you want to install R, as shown in Figure. The Notes assumes that you use the default installation location.

Installing R on Windows



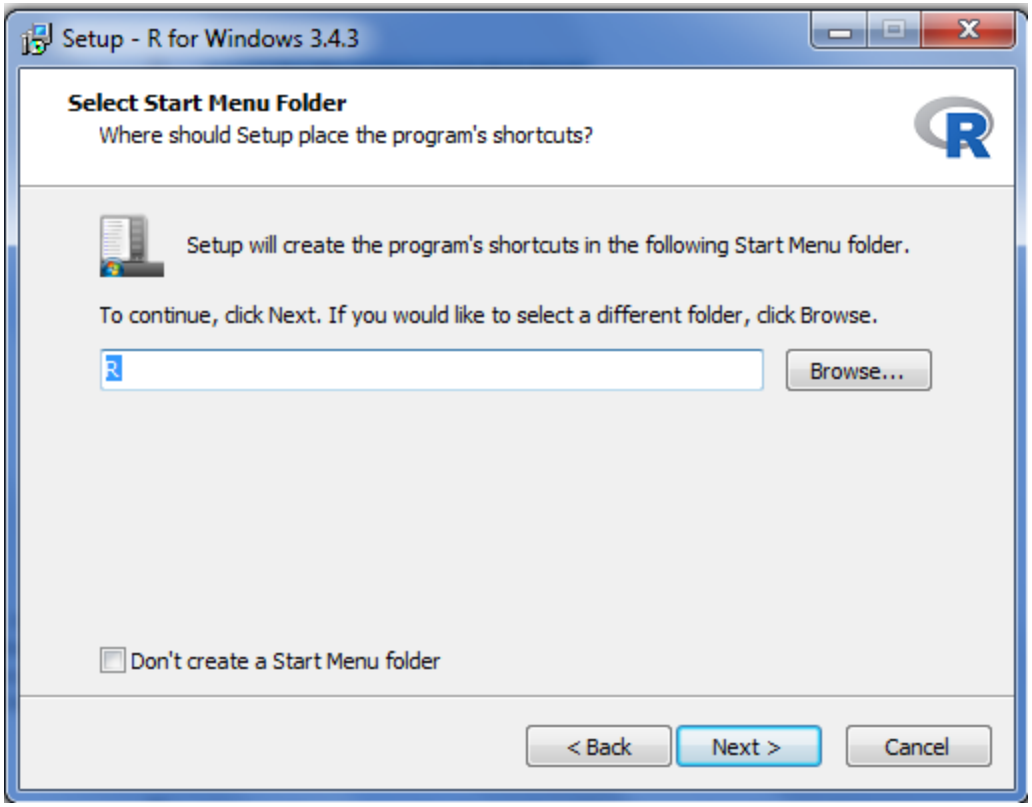
- The wizard asks which components you want to install, as shown in Figure. Given that the components don't consume a lot of space, installing everything is best to ensure that you have a complete installation.
- Modify the list of selected components (if necessary) by selecting or deselecting the check boxes; then click Next

Installing R on Windows



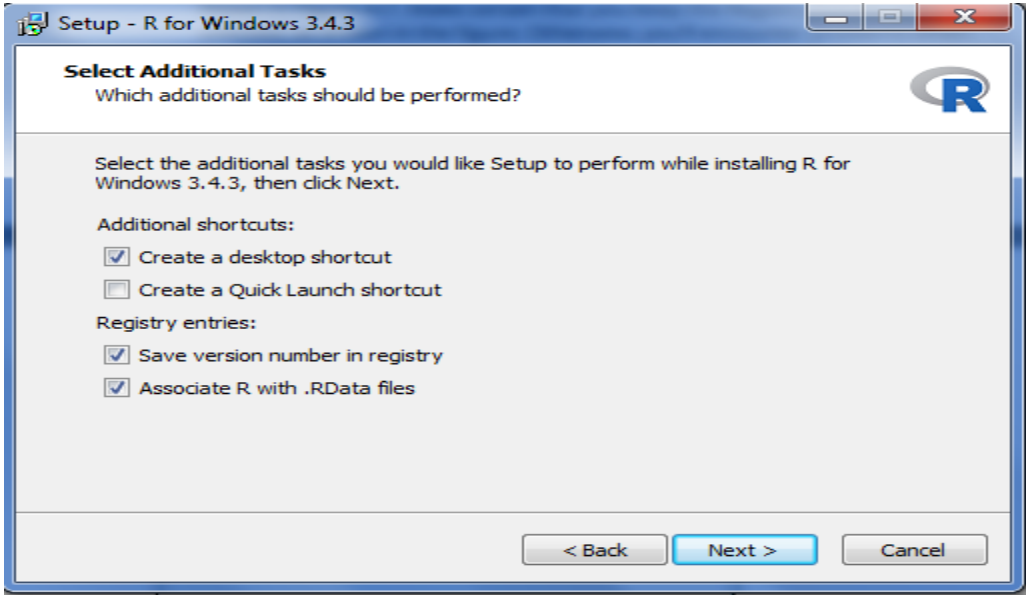
- The wizard asks whether you want to customize the startup options, as shown in Figure . Modifying the startup options requires advanced knowledge of how R works. The default options will work fine for this Notes.
- Select the No option and click Next.

Installing R on Windows



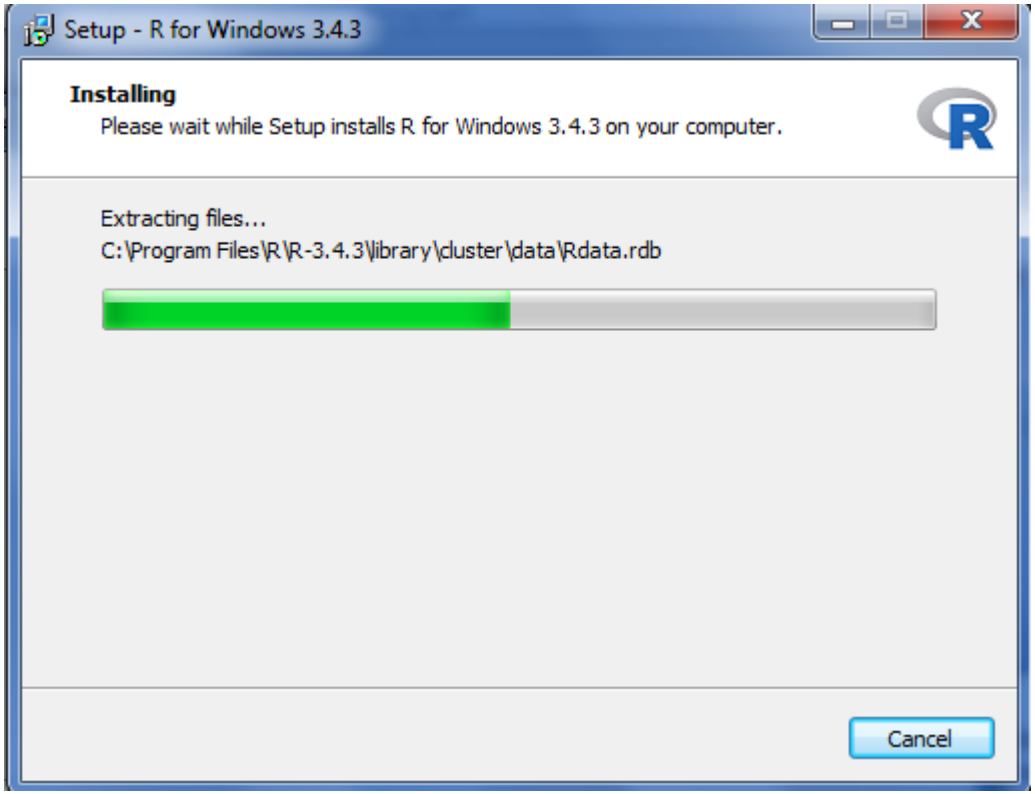
- The wizard asks where you want to place the R icons (the “program shortcuts”) on the Start menu, as shown in Figure. This notes assumes that you use the default Start menu setup.
- Choose Start Menu Configuration (if necessary) and then click Next

Installing R on Windows



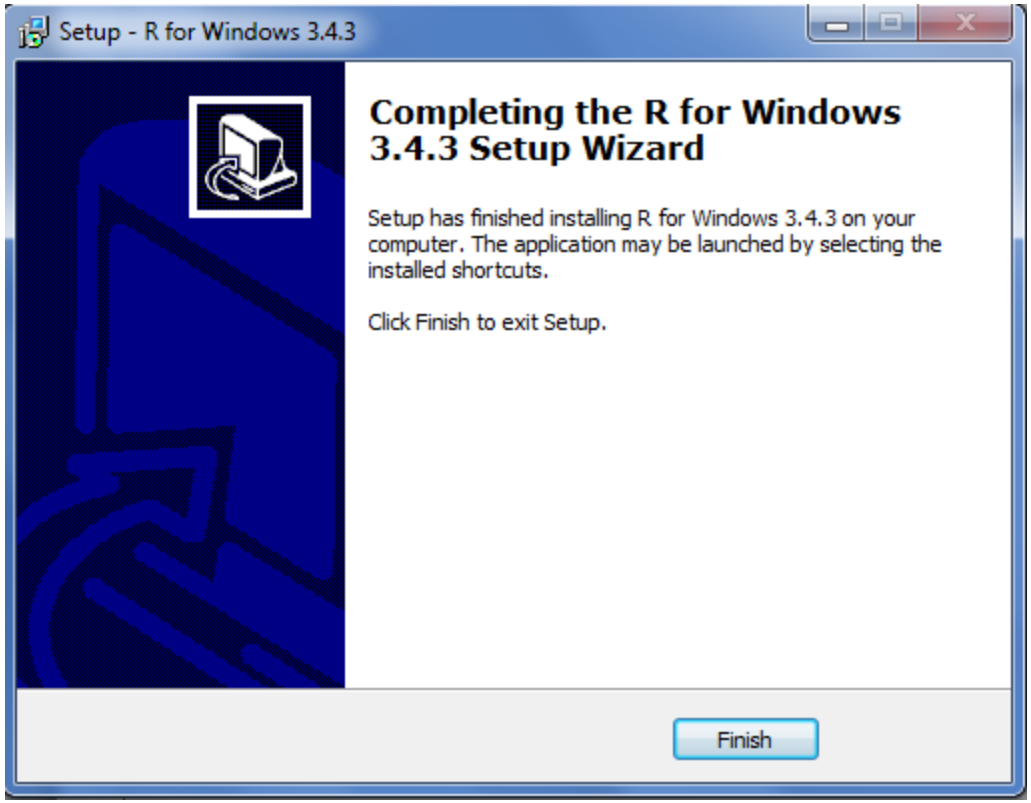
- The wizard asks you to configure any additional script configuration options, as shown in Figure . Make certain that you keep the Registry Entries options selected, as shown in the figure. Otherwise, you'll encounter problems when configuring RStudio for use.
- Modify the selected additional tasks as needed and click Next

Installing R on Windows



- You see an Installing dialog box that tells you about the installation progress.

Installing R on Windows



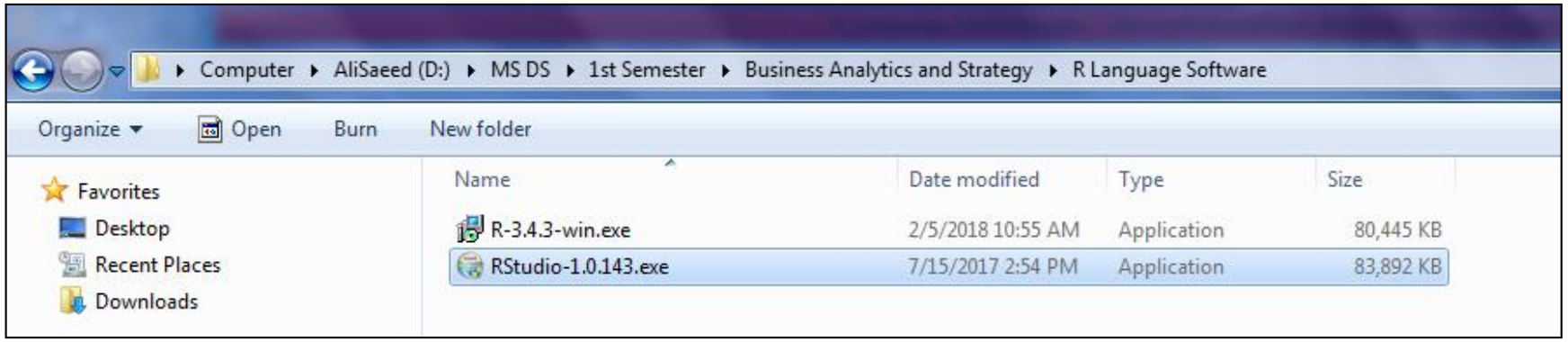
- After the process completes, you see a completion dialog box appear in its place
- Click Finish
- You're now ready to install RStudio so that you can use R with greater ease

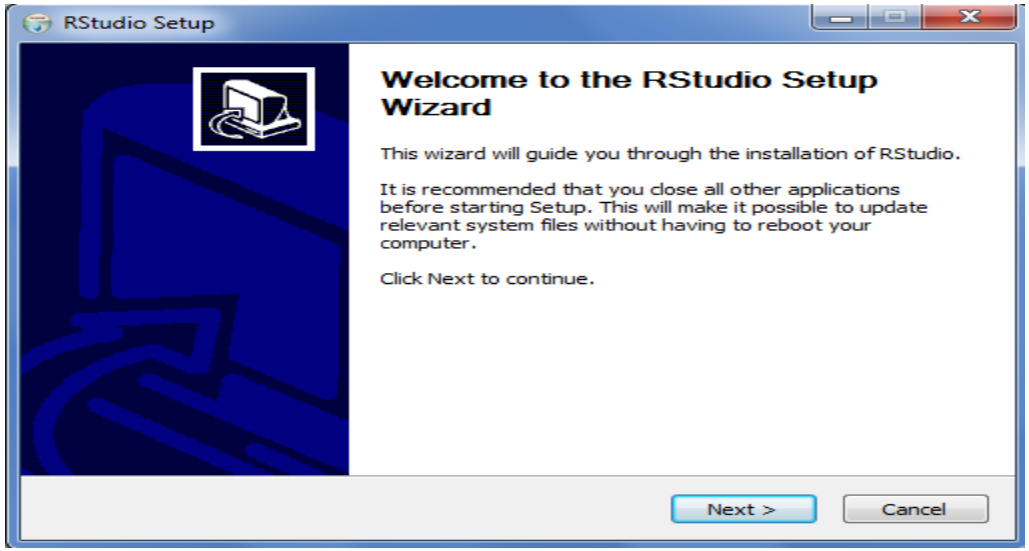
Installing R Studio on Windows

- **Locate the downloaded copy of RStudio on your system**
- The name of this file varies, but normally it appears as RStudio-1.0.143.exe.
- The version number is embedded as part of the filename.
- In this case, the filename refers to version 1.0.143, which is the version used for this notes.
- If you use some other version, you may experience problems with the source code and need to make adjustments when working with it.

Installing R Studio on Windows

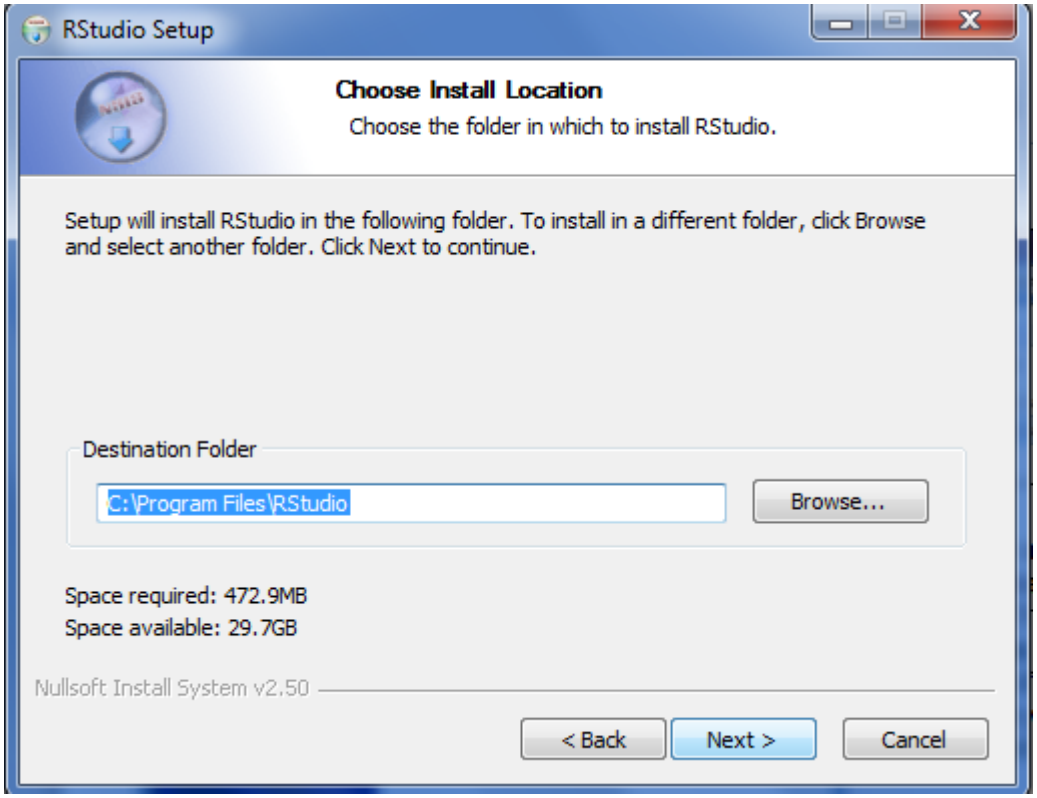
- Double-click the installation file





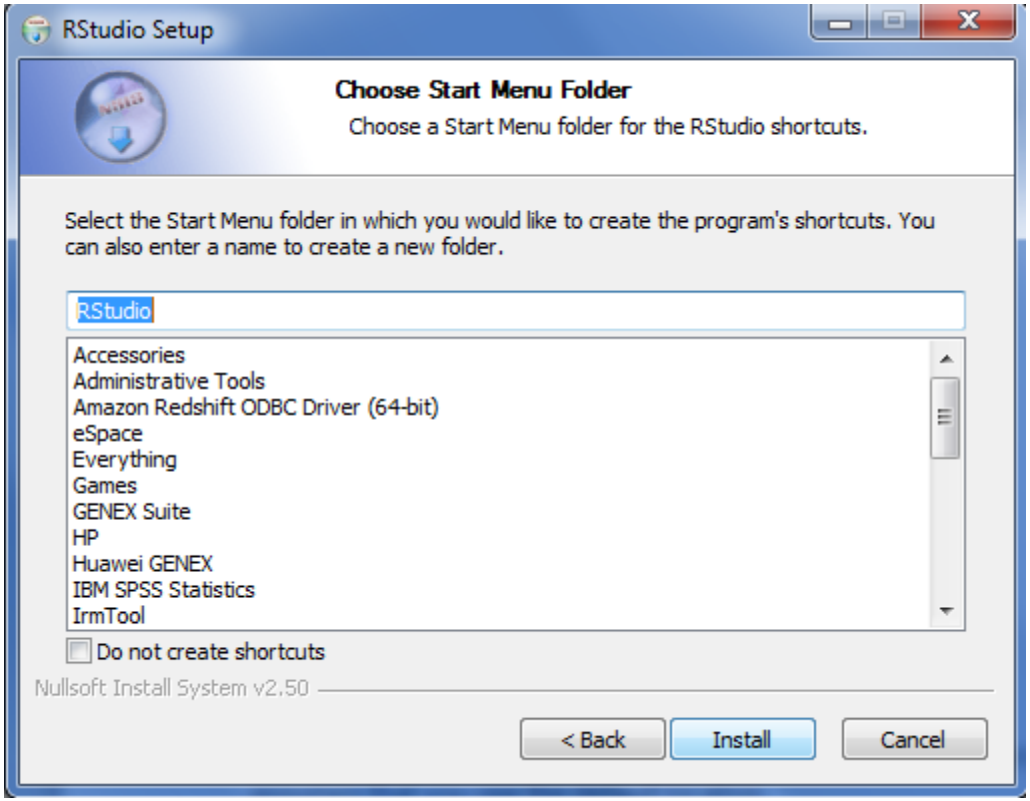
- You see an R Studio Setup dialog box similar to the one shown in Figure. The exact dialog box that you see depends on which version of the RStudio installation program you download.
- Click Next

Installing R Studio on Windows



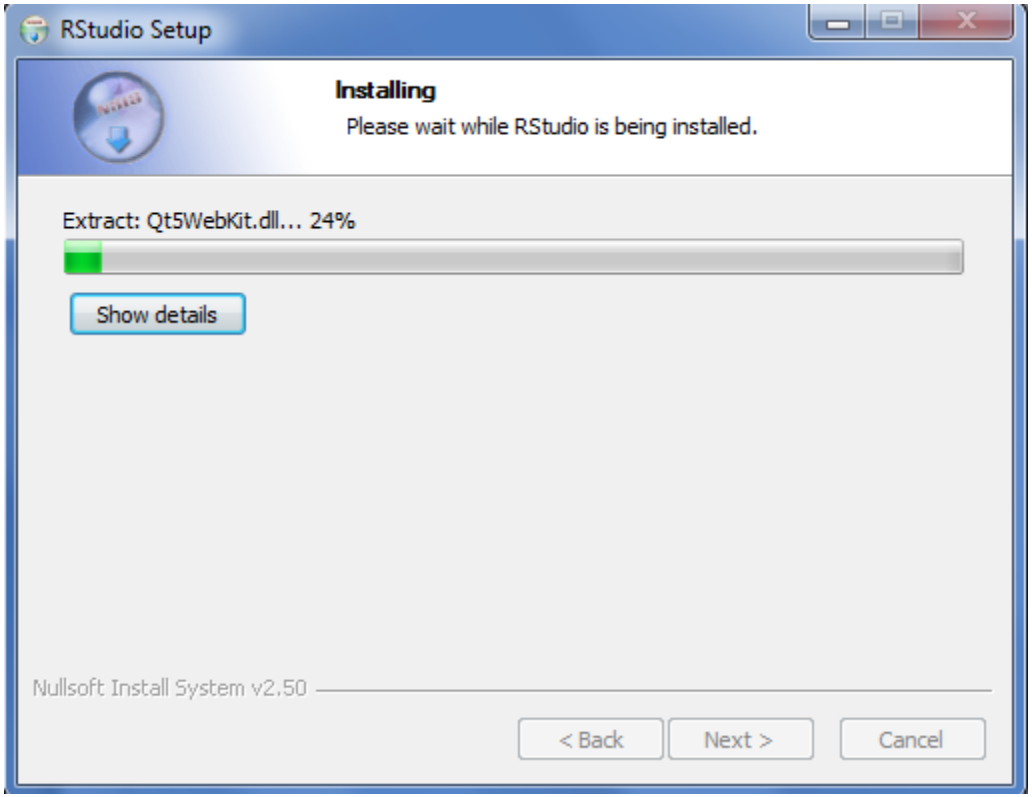
- The wizard asks where to install RStudio on disk, as shown in Figure. The notes assumes that you use the default location.
- Choose an installation location (if necessary) and then click Next

Installing R Studio on Windows



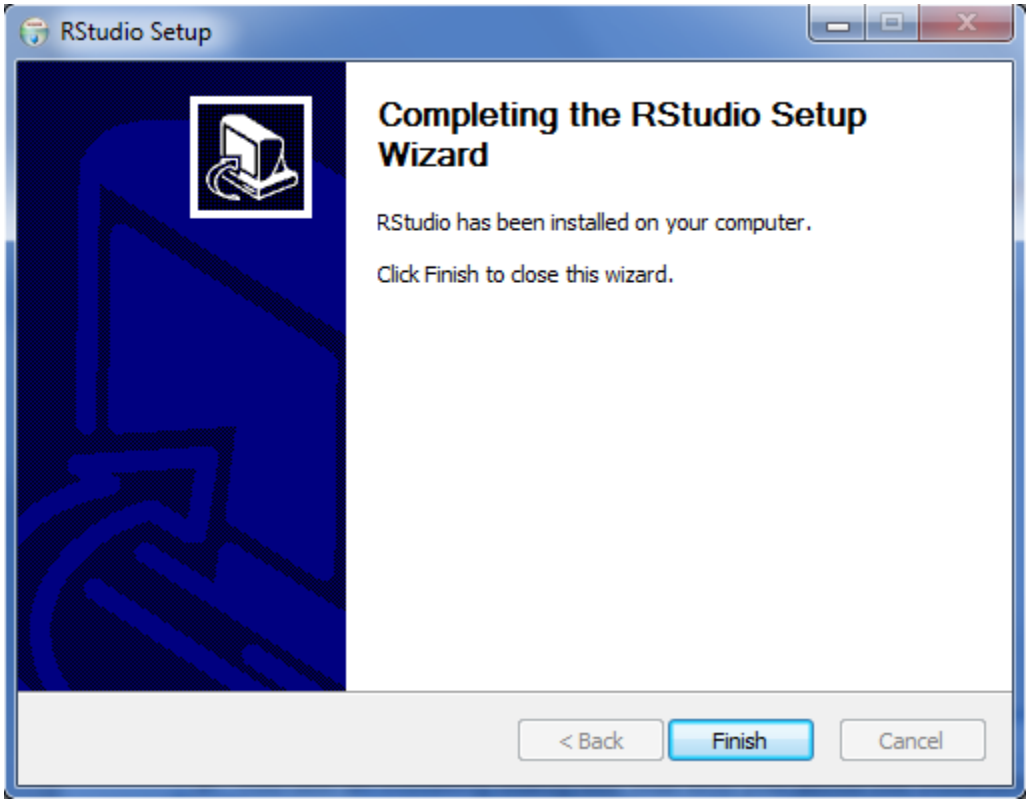
- The wizard asks where you want to place the RStudio icons (shortcuts) in the Start menu, as shown in Figure . This notes assumes that you use the default Start menu setup.
- Choose Start Menu Configuration (if necessary) and then click Install

Installing R Studio on Windows



- You see an Installing dialog box that tells you about the installation progress.

Installing R Studio on Windows



- After the process completes, you see a completion dialog box appear in its place.
- Click Finish
- You're ready to begin using RStudio

Installing R & R Studio on Windows

- We can check R Software version using command > R.version.

- **Run R online**
- If you can't install R you can get at least some of the functionality from different providers of online R.
- They may have limitations, and you may have to learn something specific to each website, but it can be a useful alterative for you at some time during the course or after.

1- http://www.tutorialspoint.com/execute_r_online.php
2- http://www.r-fiddle.org
3- http://pbil.univ-lyon1.fr/Rweb/Rweb.general.html

Introduction To Data Science Using R Programming

Comments in R Language

Comments

- Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program.
- Single comment is written using # in the beginning of the statement as follows –
- > # My first program in R Programming

- R does not support multi-line comments but you can perform a trick which is something as follows –

- > #if(FALSE) { "This is a demo for multi-line comments and it should be put inside either a single OR double quote“ }

- > #if(FALSE) { 'This is a demo for multi-line comments and it should be put inside either a single OR double quote' }

- Though above comments will be executed by R interpreter, they will not interfere with your actual program. You should put such comments inside, either single or double quote.

Introduction To Data Science Using R Programming

Pre-Define Functions

Pre-Define Functions

- # Note that the "**:**" operator is used to create integer sequences

Statement	Output
> 5+5	10
> 6*7	42
> 6/2	3
> 6-4	2
> 2^3	8
> 2**3	8
> letters	
> LETTERS	
> head(1:100)	[1] 1 2 3 4 5 6
> head(1:100,3)	[1] 1 2 3
> tail(1:10)	[1] 5 6 7 8 9 10
> tail(1:10,3)	[1] 8 9 10
> length(1:10)	[1] 10
> sqrt(1:2)	[1] 1.00 1.41
> factorial(1:3)	[1] 1 2 6
> range(1:5000)	[1] 1 5000
> mean(1:5)	[1] 3
> median(1:5)	[1] 3

Statement	Output
> sd(1:5)	[1] 1.581139
> var(1:5)	[1] 2.5
> Inf	[1] Inf
> log(1:2)	[1] 0.0 0.69
> exp(1:2)	[1] 2.71 7.38
> args(log)	
> log(8,base=2)	[1] 3
> log(8,2)	[1] 3
> pi	[1] 3.141593
> abs(-65)	65
>round(3.475,2)	[1] 3.48
> ceiling(3.475)	[1] 4
> floor(3.475)	[1] 3
> trunc(5.99)	[1] 5
>signif(3.475, digits=2)	[1] 3.5
> cos(0)	[1] 1
> sin(0)	[1] 0
> tan(0)	[1] 0

Pre-Define Functions

Statement	Output
> quantile(0:400)	0% 25% 50% 75% 100% 0 100 200 300 400

- #**ceiling** takes a single numeric argument x and returns a numeric vector containing the smallest integers not less than the corresponding elements of x.
- # **floor** takes a single numeric argument x and returns a numeric vector containing the largest integers not greater than the corresponding elements of x.
- #**trunc** takes a single numeric argument x and returns a numeric vector containing the integers formed by truncating the values in x toward 0.
- #**round** rounds the values in its first argument to the specified number of decimal places (default 0).
- # **signif** rounds the values in its first argument to the specified number of significant digits.

Introduction To Data Science Using R Programming

Getting Help & History

Getting Help & History

- Once R is installed, there is a comprehensive built-in help system. At the program's command prompt you can use any of the following:
- > help.start() # General help
- > help.search("log") # help about function log
- > help(log) # help about function log
- >?log # help about function log
- >?"+" # help about function Arithmetic Operators
- > example(sum) # show an example of function sum
- > apropos("sum") # list all functions containing string sum

- > history() # history provided the command list

Introduction To Data Science Using R Programming

R – Variables & Entering Input

R – Variables (Assignment)

- # Called Left Assignment
- > v1 <- “Umer”
- > v2 <<-”Ali”
- > v3 = “Ahmed”
- # Called Right Assignment
- > “Bilal”-> v4
- > “Mohsin” ->> v5

Entering Input

Statement	Output
> x<-1	# nothing printed
> print(x)	[1] 1 #explicit printing
> x	[1] 1 #auto-printed
> msg<-"hello"	# nothing printed
> print(msg)	[1] hello #explicit printing
> msg	[1] hello #auto-printed
> msg1<-'hello'	# nothing printed
> print(msg1)	[1] hello #explicit printing
> msg1	[1] hello #auto-printed

- The [1] shown in the output indicated that x is a vector and 1 is its first element.
- The grammar of the language determines whether an expression is complete or not
- > x<- # Incomplete expression
- > options(width=40)
- > print(x<-10:30)
- The numbers in the square brackets are not part of the vector itself, they are merely part of the printed output.

Entering Input

- # to calculate 15% tip on a meal that cost $ 19.71.
- > 0.15*19.71

- # x^2+x-1=0; calculate the values of x using Quadratic Formula
- >a<-1
- >b<-1
- >c<- -1
- > sol1<-((-b+sqrt(b^2-4*a*c))/(2*a))
- > sol2<-((-b-sqrt(b^2-4*a*c))/(2*a))

- # Nasted Function (Inside→ out calculation)
- > log(exp(x<-1:5))

R - Variables

Variable Name	Validity	Reason	Statement	Output
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.	`> print(var_name%<-"hello")`	Error: unexpected input in "print(var_name%<-"hello")"
2var_name	invalid	Starts with a number is not allowed.	`> print(2var_name<-"hello")`	Error: unexpected symbol in "print(2var_name"
.2var_name	invalid	The starting dot is followed by a number making it invalid.	`>print(.2var_name<-"hello")`	Error: unexpected symbol in "print(.2var_name"
_var_name	invalid	Starts with _ which is not valid	`> print(_var_name<-"hello")`	Error: unexpected symbol in "print(var name"
var name	invalid	Space is not allowed in variable name	`> print(var name<-"hello")`	Error: unexpected symbol in "print(var name"
var_name2.	valid	Has letters, numbers, dot and underscore	`> print(var_name2.<-"hello")`	hello
.var_name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.	`> print(.var_name<-"hello")`	hello
var.name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.	`> print(var.name<-"hello")`	hello

R - Variables

- **Finding Variables**
- To know all the variables currently available in the workspace we use the ls()function. Also the ls() function can use patterns to match the variable names.

Statement	Output
> print(ls())	[1] "a" "t" "v"
> ls()	[1] "a" "t" "v"
*1> print(ls(pattern="var"))	[1] "var.name""var_name2."
*2> print(ls(all.name = TRUE))	[1] ".Random.seed" ".um"
> print(ls(all.name = FALSE))	[1] "a" "t" "v"
objects()	[1] "a" "t" "v"

- The ls() function can use patterns to match the variable names.(*1)
- The variables starting with dot(.) are hidden, they can be listed using "all.names = TRUE" argument to ls() function. (*2)

R - Variables

- **Deleting Variables**
- Variables can be deleted by using the rm() function.

Statement	Output
> rm(x)	
> x	Error: object 'x' not found
>rm(list=ls())	
print(ls())	

Difference between ls() and ls

```
> ls()
> ls
```

To Clear Console

```
Console C:/Users/uwx161178/Downloads/specdata/
> a<-1:10
> b<-11:20
> c(a,b)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> cat(a,b)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
> print(charToRaw('hello'))
[1] 68 65 6c 6c 6f
> print(charToRaw('umer'))
[1] 75 6d 65 72
> print(numericToRaw(1:6))
Error in numericToRaw(1:6) : could not find function "numericToRaw"
> objects()
[1] "a" "b"
>
```

CTRL+L

```
Console C:/Users/uwx161178/Downloads/specdata/
>
```

Introduction To Data Science Using R Programming

Attributes & R Objects

Attributes

- R objects can have attributes;
- Class
- Names, dimensions names
- Dimensions (e.g matrices, arrays)
- Length
- Other user-defined attributes/metadata
- Attributes of an objects can be accessed using the attributes() function.
- Not all R Objects contain attributes, in which case the attributes () function returns NULL.

R Objects

- R has six basic or “atomic” classes of objects”.
- 1- Character
- 2- Numeric (real number)
- 3- Integer
- 4- Complex
- 5- logical (True/False)
- 6- Raw (There is also class for “raw” objects, but they are not commonly used directly in data analysis

Numbers

- Numbers in “R” generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like “1” or “2” in “R”, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like “1.00” or “2.00”. This isn’t important most of the time… except when it is.

Statement	Output
>class(a<-2)	[1] "numeric“
> class(a<-2.03)	[1] "numeric“
*1> class(a<-1L)	[1] "integer"

- If you explicitly want an integer, you need to specify the L suffix. So entering 1 in R gives you a numeric object; entering 1L explicitly gives you an integer object(*1)

Numbers

- **SPECIAL NUMBER**

Statement	Output
*1>1/0	[1] Inf
*2> 1/Inf	[1] 0
*3> 0/0	[1] NaN

- **Infinity:** There is also a special number “Inf” which represents infinity. This allow us to represent entities like 1/0. (*1)
- This way, “Inf” can be used in ordinary calculations; e.g. 1/Inf is 0.(*2)
- **Not a number:** The value “NaN” represents an undefined values (“not a number”) e.g. 0/0; NaN can also be thought of as a missing value (more on that later) (*3)

Introduction To Data Science Using R Programming

Vectors

Vectors

- The most basic type of R object is a vector.
- A vector can only contain objects of the same class Character, numeric, logical
- **BUT:** The one exception is a ***list***, which is represented as a vector but can contain objects of different classes (indeed, that's usually why we use them)
- **Creating vectors**
- Empty vector can be created with the vector () function.

Statement	Output
> vector()	logical(0)
> vector("numeric")	numeric(0)
> vector("logical")	logical(0)
>vector("character")	character(0)
> vector("complex")	complex(0)

Statement	Output
> vector("numeric",length = 3)	[1] 0 0 0
> vector("logical",length = 3)	[1] FALSE FALSE FALSE
> vector("character",length = 3)	[1] "" "" ""
>vector("complex",length = 3)	[1] 0+0i 0+0i 0+0i

Vectors

Statement	Output
> class(print(a<-1:5))	[1] 1 2 3 4 5 [1] "integer"
> is.vector(a)	[1] TRUE

The c() function combines multiple items into a continuous print output.

Statement	Output
> print(a<-c(2,4,6))	[1] 2 4 6
> print(b<-c(8,10,12))	[1] 8 10 12
> print(c<-c(a,b))	[1] 2 4 6 8 10 12

Vectors

- **Class of the Vectors:**

Statement	Output
> class (x<-c(0.5,0.6))	[1] "numeric"
> class(x<-c(TRUE,FALSE))	[1] "logical"
> class(x<-c(T,F))	[1] "logical"
> class(x<-c("a","b","c"))	[1] "character"
> class(x<-9:29)	[1] "integer"
> class(x<-c(1+0i,2+4i))	[1] "complex“
> class(x<-c("1","2"))	[1] "character"

- “T” & “F” are short-hand ways to specify TRUE and FALSE.
- However in general one should try to use explicit “TRUE” and “FALSE” values.
- The T and F are primarily there for when you’re feeling lazy.

Vectors

- **Mixing Objects**
- There are occasions when different classed of “R” objects get mixed together. Sometimes this happen by accident but it can also happen on purpose.

Statement	Output
> class(y<-c(1.7,"a"))	[1] "character“
> class(y<-c(TRUE,2))	[1] “numeric"
> class(y<-c("a",TRUE))	[1] "character“

- In each case above, we are mixing objects of two different classed in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

Vectors

- **Explicit Coercion**
- Objects can be explicitly coerced from one class to another using the as .* functions, if available.

Statement	Output
> class(x<-0:6)	[1] "integer"
> as.numeric(x)	[1] 0 1 2 3 4 5 6
> as.logical(x)	[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)	[1] "0" "1" "2" "3" "4" "5" "6"

Vector

- Nonsensical coercion results in NAs.
- Sometimes, R can't figure out how to coerce an object and this can result in NAs being produced.

Statement	Output
> class(x<-c("a","b","c"))	[1] "character"
> as.numeric(x)	[1] NA NA NA Warning message: NAs introduced by coercion
> as.logical(x)	[1] NA NA NA
> as.complex(x)	[1] NA NA NA Warning message:NAs introduced by coercion

- When nonsensical coercion takes plan, you will usually get a warning from R.

Types Of Operators

- We have the following types of operators in R programming −
- 1- Arithmetic Operators
- 2- Relational Operators
- 3- Logical Operators
- 4- Miscellaneous Operators

Vectors

- **1- Vectorized Operations (ARITHMETIC OPERATORS)**
- > x<-1:4 ; y<-6:9

Statement	Comments
> print(z<-x+y)	#Adds two vectors
> print(z<-x-y)	#Subtracts second vector from the first
> print(z<-x*y)	#Multiplies both vectors
> print(z<-x/y)	#Divide the first vector with the second
> print(z<-(v^y)	#The first vector raised to the exponent of second vector

- **ARITHMETIC OPERATORS-Recycling Element**
- > x<-1:8 ; y<-6:9

Statement	Comments
> print(z<-x+y)	#Adds two vectors
> print(z<-x-y)	#Subtracts second vector from the first
> print(z<-x*y)	#Multiplies both vectors
> print(z<-x/y)	#Divide the first vector with the second
> print(z<-(v^y)	#The first vector raised to the exponent of second vector

Vectors

- **ARITHMETIC OPERATORS-Recycling Element (be careful)**
- > x<-1:8 ; y<-6:8

Statement	Comments
> print(z<-x+y)	#Adds two vectors
> print(z<-x-y)	#Subtracts second vector from the first
> print(z<-x*y)	#Multiplies both vectors
> print(z<-x/y)	#Divide the first vector with the second
> print(z<-(v^y)	#The first vector raised to the exponent of second vector

- v <- c(2,5.5,6);t <- c(8, 3, 4)

Statement	Output	Comments
> print(z<-(v%%t))	[1] 2.0 2.5 2.0	Give the remainder of the first vector with the second(%%))-Modulo (Remainder)
> print(z<-(v%/%t))	[1] 0 1 1	The result of division of first vector with second (quotient)-(%/%))-Integer Quotients

Vectors

- **2- Vectorized Operations (RELATIONAL OPERATORS):**
- > x<-1:4; y<-6:9

Statement	Comments
> x>y	Checks if each element of the first vector is greater than the corresponding element of the second vector.
> x>=y	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.
> x<y	(Checks if each element of the first vector is less than the corresponding element of the second vector.)
> x==y	(Checks if each element of the first vector is equal to the corresponding element of the second vector.)
> x<=y	(Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.)
> x!=y	(Checks if each element of the first vector is unequal to the corresponding element of the second vector.)

- Notice that these logical operations return a logical vector of TRUE or FALSE.

Vectors

- **RELATIONAL OPERATORS-Recycling Element**
- > x<-1:8; y<-6:9

Statement	Comments
> x>y	Checks if each element of the first vector is greater than the corresponding element of the second vector.
> x>=y	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.
> x<y	(Checks if each element of the first vector is less than the corresponding element of the second vector.)
> x==y	(Checks if each element of the first vector is equal to the corresponding element of the second vector.)
> x<=y	(Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.)
> x!=y	(Checks if each element of the first vector is unequal to the corresponding element of the second vector.)

- Notice that these logical operations return a logical vector of TRUE or FALSE.

Vectors

- **RELATIONAL OPERATORS-Recycling Element (be careful)**
- > x<-1:4; y<-6:8

Statement	Comments
> x>y	Checks if each element of the first vector is greater than the corresponding element of the second vector.
> x>=y	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.
> x<y	Checks if each element of the first vector is less than the corresponding element of the second vector.
> x==y	Checks if each element of the first vector is equal to the corresponding element of the second vector.
> x<=y	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.
> x!=y	Checks if each element of the first vector is unequal to the corresponding element of the second vector.

- Notice that these logical operations return a logical vector of TRUE or FALSE.

Vectors

- **3- Vectorized Operations (LOGICAL OPERATORS):**

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Operator	Description	Example
&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.	v <- c(3,1,TRUE,2+3i) t <- c(4,1,FALSE,2+3i) print(v&t) it produces the following result – [1] TRUE TRUE FALSE TRUE
|	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.	v <- c(3,0,TRUE,2+2i) t <- c(4,0,FALSE,2+3i) print(v|t) it produces the following result – [1] TRUE FALSE TRUE TRUE
!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	v <- c(3,0,TRUE,2+2i) print(!v) it produces the following result – [1] FALSE TRUE FALSE FALSE

Vectors

- **4- Miscellaneous Operators**

- These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example

%in%	This operator is used to identify if an element belongs to a vector.	`v1 <- 8`
		`v2 <- 12`
		`t <- 1:10`
		`print(v1 %in% t)`
		`print(v2 %in% t)`
		it produces the following result –
		`[1] TRUE`
		`[1] FALSE`

Vectors

- **Using sequence (Seq.) operator**
- > print(a<-seq(5,9,by=0.5))
- [1] 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0

- **Argument length.out**
- > print(a <- seq(1, 10, length.out = 100))

- **Replicate Elements of Vectors**
- > rep(1:4, 2)
- > rep(1:4, each = 2)
- > rep(1:4, c(2,2,2,2))
- > rep(1:4, c(2,1,2,1))
- > rep(1:4, each = 2, len = 4)
- > rep(1:4, each = 2, len = 10)
- > rep(1:4, each = 2, times = 3)
- > rep(letters[1:5],3)

Vectors

Vector Element Sorting

Statement	Output
> sort(v<-c(3,8,4,5,0,11,-9,304))	[1] -9 0 3 4 5 8 11 304
> sort(v,decreasing = TRUE)	[1] 304 11 8 5 4 3 0 -9
> sort(v,decreasing = FALSE)	[1] -9 0 3 4 5 8 11 304
> sort(v <- c("Red","Blue","yellow","violet"))	[1] "Blue" "Red" "violet" "yellow“
> sort(v,decreasing = TRUE)	[1] "yellow" "violet" "Red" "Blue"
> sort(v,decreasing = FALSE)	[1] "Blue" "Red" "violet" "yellow"

Vectors

- > order(v<-c(3,8,4,5,0,11,-9,304))
- [1] 7 5 1 3 4 2 6 8
- > max(v<-c(3,8,4,5,0,11,-9,304))
- [1] 304
- > min(v<-c(3,8,4,5,0,11,-9,304))
- [1] -9
- > rank(v<-c(3,8,4,5,0,11,-9,304))
- [1] 3 6 4 5 2 7 1 8
- > which.max(v<-c(3,8,4,5,0,11,-9,304))
- [1] 8
- > which.min(v<-c(3,8,4,5,0,11,-9,304))
- [1] 7
- > mean(v<-c(3,8,4,5,0,11,-9,304))
- [1] 40.75

V	Index	Sort	Order	Rank
3	1	-9	7	3
8	2	0	5	6
4	3	3	1	4
5	4	4	3	5
0	5	5	4	2
11	6	8	2	7
-9	7	11	6	1
304	8	304	8	8

Subsetting R Objects

- There are three operators that can be used to extract subsets of R objects.
- 1- The [operator always returns an object of the same class as the original. It can be used to select multiple elements of an object
- 2- The [[operator is used to extract elements of a **list or a data frame**. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- 3- The $ operator is used to extract elements of a **list or data frame** by literal name. Its semantics are similar to that of [[.

Vectors

- **SUBSETTING R OBJECTS**

- Vectors are basic objects in R and they can be subsetted using [operator.

Statement	Output
> print(x<-c("a","b","c","d","e","f"))	[1] "a","b","c","d","e","f"
> x[1]	[1] “a"
> x[1:4]	[1] "a","b","c","d"
> x[c(1,3,4)]	[1] "a","c","d"
> x[-1]	[1] "b","c","d","e","f"
x[-c(1,2)]	[1] "c","d","e","f"

Vectors

- **SUBSETTING R OBJECTS**
- We can also pass a logical sequence to the [operator to extract elements of a vector that satisfy a given condition. For example, here we want the elements of x that come lexicographically *after* the letter “a”.

Statement	Output
> print(y<-x>"a")	[1] FALSE TRUE TRUE TRUE TRUE TRUE
> x[y]	[1] "b" "c" "d" "e" "f"

- Another, more compact, way to do this would be to skip the creation of a logical vector and just subset the vector directly with the logical expression.

Statement	Output
> x[x>"a"]	[1] "b" "c" "d" "e" "f“

Statement	Output
> students<-c("Umer=91","Ali=92","Ahmed=93")	
> students[c(TRUE,FALSE,TRUE)]	[1] "Umer=91" "Ahmed=93"
> students[c(TRUE,FALSE)]	[1] "Umer=91" "Ahmed=93"
> students[c(TRUE)]	[1] "Umer=91" "Ali=92" "Ahmed=93"

Vectors

- **names () function**
- R Objects can also have names, which is very useful for writing readable code and sefl-describing objects.

Statement	Output
> print(x<-1:3)	[1] 1 2 3
> names(x)	NULL
> names(x)<-c("Umer","Ali","Ahmed")	
> x	Umer Ali Ahmed 1 2 3
> names(x)	[1] Umer Ali Ahmed

Statement	Output
> marks<-c(99,98,91)	
> students<-c("Umer","Ali","Ahmed")	
> names(students)<-marks	
> students	99 98 91 "Umer" "Ali" "Ahmed"

Vectors

Statement	Output
> print(students_marks<-c("Umer=99","Ali=98","Ahmed=91"))	
> print(students_marks<-c('Umer=99','Ali=98','Ahmed=91'))	
> print(students_marks<-c(Umer=99,Ali=98,Ahmed=91))	

Introduction To Data Science Using R Programming

Matrices

Matrices

- Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. (However Vector is 1D array of data elements)
- They contain elements of the same atomic types.
- Though we can create a matrix containing only characters or only logical values, they are not of much use.
- Collection of data elements, but this time arranged into a fixed number of rows and columns.
- We use matrices containing numeric elements to be used in mathematical calculations.
- A Matrix is created using the **matrix()** function.
- **Syntax**
- The basic syntax for creating a matrix in R is − matrix(data, nrow, ncol, byrow, dimnames)
- Following is the description of the parameters used −
  - **data** is the input vector which becomes the data elements of the matrix.
  - **nrow** is the number of rows to be created.
  - **ncol** is the number of columns to be created.
  - **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
  - **dimname** is the names assigned to the rows and columns.

Matrices

- Matrices are vectors with a *dimension* attribute.
- The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)
- > print(a<-matrix(nrow=2,ncol=3))
- > dim(a)
- > attributes(a)
- Matrices can also be created directly from vectors by adding a dimension attribute.
- > a<-1:10
- > print(dim(a)<-c(2,5))
- Matrices are constructed *column-wise*, so entries can be thought of starting in the “upper left” corner and running down the columns.
- # Elements are arranged sequentially by column.
- > print(a <- matrix(1:6, nrow = 2, ncol = 3))
- # Elements are arranged sequentially by column.
- > print(a <- matrix(1:6, nrow = 3, ncol = 2))

Matrices

- # Elements are arranged sequentially by row.
- > print(M <- matrix(3:14, nrow = 4, ncol=3, byrow = TRUE))
- # Elements are arranged sequentially by column.
- > print(M <- matrix(3:14, nrow = 4, ncol=3, byrow = FALSE))
- # Define the column and row names.
- > rownames = c("row1", "row2", "row3", "row4")
- > colnames = c("col1", "col2", "col3")
- > print(P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames)))
- # Matrix recycling
- > print(a<-matrix(1:3,nrow=2,ncol=3))
- # Matrix recycling
- > print(a<-matrix(1:3,nrow=2,ncol=2))
- Warning message:
- In matrix(1:3, nrow = 2, ncol = 2) :
- data length [3] is not a sub-multiple or multiple of the number of rows [2]

Matrices

- Matrices can be created by *column-binding* or *row-binding* with the cbind() and rbind() functions.
- > a<-1:3
- > b<-11:13
- > cbind(a,b)
- > rbind(a,b)
- > print(a<-rbind(1:3,4:6))
- > print(a<-cbind(1:3,4:6))

Matrices

- # How to Add more row or col in the matrix
- > print(m<-matrix(1:6,byrow = TRUE,nrow=2))
- > rbind(m,7:9)
- > cbind(m,c(10,11))
- ----------
- > a<-matrix(1:4,nrow = 2,ncol=2)
- > b<-matrix(LETTERS[1:4],nrow=2,ncol = 2)
- > print(c<-rbind(a,b))
- > print(d<-cbind(a,b))
- > class (c)
- > class (d)

Matrices

- Various mathematical operations are performed on the matrices using the R operators.
- The result of the operation is also a matrix.
- The dimensions (number of rows and columns) should be same for the matrices involved in the operation.
- Matrix Addition & Subtraction
- # Create two 2x3 matrices.
- > print(matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2))
- > print(matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2))
- # Add the matrices.
- > print(matrix3<-matrix1+matrix2)
- # subtraction the matrices.
- > print(matrix4<-matrix1-matrix2)
- Matrix Multiplication & Division
- # Multiplication the matrices.
- > print(matrix5<-matrix1*matrix2)
- # Division the matrices.
- > print(matrix5<-matrix1/matrix2)

Matrices

- **Vectorized Matrix Operations**
- Matrix operations are also vectorized, making for nicely compact notation.
- This way, we can do element –by-element operations on matrices without having to loop over every element.
- > x<-matrix(1:4,2,2)
- > y<-matrix(rep(10,4),2,2)
- # element wise multiplication
- > a<-x*y
- # element wise division
- > b<-x/y
- # True matrix multiplication
- > c<-x%*%y
- # transpose of x
- > d<-t(x)

Matrices

- **Accessing Elements of a Matrix**
- Elements of a matrix can be accessed by using the column and row index of the element.
- # Define the column and row names.
- > rownames = c("row1", "row2", "row3", "row4")
- > colnames = c("col1", "col2", "col3")
- > print(P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames)))
- > # Access the element at 3rd column and 1st row.
- > print(P[1,3])
- # Access the element at 2nd column and 4th row.
- > print(P[4,2])
- # Access only the 2nd row.
- > print(P[2,])
- # Access only the 3rd column.
- > print(P[,3])

Matrices

- **Subsetting a Matrix**
- > print(x<-matrix(1:6,2,3))
- > x[1,2] # Extract the 1st row and 2nd column.
- > x[2,1] # Extract the 2nd row and 1st column.
- > x[1,] # Extract the 1st row.
- > x[,2] # Extract the 2nd column.
- >x[4]
- > x[2,c(2,3)]
- **Dropping Matrix Dimensions**
- By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. Often, this is exactly what we want, but this behavior can be turned off by setting drop = FALSE.
- > print(x<-matrix(1:6,2,3))
- > x[1,2,drop=FALSE]
- > x[2,1,drop=FALSE]
- > x[1, ,drop=FALSE]
- > x[,2,drop=FALSE]

Matrices

- **Identical Matrices**
- > a <- matrix(1:6, nrow = 2, ncol = 3)
- > b <- matrix(1:6, nrow = 2, ncol = 3)
- > identical(a,b)
- **Fix Functions**
- > a <- matrix(1:6, nrow = 2, ncol = 3)
- > fix(a)

Introduction To Data Science Using R Programming

Lists

Lists

- Lists are a special type of vector that can contain elements of different classes.
- Lists can be explicitly created using the list() function, which takes an arbitrary number of arguments.
- > print(x <- list(1, "a", TRUE, 1 + 4i))
- We can also create an empty list of a pre-specified length with the vector() function.
- > print(x <- vector("list", length = 4))
- **Naming List Elements**
- # Create a list containing a vector, a matrix and a list.
- > list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))
- # Give names to the elements in the list.
- > print(names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list"))
- # Show the list.
- > print(list_data)

Lists

- **Accessing List Elements**
- Elements of the list can be accessed by the index of the element in the list.
- In case of named lists it can also be accessed using the names.
- # Create a list containing a vector, a matrix and a list.
- > list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))
- # Give names to the elements in the list.
- > names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
- # Access the first element of the list.
- > print(list_data[1])
- # Access the third element. As it is also a list, all its elements will be printed.
- > print(list_data[3])
- # Access the list element using the name of the element.
- > print(list_data$A_Matrix)

Lists

- **Manipulating List Elements**
- We can add, delete and update list elements.
- We can add and delete elements only at the end of a list.
- But we can update any element.
- # Create a list containing a vector, a matrix and a list.
- > list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))
- # Give names to the elements in the list.
- > names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
- # Add element at the end of the list.
- > list_data[4] <- "New element"
- > print(list_data[4])
- # Remove the last element.
- > list_data[4] <- NULL
- > print(list_data[4])
- # Update the 3rd Element.
- > list_data[3] <- "updated element"
- > print(list_data[3])

Lists

- **Merging Lists Function**
- You can merge many lists into one list by placing all the lists inside one list() function.
- # Create two lists.
- list1 <- list(1,2,3);list2 <- list("Sun","Mon","Tue")
- # Merge the two lists.
- > print(merged.list <- c(list1,list2))
- **Converting List to Vector**
- A list can be converted to a vector so that the elements of the vector can be used for further manipulation.
- All the arithmetic operations on vectors can be applied after the list is converted into vectors.
- To do this conversion, we use the **unlist()** function. It takes the list as input and produces a vector.
- > list1 <- list(1:5) ; list2 <-list(10:14)
- > v1 <- unlist(list1) ; v2 <- unlist(list2)
- > print(v3<- v1+v2)

Lists

- **Subsetting Lists**
- Lists in R can be subsetted using all three of the operators mentioned above and all there are used for different purposes.
- > print(x<-list(foo=1:4,bar=0.6))
- 1- The [[operator can be used to extract single elements for a list.
- > x[[2]]
- [1] 0.6
- 2- The [[operator can also use named indices so that you don't have to remember the exact ordering of every element of the list.
- > x[["foo"]]
- 3- You can also use the $ operator to extract elements by name.
- > x$foo
- Notice you don't need that quotes when you use the $ operator.

Lists

- **Subsetting Lists**
- One thing that differentiates the [[operator from the $ is that the [[operator can be used with *computed* indices. The $ operator can only be used with literal names.
- > x<-list(foo=1:4,bar=0.6,baz="hello")
- > x[["foo"]]
- > x[[1]]
- > x$foo
- > name<-"foo"
- > x[[name]]
- [1] 1 2 3 4
- > x$name
- NULL

Lists

- **Subsetting Nested Elements of a List**
- The [[operator can take an integer sequence if you want to extract a nested element of a list.
- > x<-list(a=list(10,12,14),b=c(3.14,2.81))
- > # Get the 3rd element of the 1st element.
- > x[[c(1,3)]]
- [1] 14
- > x[[1]][[3]]
- [1] 14

Lists

- **Extracting Multiple Element of a List**
- The [operator can be used to extract *multiple* elements from a list.
- > x<-list(foo=1:4,bar=0.6,baz="hello")
- > #extract the first and third elements of a list, you would do the following
- > x[c(1,3)]
- $foo
- [1] 1 2 3 4
- $baz
- [1] "hello“
- Note that x[c(1, 3)] is NOT the same as x[[c(1, 3)]].
- Remember that the [operator always returns an object of the same class as the original.

Lists

- **Partial Matching**
- Partial matching of names is allowed with [[and $.
- This is often very useful during interactive work if the object you're working with has very long element names. You can just abbreviate those names and R will figure out what element you're referring to.
- > x<-list(aardvark=1:5)
- > x$a
- [1] 1 2 3 4 5
- > x[["a"]]
- NULL
- > x[["a",exact=FALSE]]
- [1] 1 2 3 4 5

Introduction To Data Science Using R Programming

R - Arrays

R - Arrays

- Arrays are the R data objects which can store data in more than two dimensions.
- For example − If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. Arrays can store only data type.
- An array is created using the **array()** function.
- It takes vectors as input and uses the values in the **dim** parameter to create an array.
- > # Create two vectors of different lengths.
- > vector1 <- c(5,9,3)
- > vector2 <- c(10,11,12,13,14,15)
- > # Take these vectors as input to the array.
- > print(result <- array(c(vector1,vector2),dim = c(3,3,2)))

R - Arrays

- **Naming Columns and Rows**
- We can give names to the rows, columns and matrices in the array by using the **dimnames** parameter.
- > # Create two vectors of differnt lengths
- > vector1 <- c(5,9,3)
- > vector2 <- c(10,11,12,13,14,15)
- > column.names <- c("COL1","COL2","COL3")
- > row.names <- c("ROW1","ROW2","ROW3")
- > matrix.names <- c("Matrix1","Matrix2")
- > # Take these vectors as input to the array.
- > print(result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames = list(row.names,column.names,matrix.names))

R - Arrays

- **Accessing Array Elements**
- > # Create two vectors of different lengths.
- > vector1 <- c(5,9,3)
- > vector2 <- c(10,11,12,13,14,15)
- > # Take these vectors as input to the array.
- > print(result <- array(c(vector1,vector2),dim = c(3,3,2)))
- > # Print the third row of the second matrix of the array
- > print(result[3,,2])
- > # Print the element in the 1st row and 3rd column of the 1st matrix.
- > print(result[1,3,1])
- > # Print the 2nd Matrix
- > print(result[,,2])

R - Arrays

- **Manipulating Array Elements**
- As array is made up matrices in multiple dimensions, the operations on elements of array are carried out by accessing elements of the matrices.
- # Create two vectors of different lengths.
- > vector1 <- c(5,9,3)
- > vector2 <- c(10,11,12,13,14,15)
- # Take these vectors as input to the array.
- > array1 <- array(c(vector1,vector2),dim = c(3,3,2))
- # Create two vectors of different lengths.
- > vector3 <- c(9,1,0)
- > vector4 <- c(6,0,11,3,14,1,2,6,9)
- > array2 <- array(c(vector1,vector2),dim = c(3,3,2))
- # create matrices from these arrays.
- > matrix1 <- array1[,,2]
- > matrix2 <- array2[,,2]
- # Add the matrices.
- > print(result <- matrix1+matrix2)

Introduction To Data Science Using R Programming

Factors

Factors

- Factors are used to represent categorical data and can be unordered or ordered.
- One can think of a factor as an integer vector where each integer has a *label*.
- Using factors with labels is *better* than using integers because factors are self-describing. Having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.
- Factors are important in statistical modeling and are treated specially by modeling functions like *lm()* and *glm()*
- Factor objects can be created with the factor() function.

- > print(x <- factor(c("yes", "yes", "no", "yes", "no")))
- > table(x)
- > attributes(x)
- > unclass(x)

Factors

- Often factors will be automatically created for you when you read a dataset in using a function like ***read.table()***.
- Those functions often default to creating factors when they encounter data that look like characters or strings.
- The order of the levels of a factor can be set using the **levels argument** to factor().
- This can be important in linear modeling because the first level is used as the baseline level.
- > x <- factor(c("yes", "yes", "no", "yes", "no"))
- The order of the levels can be set using the levels argument to factor(). This can be important in linear modeling because the first level is used as a baseline level.
- > print(x <- factor(c("yes", "yes", "no", "yes", "no"),levels = c("yes","no")))

Introduction To Data Science Using R Programming

R - Strings

R - Strings

- **Rules Applied in String Construction**
- 1- The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
- > print(a <- 'Start and end with single quote‘)
- > print(a <- “Start and end with single quote”)
- > print(a <- “Start and end with single quote’) #Examples of Invalid Strings
- 2- Double quotes can be inserted into a string starting and ending with single quote.
- > print(a<- 'Double quotes " in between single quote‘)
- 3-Single quote can be inserted into a string starting and ending with double quotes.
- > print(a<- "single quote ' in between double quotes“)
- 4- Double quotes can not be inserted into a string starting and ending with double quotes.
- > print(a <- "Double quotes " inside double quotes“) #Examples of Invalid Strings
- 5- Single quote can not be inserted into a string starting and ending with single quote.
- > print(a <- 'Single quote ' inside single quote‘) #Examples of Invalid Strings

R - Strings

- **Concatenating Strings - paste() function**
- Many strings in R are combined using the **paste()** function.
- It can take any number of arguments to be combined together.
- **Syntax**
- The basic syntax for paste function is −
- paste(..., sep = " ", collapse = NULL)
- Following is the description of the parameters used −
  - **...** represents any number of arguments to be combined.
  - **sep** represents any separator between the arguments. It is optional.
  - **collapse** is used to eliminate the space in between two strings. But not the space within two words of one string.

```
> a<-"Hello"
> b<-'How'
> c<-"are you?"
> print(paste(a,b,c))                    # Example-1
> print(paste(a,b,c, sep = "-"))         # Example-2
> print(paste(a,b,c, sep = "", collapse = ""))   # Example-3
```

R - Strings

- **Formatting numbers & strings - format() function**
- Numbers and strings can be formatted to a specific style using **format()** function.
- **Syntax**
- The basic syntax for paste function is −
- format(x, digits, nsmall, scientific, width, justify = c("left", "right", "centre", "none"))
- Following is the description of the parameters used −
  - **x** is the vector input.
  - **digits** is the total number of digits displayed.
- # Total number of digits displayed. Last digit rounded off.
- > print(result <- format(23.123456789, digits = 9))
  - **nsmall** is the minimum number of digits to the right of the decimal point.
- # The minimum number of digits to the right of the decimal point.
- > print(result <- format(23.47, nsmall = 5))

R - Strings

- **scientific** is set to TRUE to display scientific notation.
- # Display numbers in scientific notation
- > print(result <- format(c(6, 13.14521), scientific = TRUE))
- **width** indicates the minimum width to be displayed by padding blanks in the beginning.
- # Numbers are padded with blank in the beginning for width.
- > print(result <- format(13.7, width = 6))
- **justify** is the display of the string to left, right or center.
- # Left justify strings.
- > print(result <- format("Hello", width = 8, justify = "l"))
- # Justfy string with center
- > print(result <- format("Hello", width = 8, justify = "c"))
- # Format treats everything as a string.
- > print(result <- format(6))

R - Strings

- **Counting number of characters in a string - nchar() function**
- This function counts the number of characters including spaces in a string.
- **Syntax**
- The basic syntax for nchar () function is − nchar(x)
- Following is the description of the parameters used −
- **x** is the vector input.
- > print(result <- nchar("Umer Saeed"))
- [1] 10
- **Changing the case - toupper() & tolower() functions**
- These functions change the case of characters of a string.
- **Syntax**
- The basic syntax for toupper() & tolower() function is − toupper(x) & tolower(x)
- Following is the description of the parameters used −
- **x** is the vector input.

R - Strings

- # Changing to Upper case.
- > print(result <- toupper("umer saeed"))
- # Changing to lower case.
- > print(result <- tolower("UMER SAEED"))
- **Extracting parts of a string - substring() function**
- This function extracts parts of a String.
- **Syntax**
- The basic syntax for substring() function is −substring(x,first,last)
- Following is the description of the parameters used −
- **x** is the character vector input.
- **first** is the position of the first character to be extracted.
- **last** is the position of the last character to be extracted.
- # Extract characters from 5th to 7th position.
- > print(result <- substring("Extract", 5, 7))

Introduction To Data Science Using R Programming

R - Data Frames

R - Data Frames

- A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.
- Following are the characteristics of a data frame:
- 1- The column names should be non-empty.
- 2- The row names should be unique.
- 3-The data stored in a data frame can be of numeric, factor or character type.
- 4- Each column should contain same number of data items
- 5- They are represented as a special type of list where every element of the list has to have the same length
- 6- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows
- 7- Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class
- 8- Data frames also have a special attribute called **row.names**
- 9- Data frames are usually created by calling read.table() or read.csv()
- 10-Can be converted to a matrix by calling data.matrix()

R - Data Frames

```
> # Create and print the data frame.
> print(emp.data <- data.frame(
emp_id = c (1:5),
emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
 salary = c(623.3,515.2,611.0,729.0,843.25),
 start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
 "2015-03-27")),
stringsAsFactors = FALSE))
```

Get the Structure of the Data Frame

The structure of the data frame can be seen by using **str()** function.

```
> # Create and print the data frame & Get the structure of the data frame.
> str(emp.data)
```

R - Data Frames

- **Summary of Data in Data Frame**
- The statistical summary and nature of the data can be obtained by applying **summary()** function;
- > summary(emp.data)
- **Extract Data from Data Frame**
- Extract specific column from a data frame using column name.
- > # Extract Specific columns.
- > print(result <- data.frame(emp.data$emp_name,emp.data$salary))
- Extract the first two rows and then all columns.
- > # Extract first two rows.
- > print(result <- emp.data[1:2,])
- > # Extract 3rd and 5th row with 2nd and 4th column.
- > print(result <- emp.data[c(3,5),c(2,4)])

R - Data Frames

- **Expand Data Frame**
- A data frame can be expanded by adding columns and rows.
- **Add Column**
- Just add the column vector using a new column name.
- > # Add the "dept" coulmn.
- > emp.data$dept <- c("IT","Operations","IT","HR","Finance")
- > print(emp.data)

R - Data Frames

- **Add Row**
- To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the **rbind()** function.
- > # Create the second data frame
- emp.newdata <- data.frame(
- emp_id = c (6:8),
- emp_name = c("Rasmi","Pranab","Tusar"),
- salary = c(578.0,722.5,632.8),
- start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17"))
- > # Bind the two data frames.
- > print(emp.finaldata <- rbind(emp.data,emp.newdata))

Introduction To Data Science Using R Programming

Missing Values

Missing Values

- Missing values are denoted by NA or NaN for undefined mathematical operations.
- 1- is.na() is used to test objects if they are NA.
- 2- is.nan() is used to test for NaN.
- 3- NA values have a class also, so there are integer NA, character NA, etc.
- 4- A NaN values is also NA but the converse is not true.
- > # Create a vector with NAs in it.
- > class(x<-c(1,2,NA,10,3))
- > # Return a logical vector indicating which elements are NA.
- > is.na(x)
- > # Now create a vector with both NA and NaN values.
- > class(x<-c(1,2,NA,NaN,4))
- > is.na(x)
- > is.nan(x)

Missing Values

- **REMOVING "NA" VALUES**
- > x<-c(1,2,NA,4,NA,5)
- > bad<-is.na(x)
- > bad
- > x[!bad]

- > x<-c(1,2,NA,4,NA,5)
- > y<-c("a","b",NA,"d",NA,"f")
- > print(good<-complete.cases(x,y))
- > x[good]
- > y[good]

Missing Values

- **REMOVING "NA" VALUES**
- > head(airquality)
- > good<-complete.cases(airquality)
- > head(airquality[good,])
- ** airquality date stored in R Language.*
- **------**
- > a<-c(1,2,3,4,5,NA)
- > mean(a,na.rm=TRUE

Introduction To Data Science Using R Programming

Install Packages/Library/Import data

Install Packages/Library/Import data

- **How to Install Packages in R**
- > install.packages("dslabs")
- > library(dslabs)
- **How to set working directory in Rstudio**
- Session →Set Working Directory →Choose Directory
- > getwd() #how to get working directory
- > setwd("D:/MS DS/1st Semester/Business Analytics and Strategy/Data Sets")
- > data<-read.csv("mvtWeek01.csv")
- #how to get working directory
- > View(WHO)
- **How to import Files in Rstudio from Other Sources**
- Import Dataset--> From CSV/Excel/SPSS/SAS/Stata
- #how to import data from the clipboard
- > a<-read.table("clipboard",header=TRUE,sep="\t")
- # how to removes rows and columns with negative indexing.
- > a<-a[-31:33,-10:-12] Rows, Columns

Introduction To Data Science Using R Programming

Reading and Writing Data

Reading and Writing Data

- There are a few principal functions reading data in R.
- 1- *read.table, read.csv,* for reading tabular data.
- 2- *readLines*, for reading lines
- 3- *source,* for reading in R code files (inverse of dump)
- 4- *dget,* for reading in R code files (inverse of dput)
- 5- *load,* for reading in saved workspaces
- 6- *unserialize,* for reading single R objects in binary form.

Reading and Writing Data

- There are analogous functions for writing data to files.
- 1- *write.table,* for writing tabular data to text file(i.e CSV) or connections.
- 2- *writeLines,* for writing character data lines-by-line to a file or connection.
- 3-*dump,* for dumping a textual representation of multiple R objects.
- 4-*dput,* for outputting a textual representation of an R object.
- 5-*serialize*, for converting an R object into a binary format for outputting to a connection or file.
- 6- save

Reading and Writing Data

- **Reading Data Files with read.table ()**
- The read.table() function is on of the most commonly used functions for reading data.
- The read.table() function has a few important argument;
- 1- *file*, the name of a file, or a connection
- 2-*header*, logical indicating if the file has a header line
- 3-*sep*, a string indicating how the columns are separated
- 4-*colClasses*, a character vector indicating the class of each column in the dataset
- 5-*nrows*, the number of rows in the dataset. By default read.table() reads an entire file.
- 6-*comment.char*, a character string indicating the comment character. This defaults to "#". If there are no commented lines in your file, it's worth setting this to be the empty string "".
- 7- *skip*, the number of lines to skip from the beginning
- 8- *stringsAsFactors*, should character variables be coded as factors?

Reading and Writing Data

- For small to moderately sized datasets, you can usually call read.table without specifying any other arguments.
- data_csv<-read.table("WHO.csv",sep = ",",header = TRUE, quote = "\"")
- data_txt<-read.table("mvtWeek01.txt",header = TRUE,sep = ",")
- In this case, R will automatically;
- 1- Skip lines that begin with a #.
- 2- Figure out how may rows there are (and how much memory needs to be allocated)
- 3- Figure what type of variable is in each column of the table.
- The read.csv() function is identical to read.table except that some of the defaults are set differently (like the sep argument).
- > data<-read.csv("mvtWeek1.csv")

Using the readr Package

- The readr package is devloped by Hadly Wickham to deal with reading in large flat files quickly.
- The package provides replacement for functions like read.table() and read.csv. (utils; verbose, slower)
- The analogous functions in readr are read_table() ,read_delim(), read_tsv and read_csv(). This functions are oven much faster than their base R analogues and provide a few other nice features such as progress meters.
- **Step-1:** >install.packages("reader")
- **Step-2:** >library(readr)
- **Step-3:** >getwd()
- **Step-4:** >setwd()
- **Step-5:** >dir()
- **Step-6:** Import required data using read_table(),read_delim(),read_tsv() and read_csv().
- read_table() and read_table2() are designed to read the type of textual data where each column is #' separate by one (or more) columns of space.
- > other<-read_table("massey-rating.txt")
- > other<-read_table2("massey-rating.txt")
- > other<-read_delim("mvtWeek1.csv",delim = ",")
- > other<-read_csv("mvtWeek1.csv")
- > other <-read_tsv("mvtWeek1.csv")

Reading and Writing Data

- **Reading in Larger Dataset with read.csv:**
- Which much larger datasets, doing the following things will make our life easier and will prevent R from choking,
- 1- Read the help page for read.csv, which contains many hints.
- 2- Make a rough calculation of the memory required to store your datasets. If the dataset is lager than the amount of RAM on your computer, you can probably stop here.
- 3- Set comment.char=“ ”if there are nor commented lines in your file.

Reading and Writing Data

- **Reading in Larger Dataset with read.csv:**
- Use the colClasses argument. Specifying this option instead of using the default can make ‘read.table’ run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are “numeric”, for example, then you just set colClasses=“numeric”. A quick and diry way to fifer out the classes of each column is the following;
- > a<-read.csv("mvtWeek01.csv",nrow=100)
- > classes<-sapply(a,class)
- > tabAll<-read.csv("mvtWeek01.csv",colClasses = classes)

Reading and Writing Data

- **Know Thy System:**
- In general, when using R with large datasets, it's useful to know a few things about your system.
- 1- How much memory is available?
- 2- What other application are in use?
- 3- Are there other users logged into the same system?
- 4- What operating system?
- 5- Is the OS 32 or 64 bit?

Reading and Writing Data

- **Calculating Memory Required for R Ojbects:**
- R stores all of its objects physical memory, it is important to be cognizant of how much memory is being used up by all of the data objects residing in your workspace.
- **Example**
- Suppose I have a data frame with 1,500,000 rows and 120columns, all of which are numeric data.
- Roughly, how much memory is required to store this data frame?
- On most modern computers double precision floating point numbers are stored using 64 bits (8 bytes) of memory. Given that information, you can do the following calculation;
- 1,500,000*120*8bytes/numberic=1,440,000,000 bytes=1,440,000,000/2^20 bytes/MB
  =1,373.29MB=1.34GB
- So the dataset whould required about 1.34 GB of RAM. Most computers these days have at least that much RAM.
- However, you need to be aware of;
- 1- What other programs might be running on your computer, using up RAM.
- 2- What other R objects might already be taking up RAM in your workspace.

Reading and Writing Data

- **Textual Formats**
- dumping and dputing are useful because the result textual format is edit-able, and in the case corrupting, potentially recoverable.
- Unlike writing out a table or csv file, dump and dput preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.
- Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files.
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem
- Textual formats adhere to the "Unix philosophy"
- Downside: The format is not very space-efficient

Reading and Writing Data

- **Using Textual and Binary Formats for storing data:**
- **dupt-ting R Objects**
- Another way to pass data around is by deparsing the R object with duput and reading it back is using dget.
- > y<-data.frame(a=1,b="a")
- > dput(y)
- > dput(y,file="y.R")
- > print(new.y<-dget("y.R"))
- **dumping R Objects**
- Multiple objects can be deparsed using the dump function and read back in using source.
- > x<-"foo“
- > y<-data.frame(a=1,b="a")
- > dump(c("x","y"),file="data.R")
- > source("data.R")
- > y
- > x

Reading and Writing Data

- **Interfaces to the Outside World:**
- Data are read using connection interfaces. Connections can be make to files (most common) or to other more exotic things.
- file, opens a connection to a file
- gzfile, Opens a connection to a file compressed with gzip.
- bzfile, Opens a connection to a file compressed with bzip2.
- url, opens a connection to a webpage.
- **File Connections:**

```
> str(file)
function (description = "", open = "", blocking = TRUE,
    encoding = getOption("encoding"), raw = FALSE,
    method = getOption("url.method", "default"))
```

- description is the name of the file.
- Open is a code indicating.
- - “r” read only
- -”w” writing (and initializing a new file)
- - “a” appending
- - “rb”,”wb”,”ab” reading, writing or appending in binary mode (Windows)

Reading and Writing Data

- **Connections**
- In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't to deal with the connection interface directly.
- > con<-file("WHO.txt","r")
- > data<-read.csv(con)
- > close(con)
- Is the same as;
- > Data<-read.csv("WHO.txt")
- **Reading Lines of a Text File**
- > con<- gzfile("words.gz")
- > print(x<-readLines (con,10))
- writeLines takes a character vector and writes each element one line at a time to a text file.
- readLines can be useful for reading in lines of webpages.
- > con<-url("https://www.umt.edu.pk/","r")
- x<-readLines(con)
- > head(x)

Introduction To Data Science Using R Programming

Date and Time

Date And Time

- R has developed a special representation of date and time;
- Date are represented by the Date class
- Time are represented by the POSIXct or the POSIXlt class
- Date are stored internally as the number of days since 1970-01-01
- Time are stored internally as the number of seconds since 1970-01-01
- **Date in R**
- Dates are represented by the Date class and can be coerced from a character string using the as.Date() function.
- # Coerce a 'Date' object from character
- > print(x<-as.Date("1970-01-01"))
- You can see the internal representation of a Date object by using unclass() function.
- > unclass(x)
- > unclass(as.Date("1970-01-02"))
- # if date is before 1970-01-01 it shows –ve number,
- > unclass(as.Date("1960-01-02"))

Date And Time

- **Time in R:**
- Times are represented using the POSIXct or the POSIXlt class.
- *POSIXct* is just a very large integer under the hood; it use a useful class when you want to store times in something like a data frame.
- *POSIXlt* is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month
- There are a number of generic functions that work on dates and times
- weekdays: give the day of the week
- > weekdays(as.Date("2017-08-17"))
- months: give the month name
- > months(as.Date("2017-08-17"))
- quarters: give the quarter number (“Q1”, “Q2”, “Q3”, or “Q4”)
- > quarters(as.Date("2017-08-17"))

Date And Time

- **TIMES IN R**
- Times can be coerced from a character string using the as.POSIXlt or as.POSIXct function.
- > class(x<-Sys.time())
- **POSIXlt**
- The POSIXlt object contains some useful metadata.
- > names(unclass(p<- as.POSIXlt(x)))
- > p$sec; > p$hour; > p$gmtoff
- > p$wday; > p$zone
- **POSIXct**
- You can also use the POSIXct format.
- > x<- Sys.time()
- > x # Already in 'POSIXct' format.
- > unclass(x) # Internal representation
- > x$sec # Error in x$sec : $ operator is invalid for atomic vectors

Date And Time

- Finally, there is the strptime function in case your dates are written in a different format;
- > datestring <- c("January 10, 2012 10:40", "December 9, 2011 9:10")
- > print(x <- strptime(datestring, "%B %d, %Y %H:%M"))
- > class(x)
- **Operations on Dates and Times**
- You can use mathematical operations on dates and times. Well, really just + and -. You can do comparisons to (i.e == ,<=) .
- x <- as.Date("2012-01-01")
- y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
- > x-y
- Error in x - y : non-numeric argument to binary operator
- In addition: Warning message: I
- ncompatible methods ("-.Date", "-.POSIXt") for "-"
- > x<-as.POSIXlt(x)
- > x-y
- Time difference of 356.7261 days

Date And Time

- The nice thing about the date/time classes, is that they keep track of all the annoying things about dates and times, like leap years, leap seconds, daylight savings and time zones.
- **Example-1**
- Here' san example where a leap year gets involved.
- > x<-as.Date("2012-03-01")
- > y<-as.Date("2012-02-28")
- > x-y
- Time difference of 2 days
- **Example-2**
- Here's an example where two different time zones are in play (unless you live in GMT time zone, in which case they will be the same
- > x <- as.POSIXct("2012-10-25 01:00:00")
- > y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
- > y-x
- Time difference of 10 hours

Introduction To Data Science Using R Programming

Introduction to the dplyr R package

Introduction to the dplyr R package

- **dplyr**
- The data frame is a key data structure in statistics and in R.
- There is one observation per row.
- Each column represents a variable or measure or characteristics
- Primary implementation that you will use is the default R implementation
- Other implementation, particularly relational database system.
- Developed by Hadley Wickham or R-Studio.
- An optimized and distilled version of plyr package (also by Hadley)
- Does not provide any “new” functionality per se, but greatly simplifies existing functionality in R.
- Provides a “grammar” (in particular, verbs) for data manipulation.
- Is Very fast, as may key operations are coded in C++

Introduction to the dplyr R package

- **dplyr Grammar**
- Some of the key “verbs” provided by the dplyr package are;
- 1- *select*: return a subset of the columns of a data frame, using a flexible notation.
- 2- *filter*: extract a subset of rows from a data frame base on logical conditions.
- 3- *arrange:* reorder rows of a data frame.
- 4- *rename:* rename variable in a data frame.
- 5- *mutate:* add new variables/columns or transform existing variables.
- 6- *summarise /summarize:* generate summary statistics of different variable in the data frame, possibly within strata.
- 7- *%>%:* the “pipe” operator is used to connect multiple verb actions together into a pipeline.
- *There is also a handy print method that prevents you from printing a lot of data to the console.*

Introduction to the dplyr R package

- **Common dplyr Function Properties**
- The first argument is a data frame.
- The subsequent argument describe what to to with it, and you can refer to columns in the data frame directly without using $ operator (just use the names)
- The result is a new data frame.
- Data frames must be properly formatted and annotated for this to all be useful.

Introduction to the dplyr R package

- **Installing the dplyr package**
- > install.packages("dplyr",dependencies = TRUE)
- > library(dplyr)
- Error: package or namespace load failed for ‘dplyr’ in loadNamespace(j <- i[[1L]], c(lib.loc, .libPaths()), versionCheck = vI[[j]]): there is no package called ‘bindr’
- > install.packages("magrittr")
- > library(dplyr)
- Error: package or namespace load failed for ‘dplyr’ in loadNamespace(j <- i[[1L]], c(lib.loc, .libPaths()), versionCheck = vI[[j]]): there is no package called ‘bindr’
- > install.packages("bindr")
- > library(dplyr)

- Attaching package: ‘dplyr’
- The following objects are masked from ‘package:stats’:
- filter, lag
- The following objects are masked from ‘package:base’:
- intersect, setdiff, setequal, union

Introduction to the dplyr R package

- In this section we will be using a dataset containing air pollution and temperature data for the city Chicago in the U.S. The dataset is available on the following web site.
- http://www.biostat.jhsph.edu/~rpeng/leanpub/rprog/chicago_data.zip
- **select()**
- > getwd()
- > setwd("D:/MS DS/1st Semester/Business Analytics and Strategy/Data Sets")
- > getwd()
- # After unzipping the data, you can load the data into R using readRDS() function.
- > chicago<-readRDS("chicago.rds")
- # 6940 obs of 8 variables.
- **dim() function**
- > dim(chicago)
- **str() function**
- > str(chicago)

Introduction to the dplyr R package

- The select () function can be used to select columns of a data frame that you want to focus on. Often you'll have a large data frame containing "all" the data, but any given analysis might only use a subset of variables or observations. The select () function allows you to get the few columns you might need.
- **Example:**
- Suppose we wanted to take the first 3 columns only. There are few ways to do this.
- > names(chicago)
- **Method-1**
- We could for example use numerical indices.
- > names(chicago)[1:3]
- > print(subset<-select(chicago,1:3))
- > View(subset)
- **Method-2**
- But we can also use the names directly.
- > subset<-select(chicago,city:dptp)
- > View(subset)

Introduction to the dplyr R package

- **Method-3**
- You can also omit variables using the select () function by using the negative sign. With select() you can do;
- > subset<-select(chicago,-(city:dptp))
- Which indicates that we should include variable except the variable city through dptp.
- The equivalent code in base R would be;
- > i<-match("city",names(chicago))
- > j<-match("dptp",names(chicago))
- > head(chicago[,-(i:j)])
- Not super intuitive, right?
- The select () function also allows a special syntax that allows you to specify variable names based on patterns.
- **Method-4**
- If you wanted to keep variable that end with a “2”, we could do;
- > print(subset<-select(chicago,ends_with("2")))
- > str(subset)

Introduction to the dplyr R package

- **Method-5**
- If we wanted to keep every variable that starts with a “d”, we could do;
- > print(subset<-select(chicago,starts_with("d")))
- You can also use more general regular expressions if necessary. See the help page (?select) for more details.

Introduction to the dplyr R package

- **filter ()**
- The filter () function is used to extract subsets of rows from a data frame.
- This function is similar to the existing subset() function in R but is quit a bit faster.
- **Example-1**
- Suppose we wanted to extract the rows of the chicago data frame where the level of pm25tmean2 are greater than 30, we could do;
- > print(c.f<-filter(chicago,pm25tmean2>30))
- > str(c.f)
- You can see that there are now only 194 rows in the data frame and the distribution of the pm25tmean2 value is;
- > summary(c.f$pm25tmean2)
- **Example-2**
- We can place an arbitrarily complex logical sequence inside of filter(), so we could for example extract the row where pm25mean2 is greater than 30 and temperature is greater that 80 degrees Fahrenheit.
- > c.f<-filter(chicago,pm25tmean2>30 & tmpd>80)
- > print(subset<-select(c.f,date,tmpd,pm25tmean2))

Introduction to the dplyr R package

- **arrange ()**
- The arrange () function is used to reorder rows of a data frame according to one of the variable/columns.
- **Example-1**
- We can order the rows of the data frame by date, so that the first row is the earliest observation and the last row is the latest observation.
- > chicago<-arrange(chicago,date)
- We can now check the first few rows;
- > head(select(chicago,date,pm25tmean2),3)
- And the last few rows;
- > tail(select(chicago,date,pm25tmean2),3)
- **Example-2**
- Columns can be arranged in descending order too by using the special desc() operator.
- > chicago<-arrange(chicago,desc(date))
- Looking at the first three and last three rows shows the dates in descending order.
- > head(select(chicago,date,pm25tmean2),3)
- > tail(select(chicago,date,pm25tmean2),3)

Introduction to the dplyr R package

- **rename()**
- Renaming a variable in a data frame in R is surprisingly hard to do!. The rename() function is designed to make this process easier.
- Example
- You can see the names of the first five variables in the chicago data frame.
- > head(chicago[1:5],3)
- Or
- > head(chicago[,1:5],3)
- The dptp column is supposed to represent the dew point temperature and the pm25tmean2 column provides the PM2.5 data.
- > chicago<-rename(chicago,dewpoint=dptp,pm25=pm25tmean2)
- > head(chicago[,1:5],3)
- > tail(chicago[,1:5],3)
- The syntax inside the rename() function is to have the new name on the left-hand site of the =sign and the old name on the right-hand site.

Introduction to the dplyr R package

- **mutate()**

- The mutate() function exists to compute transformations of variables in a data frame.
- Often, you want to create new variables that are derived from existing variables and mutate() provides a clean interface for doing that.

- **Example:**
- With air pollution data, we often want to detrend that data by subtracting the mean form data. That way we can look at weather a given day's air pollution level is higher than or less than average.
- > chicago<-mutate(chicago,pm25detrend=pm25-mean(pm25,na.rm = TRUE))
- > head(chicago)
- > tail(chicago)

Introduction to the dplyr R package

- **transumte() function**
- There is also the related transmute() function, which does the same thing as mutate() but then drops all non-transformed variable.

- **Example**
- Here we detrend the PM10 and ozone (o3) variables.
- > chicago<-transmute(chicago,pm10detrend=pm10tmean2-mean(pm10tmean2,na.rm = TRUE))

Introduction to the dplyr R package

- **group_by ()**
- The group_by() function is used to generate summary statistics form the data frame within strata defined by a variable.
- Example:
- In this air pollution dataset, you might want to know what the average annual level of pm25tmean2.
- Step-1
- The stratum is the year, and that is something we can derive from the date variable. First, we can create a year variable using as.POSIXlt()
- > chicago<-mutate(chicago,year=as.POSIXlt(date)$year+1900)
- > head(chicago)
- > tail(chicago)
- Step-2
- Now we can create a separate data frame that splits the original data frame by year.
- > years<-group_by(chicago,year)
- Step-3
- We compute summary statistics for each year in the data frame with the summarize() function.
- > summarize(years,pm25tmean2=mean(pm25tmean2,na.rm = TRUE))

Introduction to the dplyr R package

- **%>%**
- The pipeline operator %>% is very hand for stringing together multiple dplyr functions in a sequence of operations.
- Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e.
- > third(second(first(x)))
- This nesting is not a natural way to think about a sequence of operations.
- The %>% operator allows you sting operations in a left-t0-right fashion. i.e.
- > first(x) %>% second %>% third
- Example-1
- There we had to;
- 1- create a new variable pm25.qunit
- 2- split the data frame by that new variable
- Compute the mean of o3 and no2

Introduction to the dplyr R package

- **Step-0**
- Rename the variable;
- > chicago<-rename(chicago,pm25=pm25tmean2)
- **Step-1**
- qq<-quantile(chicago$pm25,seq(0,1,0.2),na.rm = TRUE)
- **Step-2**
- >mutate(chicago,pm25.quint=cut(pm25,qq))%>%group_by(pm25.quint)%>%summarize(o3=mean(o3tmean2,na.rm=TRUE),no2=mean(no2tmean2,na.rm=TRUE))
- This way we don't have to create a set of temporary variables along the way or create a massive nested sequence of function calls.

Introduction to the dplyr R package

- Once you learn the dplyr “grammar” there are a few additional benfints;
- deplur can work with other data frame “backends”
- data.table for large fast tables.
- SQL interface for relational databases via the DBI package.

Introduction To Data Science Using R Programming

Control Structures

Control Structures

- Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions.
- Common structures are;
- 1- if, else: testing a condition
- 2- for: execute a loop a fixed number of times
- 3- while: execute a loop *while* a condition is true
- 4- repeat: execute an infinite loop
- 5- break: break the execution of a loop
- 6- next: skip an interation of a loop
- 7- return: exit a function

Control Structures

- **1- if, else: testing a condition**
- The if-else combination is probably the most commonly used control structure in R.
- This structure allows you to test a condition and act on it depending on whether it's true or false.

```
if(<condition>) {
        ## do something
} else {
        ## do something else
}
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

Control Structures

Example-1	Eaxample-2

#This is a valid if/else structure.
rm(x,y)
x<-8
if(x>3){
y<-10
} else {
y<-0
}
	#This is a vaild if/else structure.
	rm(x)
	x<-2
	if(x>3){
	10
	} else {
	0
	}

Example-3	**Example-4**

rm(x,y)
x <- 50
if(x > 3) {
print(y<-x+10)
} else{
print(y<-x/2)
}
	rm(x,y)
	x <- 8
	if(x >= 1 & x<=4) {
	print(y<-x+10)
	} else if (x>=5 & x<=8){
	print(y<-x/2)
	}

Control Structures

Example-5	Eaxample-6

rm(x,y)
x <- 9
if(x >= 1 & x<=4) {
print(y<-x+10)
} else if (x>=5 & x<=8){
print(y<-x/2)
} else {
print(y<-"I am Umer Saeed")
}
	rm(x,y,z)
	x <- 8
	if(x >= 1 & x<=4) {
	print(y<-x+10)
	}
	print(z<-"I am Umer Saeed")

Example-7	Comentes

#Generate a uniform random number.
rm(x)
x<-runif(1,0,10)
print(x)
if(x>3){
print(y<-10)
} else {
print(y<-0)
}
	Neither way of writing this expression is more correct than the other. Which one you will depend on your preference and perhaps those of the team you may be working with.
	Of course, the else clause is not necessary.

Control Structures

- You could have a series of if clauses that always get executed if their respective condition are true.

```
if(<condition1>) {

}

if(<condition2>) {

}
```

Control Structures

- **2- for: execute a loop a fixed number of times**
- For loops an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
rm(x)
x<-1:10
for(i in 1:10){
  print(x[i])
}
```

- This loop takes I variable and in each iteration of the loops it values 1,2,3…,10 and then exits.

Control Structures

Example-1

```
x<-c("a","b","c","d")
for (i in 1:4){
  print(x[i])
}
```

Example-2

```
x<-c("a","b","c","d")
for(i in seq_along(x)){
  print(x[i])
}
```

Exmple-3

```
x<-c("a","b","c","d")
for(letter in x){
  print(letter)
}
```

Example-4

```
x<-c("a","b","c","d")
for(i in 1:4) print(x[i])
```

- There four loops have the same behavior.
- However , I like to sue curly braces even for one-line loops, because that way if you decide to expand the loop to multiple lines, you won't be burned because you forget to add curly braces (and you will be burned by this)

Control Structures

- **Nested for loops**
- for loops can be nested.

```
rm(x)
x<-matrix(1:6,2,3)
for(i in seq_len(nrow(x))){
  for(j in seq_len(ncol(x))){
    print(x[i,j])
  }
}
```

- Be careful with nesting though. Nesting beyond 2-3 levels is often very difficult to read/understand.

Control Structures

- **while loop**
- While loops begin by testing a condition, if it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
count<-0
while(count<=10){
  print(count)
  count<-count+1
}
```

- While loops can potentially result in infinite loops if not written properly. Use with care!

Control Structures

- Sometimes there will be more than one condition is the test.

```
z<-5
while(z>=3 && z<=10){
  print(z)
  coin<-rbinom(1,1,0.5)
  if(coin==1){
    z<z+1
  } else{
    z<z-1
  }
}
```

- Conditions are always evaluated from left to right.

Control Structures

- **repeat**
- Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a repeat loop is to call break.

Example-1

```
i<-0
repeat{
  print(i)
  i<-i+1
  if(i>=5)
    break
}
```

Example-2

```
x<-1:5
for (val in x){
  if(val==3){
    break
  }
  print(val)
}
```

Control Structures

- **next**
- next is used to skip an iteration of a loop.

```
for(i in 1:5){
  if(i==2)
    next
  print(i)
}
```

- Control structures mentioned here are primarily useful for writing programs; for command line interactive work, the “*apply” functions are more useful.

Introduction To Data Science Using R Programming

Functions

Functions

- Functions are created using the function() directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.

```
f <- function(<arguments>) {
        ## Do something interesting
}
```

- Functions in R are “first class objects” which means that they can be treated like any other R objects. Importantly,
- 1- Functions can be passed as arguments to other function.
- 2- Functions can be nested, so that you can define a function inside of another function.
- 3- The return value of a function body to be evaluated.

Functions

- **Function Arguments**
- Functions have named arguments which potentially have default values.
- 1- The formal arguments are the arguments included in the function definition.
- 2- The formals function returns a list of all the formal arguments of a function.
- 3- Not every function call in R makes use of all the formal arguments.
- 4- Function arguments can be missing or might have default values.

Functions

- **Argument Matching**
- R functions arguments can be positionally or by name. So the following calls to sd are all equivalent.
- > mydata<-rnorm(100)
- > sd(mydata)
- > sd(x=mydata)
- > sd(x=mydata,na.rm = FALSE)
- > sd(na.rm = FALSE,x=mydata)
- > sd(na.rm = FALSE,mydata)
- Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

Functions

- **Argument Matching**
- You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out' of the argument list and the remaining arguments are mated in the order that they are listed in function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action,
          method = "qr", model = TRUE, x = FALSE,
          y = FALSE, qr = TRUE, singular.ok = TRUE,
          contrasts = NULL, offset, ...)
```

- The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Functions

- **Argument Matching**
- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the default for everything except for an argument near the end of the list.
- Named arguments also help if you can remember the name of the argument and not its position on the argument list.

- Function arguments can also be partially match, which is useful for interactive work. The order of operations when given an argument is;
- 1- Check for exact match for a named argument
- 2- Check for a partial match
- 3- Check for a position match

Functions

- **Lazy Evaluation**
- Arguments to functions are evaluated lazily, so they are evaluated only as needed.

Example-1

```
f<-function(a,b){
  a^b
}
f(2,2)
```

Example-2

```
f<-function(a,b){
  a^2
}
f(4)
```

- This function (example-2) never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

Functions

- **Lazy Evaluation**

Example-3

```
f<-function(a,b){
  print(a)
  print(b)
}
f(40,45)
```

Example-4

```
f<-function(a,b){
  print(a)
  print(b)
}
f(40)
```

Error in print(b) : argument "b" is missing, with no default

- Example-4: Notice that “45” got printed first before the error was triggered. This is because b did not have to be evaluated until after print(a). Once the function tried to evaluated print(b) if had to throw an error.

Functions

- **Lazy Evaluation**

Example-5

```
add2<-function(x,y){
x+y
}
add2(5,3)
```

Example-6

```
above10<-function(x){
use<-x>10
x[use]
}
>x<-c(5,10,15,20)
> above10(x)
```

Example-7

```
aboven<-function(x,n){
use<-x>n
x[use]
}
> x<-c(2,4,6,8,10,12,14,16)
> aboven(x,8)
```

Example-8

```
aboven10<-function(x,n=10){
use<-x>n
x[use]
}
> x<-c(2,4,6,8,10,12,14,16)
> aboven(x)
```

Functions

Example-9

```
rm(y)
cmean<-function(y,na.rm=TRUE){
  nc<-ncol(y)
  mean<-numeric(nc)
  for (i in 1:nc){
    mean[i]<-mean(y[,i],na.rm=TRUE)
  }
  mean
}
> cmean(airquality)
```

Example-10 (How to Calcuculate Mode)

```
modefun<-function(x){
        y<-table(x)
        names(y)[y==max(y)]
}
> x<-c(1,2,2,2,2,2,2,3)
> modefun(x)
```

Introduction To Data Science Using R Programming

Coding Standards for R

Coding Standards for R

- Coding standards are by no means universal and are often the subject of irrational flam wars on various languages-or project specific mailing list.
- Nevertheless, I will just give you the standards that I use and the rational behind them.
- 1- Always use text file/text editor
- 2- Indent your code
- 3- Limit the width of you code
- 4- Limit the length of individual functions

Coding Standards for R

- **1- Always use text file/text editor**
- Using text files/text editor is fundamental to coding.
- If you're writing you code in an editor like Microsoft Word, you need to stop.
- Interactive development environments like Rstudio have nice editor built in, but there are many other out there.

Coding Standards for R

- **2- Indent your code**
- Indenting is very important for the readability of your code.
- Some programming languages actually require it as part of their syntax, but R does not.
- Indenting is very important, however how much you should indent is up for debate, but I think each indent should be a minimum of 4 spaces and ideally it should be 8 spaces.

Coding Standards for R

- **3- Limit the width of you code**
- I like to limit the width of my text editor so that the code I write doesn't fly off into the wilderness on the firth hand site.
- This limitation, along with the 8 space indentation, forces you to write code that is clean, readable and naturally broken down into modular unites.
- In particular, this combination limits your ability to write very long function with may different level of nesting.

Coding Standards for R

- **4- Limit the length of individual functions**
- If you are wiring functions, it's usually a good idea to not let your functions run for pages and pages.
- Typically, purpose of a function is to execute on activity or idea.
- If your function is doing lots of things, it probably needs to be broken into multiple functions.
- Function should not take up more than one page of your editor (of course, this depends on the size of your monitor).

Introduction To Data Science Using R Programming

Loop Functions

Loop Functions

- **Looping on the command Line:**
- Writing for, while loops is useful when programing but not particularly easy when working interactively on the command line.
- There are some functions which implement looping to make life easier.
- 1- *lapply*: Loop over a list and evaluate a function on each element.
- 2- *sapply*: Same as lapply but try to simplify the result.
- 3- *tapply*: Apply a function over subsets of a vector.
- 4- *apply*: Apply a function over the margins of an array.
- 5- *mapply:* Multivariate version of lapply.
- An auxiliary function *split* is also useful, particularly in conjunction with *lapply*.

Loop Functions

- **1- *lapply()*:**
- The lapply () function does the following simple series of operations:
- 1- it loops over a list, iterating over each element in the list.
- 2- it applies a function to each of the llist(a function that you specify)
- 3- and returns a list (the 1 is for “list”)

- This function take three arguments (1) a list X (2) a function (or the name of a function) FUN; (3) Other arguments via its….arguments.
- It X is not a list, it will be coerced to a list using as.list().
- It’s important to rember that lapply () always returns a list, regardless of the class of the input.
- The body of the lapply () function can be seen here.

```
> lapply
function (X, FUN, ...)
{
    FUN <- match.fun(FUN)
    if (!is.vector(X) || is.object(X))
        X <- as.list(X)
    .Internal(lapply(X, FUN))
}
<bytecode: 0x0000000007f1e030>
<environment: namespace:base>
>
```

- The actual looping is done internally in C code.

Loop Functions

- **Example-1**
- Applying the mean() function to all elements of a list. If the original list has names, the names will be preserved in the output.
- > x<-list(a=1:5,b=rnorm(10))
- > lapply(x,mean)
- or
- > lapply(x<-list(a=1:5,b=rnorm(10)),mean)
- **Example-2**
- > x<-list(a=1:4,b=rnorm(10),c=rnorm(20,1),d=rnorm(100,5))
- > lapply(x,mean)
- or
- > lapply(x<-list(a=1:4,b=rnorm(10),c=rnorm(20,1),d=rnorm(100,5)),mean)

Loop Functions

- **Example-3**
- You can use lapply() to evaluate a function multiple times each with different argument. Below is an example where I call the runif() function four times, each time generating a different number of random numbers.
- > lapply(x<-1:4,runif)
- **Example-4**
- When you pass a function to lapply(), lapply() takes elements of the list and passes them as the first argument of the function you are applying. In the above example, the first argument of runif() is n, and so the elements of the sequence 1:4 all got passed to the n argument of runif().
- Functions that you pass to lapply() may have other arguments. For example, the runif() function has a min and max argument too. In example-3 I used the default values for min and max. How would you be able to specify different values for that is the context of lapply()?
- Here is where the … argument to lappy() comes into play. Any arguments that you place in the … argument will be passed down to the function being applied to the elements of the list.
- Here the min=0 and max=10 arguments are passed down to runif() every time it gets called.
- > lapply(x<-1:4,runif,min=0,max=10.0)

Loop Functions

- The lapply() function and its friends make heavy use of anonymous functions. These functions are generated “on the fly” as you are using lapply() . Once the call to lapply () is finished, the function disappears and does not appear in the workspace.
- Here I am creating a list that contains tow matrices.
- > x<-list(a=matrix(1:4),b=matrix(1:6,2,3))
- Suppose I wanted to extract the first column of each matrix in the list. I could write an anonymous function for extracting the first column of each matrix,
- **Method-1**
- > lapply(x,function(elt){elt[,1]})
- **or**
- > lapply(x<-list(a=matrix(1:4),b=matrix(1:6,2,3)),function(elt){elt[,1]})
- Notice that I put the function() definition right in the call to lapply().
- This is perfectly legal and acceptable.
- You can put an arbitrarily complicated function definition inside lapply(), but if it’s going to be more complicated, it’s probably a better idea to define the function separately.

Loop Functions

- **Method-2**

```
f<-function(elt){
        elt[,1]

}
```

- > lapply(x,f)

- Now the function is no longer anonymous; it's name is f.
- Whether you use an anonymous function or you define a function first depends on your context.

Loop Functions

- **2- *sapply*:**
- The sapply() function behaves similarly to lappy(); the only real difference is in the returen values.
- sapply() will try to simplify the result of lapply() if possible.
- Essentially, sapply() calls lapply() on its input and then applies the following algorithm:
- 1- If the result is a list where every element is length 1, then a vector is returned.
- 2- if the result is a list where every element is a vector of the same length(>1), a matrix is retuned.
- 3- IF it can't figure things out, a list is returned.
- > set.seed(1:100)
- > sapply(x<-list(a=1:4,b=rnorm(10),c=rnorm(20,1),d=rnorm(100,5)),mean)
- > set.seed(1:100)
- > lapply(x<-list(a=1:4,b=rnorm(10),c=rnorm(20,1),d=rnorm(100,5)),mean)
- Notice the lapply() returns a list , but each element of the list has length 1, sapply() collapsed the output into a numeric vector, which is often more useful than a list.

Loop Functions

- **An auxiliary function *split* ()is also useful, particularly in conjunction with *lapply*.**
- Split takes a vector or other objects and splits it into groups determined by a factor or list of factor.

```
> str(split)
function (x, f, drop = FALSE, ...)
```

- x is a vector (or list) or data frame.
- f is a factor (or coerced to one) or list of factors.
- Drops indicates whether empty factor levels should be dropped.

- The combination of split () and a function like lapply() or sapply() is a common paradigm in R.
- The basic idea is that you can take a data structure, split it into subsets defined by another variable, and apply a function over those subsets.

Loop Functions

- **Example**
- We simulate some data and split it according to a factor variable.
- > set.seed(1:10)
- > x<-c(rnorm(10),runif(10),rnorm(10,1))
- > f<-gl(3,10)
- > split(x,f)
- > set.seed(1:10)
- > lapply(split(x,f),mean)
- > sapply(split(x,f),mean)

Loop Functions

- **Splitting a Data Frame**
- > airquality # R Data Set
- > head(airquality)
- > tail(airquality)
- # We can split the airquality data frame by Month variable so that we have separate sub-data frames fro each month.
- > s<-split(airquality,airquality$Month)
- > str(s)
- # Then we can take the column means for “ozone”,”Solar.R” and “Wind” for each sub-data frame.
- > lapply(s,colMeans)
- > lapply(s,colMeans,na.rm=TRUE)
- > sapply(s,colMeans)
- > sapply(s,colMeans,na.rm=TRUE)
- > lapply(s,function(x){colMeans(x[,c("Ozone","Solar.R","Wind")])})
- > lapply(s,function(x){colMeans(x[,c("Ozone","Solar.R","Wind")],na.rm=TRUE)})
- > sapply(s,function(x){colMeans(x[,c("Ozone","Solar.R","Wind")])})
- > sapply(s,function(x){colMeans(x[,c("Ozone","Solar.R","Wind")],na.rm=TRUE)})

Loop Functions

- **interaction ()**
- Occasionally, we may want to split an R object according to levels defined in more than one variable. We can do this by creating an interaction of the variables with interaction () function.
- Example-1
- > set.seed(1:10)
- > x<-rnorm(10)
- > f1<-gl(2,5)
- > f2<-gl(5,2)
- # Create interaction of two factors
- > interaction(f1,f2)
- > str(split(x,list(f1,f2)))
- # Notices that there are 4 categories with no data. But we can drop empty levels when we call the split() function.
- > str(split(x,list(f1,f2),drop=TRUE))

Loop Functions

- **3- *tapply*:**
- Function tapply() is used to apply a function over subsets of a vector.
- If can be thought of as a combination of split() and sapply() for vector only.
- “t” in tapply() refers to “table” but that is unconfirmed.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- The arguments to tapply() are as follows;
- X is a vector.
- INDEX is a factor or a list of factors (or else they are coerced to factors)
- …contains other argument to be passed FUN
- Simplify, should we simplify the result?

Loop Functions

- **Example-1**
- Given a vector of numbers, one simple operation is to take group means.
- # Simulate some data
- > set.seed(1:10)
- > x<-c(rnorm(10),runif(10),rnorm(10,1))
- # Define some groups with a factor variable
- > f<-gl(3,10)
- > tapply(x,f,mean)
- We can also take the group means without simplifying the result, which will give us a list. For functions that return a single value, usually, this in not what we want, but it can be done.
- > tapply(x,f,mean,simplify = FALSE)
- # Find group ranges.
- > tapply(x,f,range)

Loop Functions

- **4- *apply*:**
- The apply() function is use to a evaluate a function (often an anonymous one) over the margins of an array.
- It is most often used to apply a function to the rows or columns of a matrix.
- It can be used with general arrays, e.g. taking the average of an array matrices.
- It is not really faster than wiring a loop, but it works in on line!

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- X is an array.
- MARGIN is an integer vector indicating which margins should be “retained”
- FUN is a function to be applies
- … is for other arguments to be passed to FUN

Loop Functions

- **Example-1**
- Here I create a 20 by 10 matrix of Normal random number.
- Then compute the mean of each column.
- > set.seed(0:1)
- > x<-matrix(rnorm(200),20,10)
- > apply(x,2,mean) #c(1)-for row calculation, c(2)-for column calculation
- **Example-2**
- Here I create a 20 by 10 matrix of Normal random number.
- Then compute the sum of each row.
- > set.seed(0:1)
- > x<-matrix(rnorm(200),20,10)
- > apply(x,1,sum) #c(1)-for row calculation, c(2)-for column calculation
- Possible functions used in apply include mean, sd, var, min, max, median, range, and quantile.

Loop Functions

- **Col/Row Sums and Means**
- For special case for column/row sums and column/row means of matrices, we have some useful shortcuts.
- rowSums=apply(x,1,sum)
- rowMeans=apply(x,1,mean)
- colSums=apply(x,2,sum)
- colMeans=apply(x,2,mean)
- The shortcut functions are heavily optimized and hence are much faster, but you probably won't notice unless you're using a large matrix.
- Another nice aspect of these functions is that they are a bit more descriptive.

Loop Functions

- **Example**
- You can do more than take sums and means with the apply function. For example, you can compute quantiles of the rows a matrix using quantile() function.
- > set.seed(1:10)
- > x<-matrix(rnorm(200),20,10)
- > apply(x,1,quantile,probs=c(0.25,0.75))

Loop Functions

- **5- *mapply:***
- The mapply() function is a multivariate apply of sorts which applies a function in parallel over a set of arguments.
- Recall the lapply() and friends only iterate over a single R object.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
          USE.NAMES = TRUE)
```

- The arguments to mapply() are;
- FUN is a function to apply
- … contains R objects to apply over
- MoreArgs is a list of other arguments to FUN
- SIMPLIFY indicates whether the result should be simplfied.

Loop Functions

- Example
- The following is tedious to type;
- a<-list(rep(1,4),rep(2,3),rep(3,2),rep(4,1))
- With mapply(), instead we can do;
- > mapply(rep,1:4,4:1)

Introduction To Data Science Using R Programming

Simulation

Simulation

- **Generating Random Numbers**
- Function for probability distribution in R;
- 1-**rnorm**: generate random Normal variates with a given mean and standard devation.
- 2- **dnorm**: evaluate the Normal probability density (with a give mean/SD) at a point (or vector of points).
- 3- **pnorm**: evaluate the cumulative distribution function for Normal distribution.
- 4- **rpois**: generate random Poisson variates with a given rate.
- Probability distribution functions usually have four functions associated with them. The functions are prefixed with a;
- 1- **d** for density.
- 2- **r** for random number generation.
- 3- **p** for cumulative distribution.
- 4- **q** for quantile function.

Simulation

- Working with the Normal distributions requires using these four functions;

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

- **Example-1**
- We simulate standard Normal random numbers with mean 0 and standard deviation 1.
- > x<-rnorm(10)
- **Example-2**
- We can modify the default parameters to simulate number with mean 20 and standard deviation 2.
- > x<-rnorm(10,20,2)
- > summary(x)

Simulation

- **Setting the random number seed**
- When simulating and random numbers it is essential to set the *random number seed*.
- Setting the random number seed with set.seed() ensures reproducibility of the sequence of random numbers.
- **Example:**
- Generate 5 Normal random numbers with rnorm().

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
> # Notice that if i call rnorm() again i will of course get a different set 5 random numbers.
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
> # If i want to reproduce the original set of random numbers, i can just rest the seed with set.seed().
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

- In general, you should always set the random number seed when conduction a simulation.
- Otherwise, you will not be able to reconstruct the exact numbers that you produced in an analysis.

Simulation

- **Generating Poisson Data**

```
> x<-rpois(10,1)
> mean(x)
[1] 1.2
> y<-rpois(10,2)
> mean(y)
[1] 2
> z<-rpois(20,20)
> mean(z)
[1] 20.45
```

```
> print(a<-ppois(2,2))                    # Cumulative distribution [pr(x<=2)]
[1] 0.6766764
> print(b<-ppois(4,2))                    # Cumulative distribution [pr(x<=4)]
[1] 0.947347
> print(b<-ppois(6,2))                    # Cumulative distribution [pr(x<=6)]
[1] 0.9954662
```

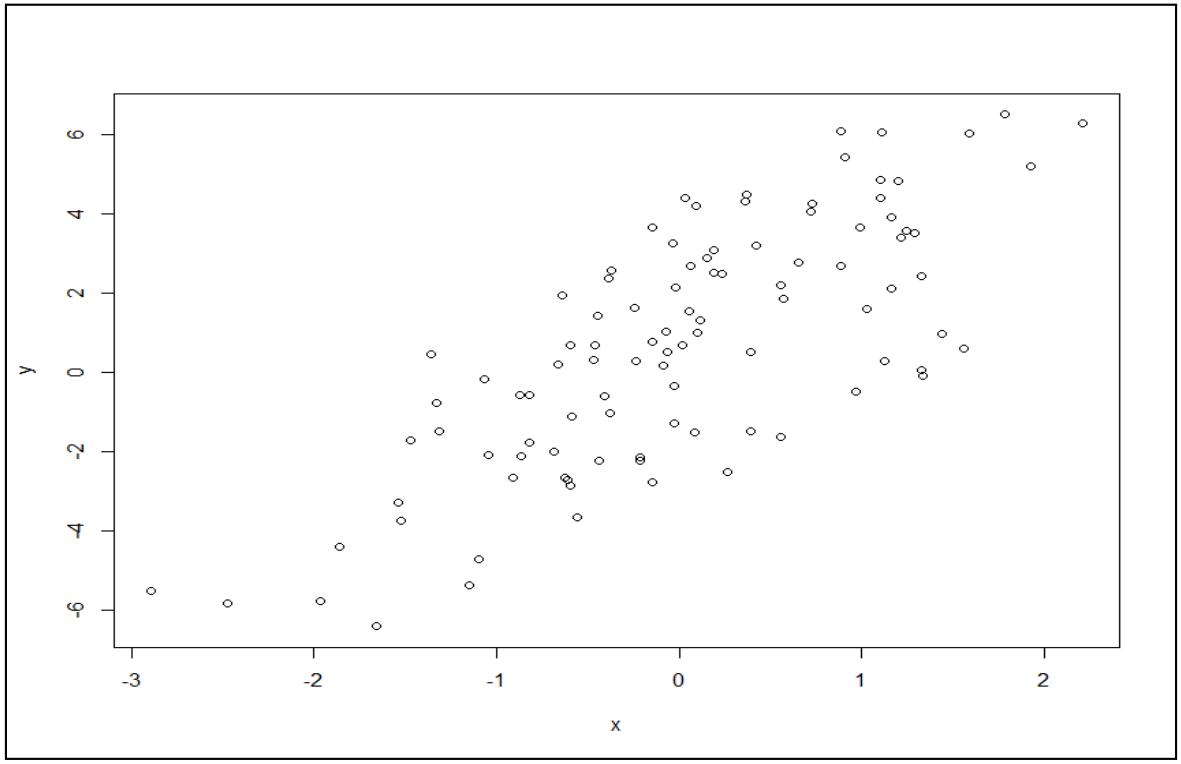
Simulation

- Suppose we want to simulate from the following model;

$$y = \beta_0 + \beta_1 x + \varepsilon$$

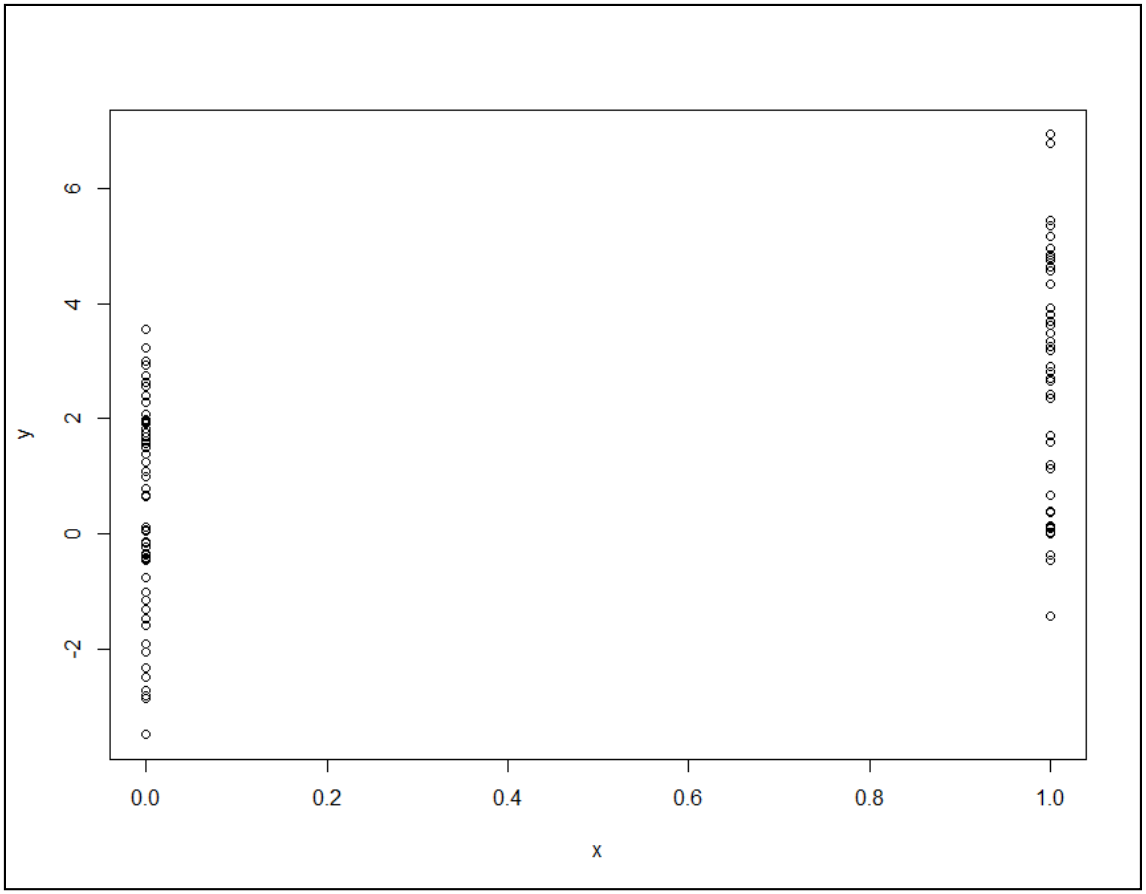
where $\varepsilon \sim \mathcal{N}(0, 2^2)$. Assume $x \sim \mathcal{N}(0, 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

- > set.seed(20)
- > x<-rnorm(100)
- > e<-rnorm(100,0,2)
- > y=0.5+2*x+e
- > summary(y)
- > plot(x,y)



Simulation

- What if x is binary?
- > set.seed(10)
- > x<-rbinom(100,1,0.5)
- > e<-rnorm(100,0,2)
- > y<-0.5+2*x+e
- > summary(y)
- > plot(x,y)



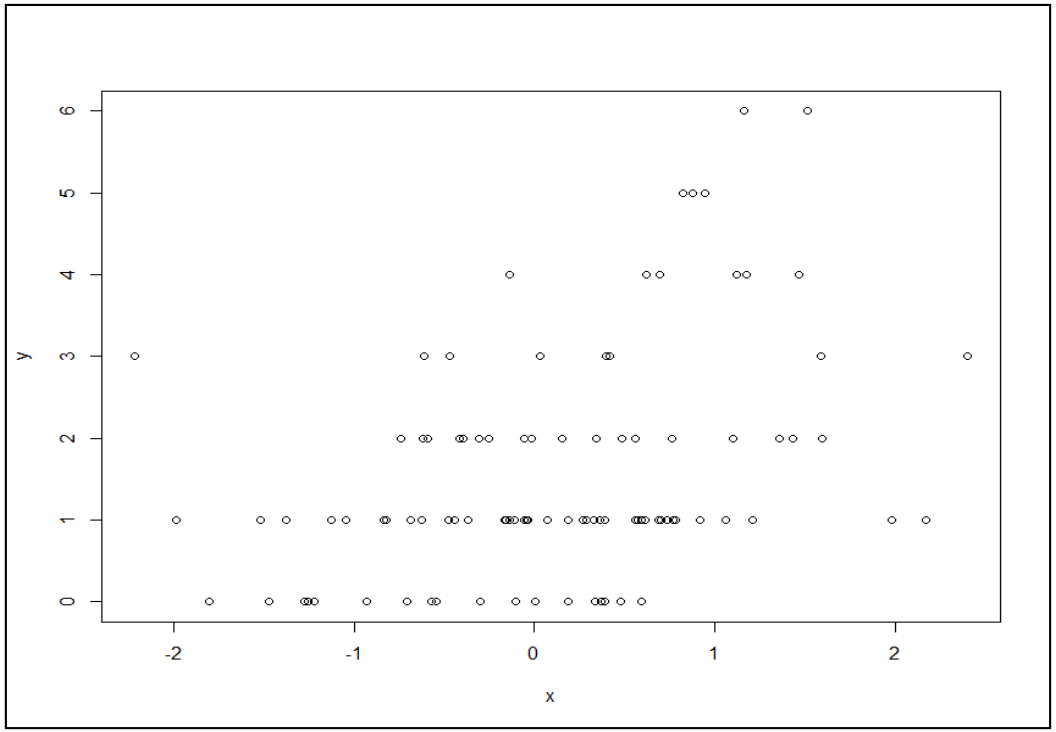
Simulation

- Suppose we want to simulate from a Poisson model where;
- Y~Poisson(μ)

$\log \mu = \beta_0 + \beta_1 x$

and $\beta_0 = 0.5$ and $\beta_1 = 0.3$. We need to use the `rpois` function for this

- > set.seed(1)
- > x<-rnorm(100)
- > log.mu<-0.5+0.3*x
- y<-rpois(100,exp(log.mu))
- > summary(y)
- > plot(x,y)



Simulation

- **Random Sampling**
- The sample() function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions numbers.
- > sample(1:10,4)
- > sample(1:10,4)
- > sample(letters,5)
- > sample (1:10) # Do a random permutation
- > sample (1:10) # Do a random permutation
- > sample (1:10, replace=TRUE) # Sample with replacement

Simulation

- To sample more complicated things, such as rows from a data frame or a list, you can sample indices into an object rather than the elements of the object itself.
- > data("airquality")
- > head(airquality)
- > tail(airquality)
- Now we just need to create the index vector indexing the row of the data frame and sample directly from the index vector.
- > ind<-seq_len(nrow(airquality))
- Sample from the index vector
- > samp<-sample(ind,10)
- > airquality[samp,]

Introduction To Data Science Using R Programming

Debugging

Debugging

- **Something's Wrong!**
- R has a number of ways to indicate to you that something's not right.
- 1- *message*: A generic notification/diagnostic message produced by the message() function; execution of the function continues.
- 2- *warning*: An indication that something is wrong but not necessarily fatal; execution of the function continues. Warnings are generated by warning () function.
- 3- *error*: An indication that a fatal problem has occurred and execution of the function stops. Errors are produced by the stop() function.
- 4- *condition*: A generic concept for indicating that something unexpected bas occurred; programmers can create their own custom conditions if they want.

Debugging

- **2- *warning***
- Here is an example of a warning that you might receive in the course of using R.
- **Example-1**
- > log(-4)
- [1] NaN
- Warning message: In log(-4) : NaNs produced
- This warning lets you know that taking the log of a negative number result in NaN value because you can't take the log of negative numbers.
- R doesn't give an error, because it has a useful value that it can return, the NaN value.
- The warning is just there to let you know that something unexpected happen.

Debugging

- **Example-2**
- Here is another function that is designed to print a message to console depending on the nature of its input.

```
printmessage<-function(x){
        if(x>0)
                print("x is > Zero")
        else
                print("x is < zero")
        invisible(x)
}
```

- This function is simple- it prints a message telling you whether x is greater than zero or less than zero.
- It also returns its input *invisibly*, which is a common practice with “print” function. Returning an object invisibly means that the return values does not get auto-printed when the function is called.
- Take a hard look at the function above and see if you can identify any bugs or problems.
- We can execute the function as follows.
- > printmessage(1) [1]
- > printmessage(1) [-1]

Debugging

- The function seems to work fine at this point. No errors, warnings or messages.
- > printmessage(NA)
- Error in if (x > 0) print("x is > Zero") else print("x is < zero") : missing value where TRUE/FALSE needed

```
printmessage<-function(x){
        if(is.na(x) | is.nan(x))
                print("x is missing")
        else if(x==0)
                print("x is zero")
        else if (x>=1)
                print("x is > Zero")
        else
                print("x is < zero")
        invisible(x)
}
```

- And all is fine. Now what about the following situation.
- > printmessage(log(c(-1,2)))

```
Warning messages:
1: In log(c(-1, 2)) : NaNs produced
2: In if (is.na(x) | is.nan(x)) print("x is missing") else if (x ==  :
  the condition has length > 1 and only the first element will be used
```

Debugging

- The problem here is the I passed printmessage() vector x that was of length 2 rather then length 1.
- Inside the body of printmessage() the expression is.na(x) returns vector that tested in the if statement.
- However, if cannot take vector arguments so you get a warning.
- The fundamental problem here is that printmessage() is not vectorized.
- We can solve this problem two ways.

Debugging

Method-1

One is by simple not allowing vector arguments.

```
printmessage<-function(x){
        if(length(x)>1L)
                stop("x has length >1")
        if(is.na(x) | is.nan(x))
                print("x is missing")
        else if(x==0)
                print("x is zero")
        else if (x>=1)
                print("x is > Zero")
        else
                print("x is < zero")
        invisible(x)
}
```

```
> printmessage(1:2)
Error in printmessage(1:2) : x has length >1
> printmessage(c(log(-1,2)))
[1] "x is missing"
Warning message:
In printmessage(c(log(-1, 2))) : NaNs produced
```

Debugging

- **Method-2**
- The other way is to vectorize the printmessage() function to allow it to take vector arguments.
- Vectorizing the function can be accomplished easily with the vectorize() function.
- > a<-Vectorize(printmessage)
- > out<-a(c(-1,2))

Debugging

- **Figuring Out What's Wrong:**
- How do you know that something is wrong with your function?
- **1-** What was your input? How did you call the function?
- **2-** What were you expecting? Output, messages, other results?
- **3-** What did you get?
- **4-** How does what you get differ from what you were expecting?
- **5-** Were your expectation correct in the first place?
- **6-** Can you reproduce the problem(exactly)?

Debugging

- **Debugging Tools in R:**
- R Provides a number of tools to help you with debugging your code. The primary tools for debugging functions in R are;
- 1- traceback(): prints out the function call stack after an error occurs; does nothing if there's no error.
- 2- debug(): flags a function for "debug" mode which allows you to step through execution of a function one line at a time
- 3- browser(): suspends the execution of a function wherever it is called and puts the function in debug mod.
- 4- trace(): allows you to insert debugging code into a function a specific place.
- 5- recover(): allows you to modify the error behavior so that you can browse the fucntion call stack.

Debugging

- **1- traceback():**
- The traceback () function prints out the function call stack after an error has occurred.
- The function call stack is the sequence of function that was called before the error occurred.
- For example, you may have a function a() which subsequently call function b() which calls c() and then d(). If an error occurs, it may not immediately clear in which function the error occurred.
- The raceback() function shows you how many levels deep you were when the error occurred.

```
> mean(x)
Error in mean(x) : object 'x' not found
> traceback()
1: mean(x)
```

- Here, it's clear that the error occurred inside the mean() function because the object x does not exit.
- The traceback() function must be called immediately after an error occurs. Once another function is called, you lose the traceback.

Debugging

- Here is slightly more complicated example using the lm() function for linear modeling.

```
> lm(y~x)
Error in eval(predvars, data, env) : object 'y' not found
> traceback()
7: eval(predvars, data, env)
6: eval(predvars, data, env)
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: stats::model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(mf, parent.frame())
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

- You can see now that the error did not get thrown until the 7th level of the function call stack, in which case the eval() function tried to evaluate the formula y~x and realized the object y dit not exit.
- Looking at the traceback is useful for figuring out roughly where an error occurred but it's not useful for more detailed debugging.

Debugging

- **Using debug()**
- The debug() function initiates an interactive debugger (also know as the “brower” in R) for a function.
- With the dubugger, you can step through an R function one expression at a time to pinpoint exactly where an error occurs.
- The debug() function takes a function as its first argument.
- Here is an example of debugging the lm() function.

```
> debug(lm)
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    ...
    if (!qr)
        z$qr <- NULL
    z
}
Browse[2]>
```

Debugging

- Now, every time you call the lm() function it will launch the interactive debugger.
- To turn this behavior of you need to call the undebug() function.
- The debugger calls the browser at the very top level of the function body.
- From ther you can step through each expression in the body.
- There are few special commands you can call in the browser:
- 1- n executes the current expression and moves to the next expression.
- 2- c continues execution of the function and does not stop until either an error of the function exits.
- Q quits the browser.

Debugging

- Here's an example of a browser session with the lm() function.

```
Browse[2]> n
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n
debug: m <- match(c("formula", "data", "subset", "weights", "na.action",
    "offset"), names(mf), 0L)
```

Debugging

- **Using recover()**
- The recover() function can be used to modify the error behavior or R when an error occurs.
- Normally, when an error occurs in a function, R will print out an error message, exit out of the function, and return you to your workspace to await further commands.
- With revover() you can tell R that when an error occurs, it should halt execution at the exact point at which the error occurred.
- That can give you the opportunity to poke around in the environment in which error occurred.
- This can be useful to see if there are any R objects or data that have been corrupted or mistakenly modified.

```
> options(error = recover)
> read.csv("nosuchfile")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nosuchfile': No such file or directory

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote = quote, dec = dec, fill = fill,
3: file(file, "rt")

Selection:
```

Debugging

- The recover() function will first print out the function call stack when an error occurs. Then, you can choose to jump around the call stack and investigate the problem.
- When you choose a frame number, you will be put in the browser (just like the interactive debugger triggered with debug()) and will have the ability to poke around.

Introduction To Data Science Using R Programming

Scoping Rules

Scoping Rules

- **A Diversion on Binding Values to Symbol**
- How does R knows which value to assign to which symbol? When I type;
- > lm<-function(x){x*x}
- > lm
- function(x){x*x}
- How does R know what value to assign to the symbol lm?
- Why doesn't it give if the value of lm that is in the stats packages?

- When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value.
- When you are working on the command line and need to retrieve the value of an R object, the order is roughly;
- 1- Search the global environment for a symbol name matching the one requested.
- 2- Search the namespaces of each of the packages on the search list.

Scoping Rules

- **Binding Values to Symbol**
- The search list can be found by using search function.

```
> search()
 [1] ".GlobalEnv"        "package:dplyr"     "tools:rstudio"     "package:stats"     "package:graphics"
 [6] "package:grDevices" "package:utils"     "package:datasets"  "package:methods"   "Autoloads"
[11] "package:base"
```

- The global environment or the user's workspace is always the first element of the search list and the bas package is always the list.
- For better or for worse, the order of the packages on the search list matter, particularly if there are multiple objects with the same name in different packages.
- Users can configure which packages get loaded on startup so if you are writing a function (or a package), you cannot assume that there will be a set list of packages available in a given order. When a user load a package library() the namespace of the that package get put in position2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespace for functions and non-functions so it's possible to have an object named c and a function named c().

Scoping Rules

- The scoping rules for R are the main feature that make it different from the original S languages.
- 1- The scoping rules determine how a value is associated with a free variable in a function.
- 2- R uses lexical scoping or static scoping. A common alternative is dynamic scoping.
- 3- Related to the scoping rules is how R uses the search list to bind a value to a symbol.
- 4- Lexical scoping turns out to be particularly useful for simplifying statistical computations.
- **Lexical Scoping**:

```
f<-function(x,y){
      x^2+y/z
}
```

- This function has 2 formal arguments x and y.
- In the body of the function there is another symbol z.
- In this case z is call a free variable.
- The scoping rues of a language determine how values are assigned to free variables.
- Free variables are not formal arguments and are not local variables (assigned inside the function body.

Scoping Rules

- Lexical Scoping in R means that the values of free variables are searched for in the environment in which the function was defined.

- *What is an environment ?*
- An environment is a collection of (symbol, value) pairs i.e x is a symbol and 3.14 might be it value.
- Every environment has a parent environment ; it is possible for an environment to have multiple “children”.
- The only environment without a parent is the empty environment .
- A function + an environment= a closure or function closure.

Scoping Rules

- Searching for the value for a free variable;
- If the value of a symbol is not fund in the environment in which a function was defined, then the search is continued in the parent environment.
- The search continue down the sequence of parent environment until we hit the top-level environment; this usually the global environment(workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the empty environment. If a value for a give symbol can not found the empty environment is arrived at, then an error is thrown.

Scoping Rules

- **Lexical Scoping: Why does all this matter?**
- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace.
- This behavior is local for most people and is usually the "right thing" to do.
- However, in R you can have functions define inside other function (Languages like C don't let you do this)
- Now things get interesting- in this case the environment in which a functions defined is the body of another function!

Scoping Rules

```
make.power<-function(n){
        pow<- function(x){
                x^n
        }
        pow
}
```

- The make. Power() is a kind of “constructor function” that can be used to construct other function.
- This function returns another function as its value

```
> cube(3)
[1] 27
> square(3)
[1] 9
> cube
function(x){
                x^n
        }
<environment: 0x0000000019728630>
> square
function(x){
                x^n
        }
<bytecode: 0x00000000182c6a38>
<environment: 0x0000000018442d48>
```

Scoping Rules

- **Exploring a Function Closure**
- What's in a functions environment?

```
> ls(environment(cube))
[1] "n"   "pow"
> get("n",environment(cube))
[1] 3
> ls(environment(square))
[1] "n"   "pow"
> get("n",environment(square))
[1] 2
```

Scoping Rules

- **Lexical Vs Dynamic Scoping**

```
y<-10
f<-function(x){
      y<-2
      y^2+g(x)
}
g<-function(x){
      x*y
}
f(3)
```

- With lexical scoping the value of y in the function g is looked up in the environment in which the function was defined, in this case the global environment, so the value of y is 10.
- With dynamic scoping, the value of y is looked up in the environment from which the function was called (sometimes referred to as the calling environment)
- In R the calling environment is know as the parent frame.
- So the value of y would be 2.

Scoping Rules

- When a function is defined in the global environment and is subsequently called from the global environment , then defining environment and the calling environment are the same. This can sometimes give the appearance od dynamic scoping.

```
g<-function(x){
        a<-3
        x+a+y
}
g(2)
Error in g(2) : object 'y' not found
y<-3
g(2)
```

Scoping Rules

- **Other Languages**
- Other languages that support the lexical scoping;
- 1- Scheme
- 2- Perl
- 3- Python
- 4- Common Lisp (all languages converge to Lisp)

Scoping Rules

- **Consequences of Lexical Scoping**
- In R, all objects must be stored in memory.
- All functions must carry a pointer to their respective defining environments, which could be anywhere.
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored ion the disk because the “defining environments” of all functions in the same.

Introduction To Data Science Using R Programming

Profiling R Code

Profiling R Code

- **Why is My code So Slow?**
- Profiling is a systematic way to examine how much time is spend in different parts of a program.
- Often code runs fine once, but what if you have to put it in a loop for 1,000 iterations? Is it still fast enough?
- Profiling is better than guessing.
- **On Optimizing Your Code**
- Getting biggest impact on speeding up code depends on knowing where the code spends most of its time.
- This cannot be done without performance analysis or profiling.
- *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."* –Donald Knuth

Profiling R Code

- **General Principles of Optimization**
- The basic principles of optimizing your code are;
- 1- Design first, then optimize.
- 2- Remember: Premature optimization is the root of all evil.
- 3- Measure (collector data), don't guess.
- 4- If you're to be scientist, you need to apply the same principles here!

Profiling R Code

- **Using system.time ()**
- Takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression.
- Computer the time (in second) need to execute an expression. If there's an error, gives time until the error occurred.
- Return an object of class proc_time;
- 1- user time: time charged to the CPU(s) for this expression.
- 2- elapsed time: "wall clock" time
- Usually, the user time and elapsed time are relatively close, for straight computing tasks.
- Elapsed time may be greater than user time it the CPU spends a lot of time waiting around.
- Elapsed time may be smaller than the user time if your machine has multiple cores/processors (and is capable of using the)
- - Muti-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL)
- - Parallel processing via the parallel package

Profiling R Code

- # Elapsed time> user time
- > system.time(readLines("http://www.jhsph.edu"))

```
 user  system elapsed
 0.55    0.00    6.67
```

Profiling R Code

- **The R Profiler**
- The Rprof() function starts the profiler in R.
- R must be compiled with profiler support (but this usually the case)
- The summaryRprof() function summarizes the output from Rprof() (otherwise it's not readable)
- DO NOT use system.time() and Rprof () together or you will be sad.
- Rprof() keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spend in each function.
- Default sampling interval is 0.02 sec.
- Note: If your code runs very quickly, the profiler is not useful, but then you probably don't need it in that case.

Profiling R Code

- **Using summary Rprof()**
- The summaryRprof() function tabulates the R profiler output and calculate how much time is spend in which function.
- There are tow method for normalizing the data.
- 1- “by.total” divides the time spend in each function by the total run time.
- 2- “by.sel” does the same as “by.total” but first subtracts out time spend in functions above the current functions in the call stack.

The grep() function

- **The grep () Function:**
- When called it's default arguments, the grep() returns the indices for strings in a character vector that match a given regular expression.
- When the values argument is specified as TRUE, grep() returns the strings themselves for the character vector.
- **colours() Function:**
- List of colours recognised by R, structured as character vector.
- **Example:**
- > grep("red",colours())
- > colours()[504:507]
- > grep("red",colours(),value=TRUE)

The attach() & detach() function

- > data()
- > head(infert)
- > names(infert)
- > education
- Error: object 'education' not found
- > attached (infert) #I can use the command attach() for infert to make all the objects visible.
- > education
- > detach() # detach all the objects of infert
- > education
- Error: object 'education' not found
- > save(infert,file = “he.rda") # Make a copy of Infert File
- >load("D:/MS DS/1st Semester/Business Analytics and Strategy/Data Sets/he.rda")
- #load he.rda file
- > save.image(file = "umer.Rdata") #save work space
- > history(“d:/filename.Rhistory”) #save history

Package “maps”

- > search()
- > install.packages("maps")
- > search()
- > library(maps)
- > search()
- > map("world")
- > points(19,59) #long, lat
- > points(19,59,pch=19,col="blue")

Sub, Sprsplit and Cut Functio

- **Sub Function**
- > sub("\\s","Umer","Hello There")
- [1] "HelloUmerThere“
- **strsplit Function**
- > strsplit("abc", "")
- [[1]]
- [1] "a" "b" "c“
- **Cut Function**
- > a<-c(2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020)
- > print(b<-cut(a,3))
- > print(b<-cut(a,3,labels=c("1st","2nd","3rd")))

Data Sets URLs

- data.gov
- healthdata.gov
- data.gov.in
- data.gov.uk

- **Data Sets in R:**
- > data()

Quick Questions

Quick Question-1

- At which university was the first version of R developed?
- 1- MIT
- 2- University of Auckland (Answer)
- 3- Stanford University
- 4- Oxford University

Quick Question-2

- Which of the following are recommended variable names in R? (Select all that apply.)
- 1- SquareRoot2 (Answer)
- 2-Square Root2
- 3-Square2.Root (Answer)
- 4-2SquareRoot

Quick Question 3

- If you want to add new observations to a data frame, what should you use?
- 1- The data.frame function.
- 2- The rbind function. (Answer)
- 3- The dollar sign notation.
- If you want to combine two vectors into a data frame, what should you use?
- 1-The data.frame function. (Answer)
- 2- The rbind function.
- 3- The dollar sign notation.
- If you want to add a variable to your data frame, what should you use?
- 1-The data.frame function.
- 2- The rbind function.
- 3- The dollar sign notation. (Answer)

Quick Question 4

- If you want to know the mean value of a numerical variable, which function could you use?
- 1- str
- 2- summary (Answer)
- 3- I can't figure out this information using either of these functions.
- If you want to know the standard deviation of a numerical variable, which function could you use?
- 1- str
- 2- summary
- 3- I can't figure out this information using either of these functions. (Answer)

Quick Question 5

- What is the mean value of the "Over60" variable?
- > mean(WHO$Over60)
- [1] 11.16366
- Which country has the smallest percentage of the population over 60?
- > a<-which.min(WHO$Over60)
- > WHO$Country[a]
- Which country has the largest literacy rate?
- > b<-which.max(WHO$LiteracyRate)
- > WHO$Country[b]

Quick Question 6

- Use the tapply function to find the average child mortality rate of countries in each region.
- Which region has the lowest average child mortality rate across all countries in that region?
- > corvit<-min(tapply(WHO$ChildMortality,WHO$Region,min))
- > WHO$Region[corvit]

Activities

Activities(1) 1-3

- This is your first set of activities. Every week you have the opportunity to test yourself by doing activities in R and answering questions. These activities are related to the video prior to the activities or to the overall course. You may try as many times as you want.
- Some of the questions have more than one correct alternative (checkboxes). If several options are correct, you need to click on all correct alternatives before you succeed with the question.
- Go ahead and test yourself!
  Activity 1
- Have you visited the R-project.org website? (Yes/No)
- **Activity 2**
- In what ways can you learn about R?
- 1- By meeting people in R user groups and the R user conferences.
- 2- By reading books.
- 3- By watching videos and reading online.
- 4- By experimenting in R, using the help function.
- **Activity 3**
- Is there a user group in your country? (Yes/No)
- **Tip:** Try searching something like Stockholm R user group or look up http://blog.revolutionanalytics.com/local-r-groups.html to find out.

Activities(1) 4-5

- **Activity 4**
- Consult the general FAQ under Documentation in r-project.org. Find out about the difference between package and library. What statement is true according to the information you find?
- 1- A library is a place (directory) where R stores installed packages. (Answer)
- 2- A package can contain several different libraries where you store your R-code.
- 3- A library is a standardized collection of material extending R.
- **Activity 5**
- What do you need to type to find out which additional packages are available on your current installation?
- > library()

Activities(2) 1-2

- **Activity 1**
- Where can you download the latest version of R?
- 1- http://www.statsoft.com/textbook/
- 2- http://www.r-statistics.com/
- 3- http://www.gapminder.org/
- 4- http://www.r-project.org/ (Answer)
- 5- http://www.r-bloggers.com/
- **Activity 2**
- On which operating systems can R be installed?
- 1- LINUX (Answer)
- 2- Mac (Answer)
- 3- Windows (Answer)

Activities (2) 3-5

- **Activity 3**
- Did you manage to install R on your computer? (No/Yes)
- **Activity 4**
- Have you managed to successfully open R? (Yes/No)
- **Activity 5**
- Now it is time to type some code in the window R Console. What result do you get if you type mean(1:1337) ?
- > mean(1:1337)
- [1] 669

Activities(3) 1-3

- **Activity 1**
- If you want to access the documentation pages for the function plot(), what should you write in the R console?
- 1- ?plot() (Answer)
- 2- example(plot)
- **Activity 2**
- Write one of the symbols you can use to assign values to objects in R?
- <- (Answer)
- **Activity 3**
- Create an object called a and fill it with the numbers 1, 2, 3, 4, 5. Create an object called b and fill it with the numbers 6, 7, 8, 9, 10. Then add them together (a+b). What's the result?
- 1- 7 9 11 13 15 (Answer)
- 2- 1 2 3 4 5 6 7 8 9 10
- **Soltution:**
- > a<-c(1, 2, 3, 4, 5)
- > b<-c(6, 7, 8, 9, 10)
- > a+b

Activities(3) 4-6

- **Activity 4**
- Create an object called a and fill it with the numbers 1, 2, 3, 4, 5. Create an object called b and fill it with the numbers 6, 7, 8, 9, 10. Then create a third object called ab, and fill it with a and b using the function c(). Which numbers does ab contain?
- 1- 7 9 11 13 15
- 2- 1 2 3 4 5 6 7 8 9 10 (Answer)
- **Soltution:**
- > a<-c(1, 2, 3, 4, 5)
- > b<-c(6, 7, 8, 9, 10)
- > c(a,b)
- **Activity 5**
- What built in R function calculates the standard deviation? Write in the form **function()**
- sd() (Answer)
- **Activity 6**
- What built in R function calculates the arithmetic mean? Write in the form **function()**
- mean() (Answer)

Activities(3) 7-9

- **Activity 7**
- What built in R function calculates median? Write in the form **function()**
- median() (Answer)
- **Activity 8**
- Create an object called d with this code. d <- c(1:10, 30:40, 5, 7, 9, 12). What is the median of d?
- 10 (Answer)
- **Activity 9**
- Now that you have created an object called d. What response do you get if you write sum(d)?
  473 (Answer)

Activities(3) 10-14

- **Activity 10**
- What built in R function draws a histogram? Write in the form **function()**
- hist() (Answer)
- **Activity 11**
- What built in R function draws a barplot? Write in the form **function()**
- barplot() (Answer)
- **Activity 12**
- What built in R function draws a boxplot? Write in the form **function()**
- boxplot() (Answer)
- **Activity 13**
- What built in R function draws a piechart? Write in the form **function()**
- pie() (Answer)
- **Activity 14**
- What object types can hold several different types at the same time?
- 1- data.frame (Answer)
- 2- list (Answer)
- 3- matrix
- 4- vector

Activities(3) 15-17

- **Activity 15**
- Create a vector named d, with a series of 20 integers from 31 through 50
- d<-31:50 (Answer)
- **Activity 16**
- R has a family of functions coupled to the normal distribution. What function would you use to generate random numbers?
- 1- pnorm()
- 2- qnorm()
- 3- dnorm()
- 4- rnorm() (Answer)
- **Activity 17**
- Imagine your working directory has five files and a number of folders. What command prints a list of just the five files in your working directory?
  1- getwd() 2-dir() (Answer)
- 3-dir(recursive=T) 4-length(dir()) 5-dir()[5]
- 6-history()

Activities(3) 18-20

- **Activity 18**
- List all objects currently in your workspace:
- 1- dir() 2-ls() (Answer)
- 3-data() 4-names() 5-search()
- **Activity 19**
- Select the code that saves the object x.
- 1-data(x) 2-tail(x) 3-attach(x)
- 4-load(x) 5-save(x, file="x.rda") (Answer)
- **Activity 20**
- Which of the following commands do you need to run the first time you want to draw a world map with a blue dot marking Greenwich?
- 1-install.packages("maps") 2- library("maps")
- 3-map("world") 4-points(0, 51.5)
- 5-search() 6-data()
- (Answer 1,2,3,4)

Activities(3) 21-22

- **Activity 21**
- Imagine that you have already drawn some maps with "maps" in R. Now you have started a fresh new R session and you want to draw a world map with a dot for Greenwich.
- Which of the following commands do you need to run?
- 1-install.packages("maps") 2- library("maps")
- 3-map("world") 4-points(0, 51.5)
- 5-search() 6-data()
- (Answer 2,3,4)
- **Activity 22**
- What is the number of built in objects, such as letters, LETTERS, pi, in your installation of R?
- 1-length(builtins()) (Answer)
- 2-search() 3- length(search()) 4-45
- 5-sum(length(builtins())

Activities(3) 23-25

- **Activity 23**
- How can you make names inside an object, such as a data frame or a list, visible in the R search path?
- 1- names() 2-attach() 3-requrie() (Answer 2,3)
- 4-data() 5-search()
- **Activity 24**
- How can you recreate an r object that was saved in the file x.rda?
- 1-load() (Answer) 2-install.packages() 3-library()
- 4-head() 5-attach() 6-require()
- **Activity 25**
- How can you access the vector weight inside the data frame x?
- 1- weight
- 2-x$weight (Answer)
- 3-x[1]
- 4-attach(x); weight (Answer)
- 5-require(x); weight

Activity(4) 1-3

- **Activity 1**
- What should you pay attention to when selecting a statistical method?
- 1- Variable type and scale type.
- 2- The distribution. Is the data skewed, symmetrical, normally distributed?
- 3- The scientific question you want to ask.
- 4- All of the above (Answer)
- **Activity 2**
- Choose the most informative measure of centrality for the variable annual salary
- 1-Mean 2- Mood 3- Median (Answer)
- **Activity 3**
- Combine the variables with the appropriate scale type by dragging the appropriate variable to the box beside the scale type:
- 1- Temperature, in Celsius (Interval)
- 2- Temperature, in Kelvin (Ratio)
- 3- Highest educational level reached (Ordinal)
- 4- Occupation (Nominal)

Activity(4) 4-6

- **Activity 4**
- If your data is skewed and has extreme values, how should you summarize it?
- 1- Mean
- 2-Median (Answer)
- **Activity 5**
- Which of the following are qualitative variables?
- 1- Height 2-Hair colour 3-Occupation (Answer)
- 4-Blood sugar levels
- **Activity 6**
- Your variable is Body Mass Index range, meaning that your variable indicates whether a person is healthy/ overweight/ obese. What would be a good graph to explore the distribution?
- 1- A histogram (useful for displaying the distribution of continuous numerical data)
- 2- Barplot (useful for displaying a frequency distribution for nominal or ordinal scale data) (Answer)
- 3- Boxplot (useful for displaying the distribution of continuous numerical data)

Activity(4) 7-8

- **Activity 7**
- If your variable is blood pressure measured for 100 subjects, which of these graphs would be suitable for exploring the distribution?
- 1- Histogram (Answer)
- 2- Box plot (Answer)
- 3- Pie chart
- **Explanation**
- A pie chart can be used to describe data on nominal scale. Blood pressure is a continuous variable at ratio scale. Both histogram and the more compact boxplot are excellent for describing such data.
- **Activity 8**
- If your variable is blood pressure measured for 10 subjects, what would be a good graph for exploring the distribution?
- 1- A dotplot displaying each measurement is good since there are so few measurements (Answer)
- 2- Histogram
- 3- Pie chart

Activity(4) 9-12

- **Activity 9**
- Choose a good measure of centrality for the variable body weight
- 1- Median 2-Mode 3-Mean (Answer)
- **Activity 10**
- If your qualitative variable has one dominating value, what could be a useful measure of centrality?
- 1- Median 2-Mean 3-Mode (Answer)
- **Activity 11**
- What kind of variable is blood type?
- 1- Qualitative (Answer)
- 2-Quantitative – discrete
- 3-Quantitative - continuous
- **Activity 12**
- What kind of variable is number of children in the family?
- 1- Qualitative
- 2-Quantitative – discrete (Answer)
- 3-Quantitative - continuous

Activity(4) 13-15

- **Activity 13**
- What kind of variable is marital status?
- 1- Qualitative (Answer)
- 2-Quantitative – discrete
- 3-Quantitative - continuous
- **Activity 14**
- What kind of variable is age?
- 1- Qualitative
- 2-Quantitative – discrete
- 3-Quantitative – continuous (Answer)
- **Activity 15**
- What kind of variable is: *Number of years of education*?
- 1- Qualitative
- 2-Quantitative – discrete
- 3-Quantitative – continuous (Answer)

Activity(4) 16-17

- **Activity 16**
- Which of the following variables can be displayed on a ratio scale?
- 1-Different diagnoses
- 2-Gender
- 3- Body weight (Answer)
- 4- Level of education
- 5- Temperature in Fahrenheit
- **Activity 17**
- What characteristics are true concerning data on ratio scale?
- 1-The distance between 1 and 3 equals the distance between 11 and 13
- 2-It can under some circumstances be meaningful to calculate mean, median and standard deviation
- 3-There is an absolute point zero
- 4- The variable must be quantitative
- (Answer 1,2,3,4)

Questions

Question 1-2

- **Question 1**
- Where can you download the latest version of R?
- 1- http://www.statsoft.com/textbook/
- 2- http://www.r-statistics.com/
- 3- http://www.gapminder.org/
- 4- http://www.r-project.org/ (Answer)
- 5- http://www.r-bloggers.com/
- **Question 2**
- How often is there a new full version release of R? So called x.y.0 releases.
- 1-Every month
- 2-Every year or more often (Answer)
- 3- Every Six months

Question 3

- **Question 3**
- You have read about a useful R package, sweSCB, in a blog. What code do you need to run before you can use the package?
- 1-installed.packages()
- 2-search()
- 3-install.packages("sweSCB") (Answer)
- 4-ls()
- 5- library("sweSCB") (Answer)

Explanation

- install.packages("sweSCB") is only needed once.
- library("sweSCB") is needed once every session.

Question 4-6

- **Question 4**
- Create a vector f and assign 100 random numbers to it using this code f <- rnorm(100), What code is needed to draw a histogram of f
- 1- summary(f) 2-length(f) 3-tail(f)
- 4-plot(f) 5-hist(f) (Answer)
- **Question 5**
- Type code that selects the third element from the vector **f** , based on the previous question
- > f <- rnorm(100)
- > f[3]
- **Question 6**
- Based on the previous question, replace the third element in f with 2.3
- 1-f <- 2.3
- 2- f[3] <- 2.3 (Answer)
- 3-f <- rep(2.3, 3)
- 4- 2.3 -> f[3] (Answer)

Question 7-9

- **Question 7**
- Create a sequence of integers from 5 through 15
- 1-rep(5:15,5) 2-paste("A", 1:15)
- 3- 5:15 (Answer) 4-seq(5, 15, 1) (Answer)
- **Question 8**
- Select five different letters at random.
- 1- sample(letters, 5, replace=T)
- 2-sample(letters, 5, replace=F) (Answer)
- 3-sample(1:26, 5, replace=T)
- 4-c(5,letters)
- 5-sample(LETTERS, 5, replace=F) (Answer)
- **Question 9**
- Create a vector with a 100 long random sequence of the letters a-d and visualize the proportions of the different letters using a pie-chart. What code below is relevant?
- 1- a <- sample(letters[1:4], 100, replace=F) 2-pie(a) 3-plot(a) 4-plot(table(a))
- 5- a<- sample(letters[1:4], 100, replace=T) 6-pie(table(a)) (Answer)

Question 10

- **Question 10**
- Use the code to create a vector length of ten random body lengths in cm. Mean = 180 cm, standard deviation = 10 cm. Use the code to create a vector length of ten random body lengths in cm. Mean = 180 cm, standard deviation = 10 cm.
- > length <- rnorm(10, 180, 10)
- Use the code to create a vector of corresponding body weights, given a BMI of 25.
- > weight <- (length/100)^2 * 25
- Create a data.frame of the two vectors.
- > measurements <- data.frame(cbind(length, weight))
- What code creates a new data.frame with a reasonable number of decimals?
- 1- measurements <- round(measurements,4)
- 2- measurements$length <- round(measurements$length)
- 3-str(measurements)
- 4-measurements$weight
- 5- measurements <- round(measurements) (Answer)