Introduction to R

Slides by Pimprapai Thainiam

About R and R Studio

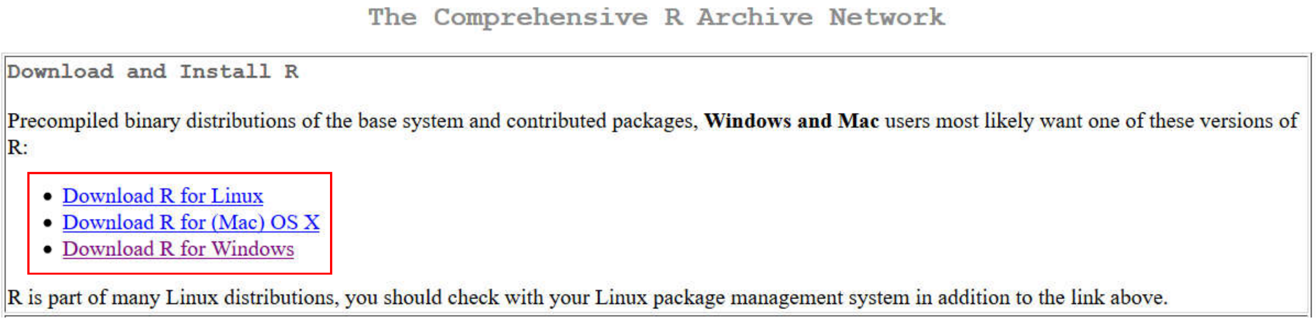
- R is an open source and free software environment for statistical computing and graphics.
- R studio is a powerful and productive user interface for R. It's free and open source, and works great on Windows, Mac, and Linux.
- R is an object oriented programming language where we create objects and manipulate them as intended. Objects can be data frames, vectors, matrices, lists, raw data, spatial objects , maps etc.

Why R Language?

- R is not just a statistics package, it’s a language (allows you to specify the performance of new tasks without any limitations)
- R is open-source and free to use
- R has a large and active community
- R provides state-of-the-art algorithm (> 10,000 extension packages on CRAN)
- R creates beautiful visualizations (as seen in the New York Times and The Economist)
- R is designed to operate the way that problems are thought about and has very simple syntax.
- R is very interactive and thus suitable for data analysis.

Installing R and R Studio

- R : https://cran.r-project.org/



The Comprehensive R Archive Network

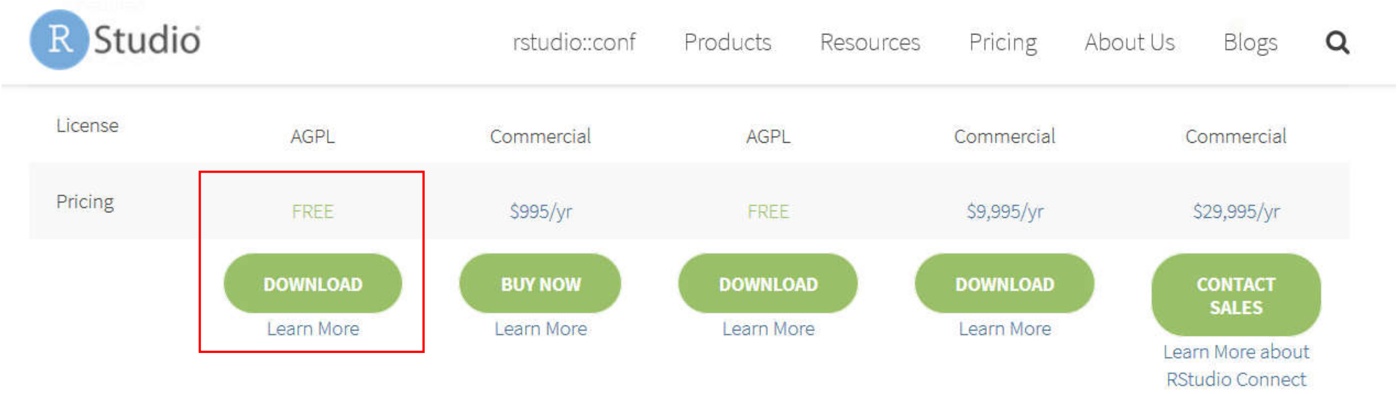
Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

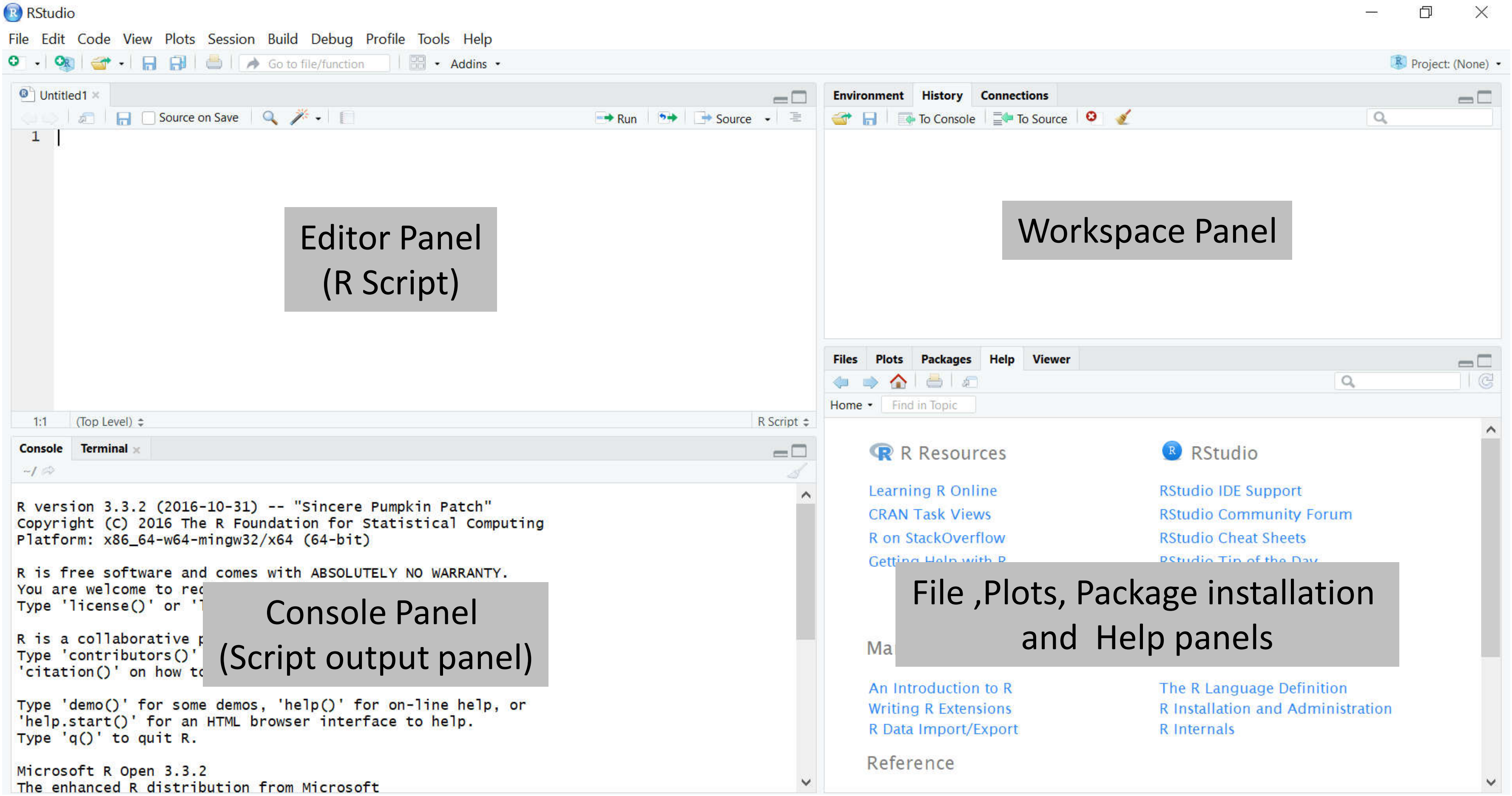
R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

- R studio : https://www.rstudio.com/products/rstudio/download/



License	AGPL	Commercial	AGPL	Commercial	Commercial
Pricing	FREE	$995/yr	FREE	$9,995/yr	$29,995/yr
	DOWNLOAD Learn More	BUY NOW Learn More	DOWNLOAD Learn More	DOWNLOAD Learn More	CONTACT SALES Learn More about RStudio Connect

R Studio



Editor Panel (R Script)

Workspace Panel

Console Panel (Script output panel)

File ,Plots, Package installation and Help panels

```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to rec
Type 'license()' or '

R is a collaborative p
Type 'contributors()'
'citation()' on how to

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Microsoft R Open 3.3.2
The enhanced R distribution from Microsoft
```

Topics

- Simple manipulations, numbers and vectors
- Objects, their modes and attributes
- Ordered and unordered factors
- Arrays and matrices
- Lists and data frames
- Reading data from files
- Grouping, loops and conditional execution
- Writing function
- Packages
- Getting help
- Exercises

Simple manipulations, numbers and vectors

- **Assignment:** An object can be created with the “assign” operator (<-) which is written as an arrow with a less-than sign (<) and a minus sign (-) or (=) or function assign( ) .

```
> x <- 2
> y = 3
> assign("z",4)
```

- **Vectors:** A vector can be created by using operator c( ) .

```
> x <- c(9,4,2,5,8)
> b <- c(x,0,x)
> c <- 2*x + y + 1
```

- **Common arithmetic operators:** + , ~ , * , / , min( ), max( ), sum( ), length( ), prod( ), mean( ) , etc.

```
> 1/x
> x+1
> mean(x)
> length(x)
```

Simple manipulations, numbers and vectors

- **Generating regular sequence:** A regular sequence can be created by using colon operator (:) or function $seq(\,)$ or function $rep(\,)$.

```
> x1 <- 1:10 # create a vector starting from 1 to 10 (incrementing by 1)
> x2 <- 10:1 # create a vector starting from 10 to 1 (decrementing by 1)
> x3 <- seq(-5,5,by=0.2) #incrementing by 0.2
> x4 <- seq(5,-5,by=-0.2) #decrementing by -0.2
> x5 <- seq(length=10, from=-5, by=0.2)
> x6 <- rep(2, times=5)
> x6 <- rep(2, 5) #for short of the above command
> x7 <- rep(x, 5) #put 5 copies of x end-to-end
> x8 <- rep(x, each=5) #repeat each element of x 5 times before moving to the next
```

- **Logical Vectors:** Logical operators are < , > , <= , >= , ==.

  Logical expressions are & (and), | (or), == (equality), ! (negation).

```
> tf <- x>5
> as.numeric(tf)
```

Simple manipulations, numbers and vectors

- **Missing values:** Test missing value by using function is.na( )

```
> m <- c(1:3, NA) #create a vector of size 4 with missing value
> mm <- is.na(m) #find missing values
```

- **Other values:**

```
> n <- 0/0 #create a NaN (not a number)
> n
[1] NaN
> i <- 2^5000 #cretate infinity
> i
[1] Inf
```

Simple manipulations, numbers and vectors

- **Character vectors:** Use “ ” to enter character string.

```
> s <- c("Michael","Nancy","Vicky")
> s
[1] "Michael" "Nancy" "Vicky"
> paste("Hello", s) #pasting strings together
[1] "Hello Michael" "Hello Nancy" "Hello Vicky"
> labs <- paste(c("X","Y"), 1:10, sep="") #c("X","Y") is repeat 5 times to match 1:10
> labs
[1] "X1" "Y2" "X3" "Y4" "X5" "Y6" "X7" "Y8" "X9" "Y10"
```

Simple manipulations, numbers and vectors

- **Selecting and modifying subsets:** Use operator [ ]

```
> x <- c(8,6,4,2,0)
> x[1] #select the first element
[1] 8
> x[-1] #remove the first element
[1] 6 4 2 0
> x[2:4] #select elements 2 to 4
[1] 6 4 2
> x[-(2:4)] #remove elements 2 to 4
[1] 8 0
> x[x>4] #select elements that are more than 4
[1] 8 6
> x[2:4] <- 1:3 #replace elements
> x
[1] 8 1 2 3 0
```

Simple manipulations, numbers and vectors

- **Function names( ):**

```
> fruit <- c(5,10,1,20)
> names(fruit) <- c("orange","banana","apple","peach")
> fruit
orange banana apple peach
     5     10     1    20
> lunch <- fruit[c("apple","orange")]
> lunch
apple orange
    1      5
```

Objects, their modes and attributes

All entities in R are called objects. Each object has two intrinsic attributes: “mode” and “length”.

- **Intrinsic attribute: mode**
  - numeric, complex, logical, character, integer, single, double, expression, list, raw
  - Vectors must have their values all of the same mode.
  - R also operates on objects called lists, which are of mode list. These are ordered sequences of objects which individually can be of any mode.

```
> z <- 0:9
> mode(z)
[1] "numeric"
> zz <- as.character(z) #coercion
> mode(zz)
[1] "character"
> zzz <- as.numeric(zz)
> mode(zzz)
[1] "numeric"
```

Objects, their modes and attributes

- **Intrinsic attribute: length**

```
> length(z)
[1] 10
> e <- numeric() #make e an empty vector structure of mode numeric
> e
numeric(0)
> length(e)
[1] 0
> e[5] <- 12 #implicitly change length of e
> e
[1] NA NA NA NA 12
> length(e) <- 7 #changing the length of e explicitly (vector can be extended its length by missing value)
> e
[1] NA NA NA NA 12 NA NA
> aa <- 11:20
> aa
[1] 11 12 13 14 15 16 17 18 19 20
> aa <- aa[2*1:5] #make it an object of length 5 consisting of just the former components with even index
> length(aa)
[1] 5
> length(aa) <- 3 #retain just the first 3 values
> aa
[1] 12 14 16
```

Objects, their modes and attributes

- **Getting and setting attributes:**
  - attributes(obj) is used to return a list of all the non-intrinsic attributes currently define for that object.
  - attr(obj, “name”) is used to select a specific attribute

```
> z <- 1:4
> attributes(z)
NULL
> class(z)
[1] "integer"
> z
[1] 1 2 3 4
> attr(z, "dim") <- c(2,2)
> attributes(z)
$dim
[1] 2 2
> class(z)
[1] "matrix"
> z
    [,1] [,2]
[1,]  1   3
[2,]  2   4
```

The dim attribute allows R to treat z as a matrix

Objects, their modes and attributes

- **The class of an object:** Class of an object is reported by function class( )
  - For simple vectors this is just the mode, for example "numeric", "logical", "character" or "list", but "matrix", "array", "factor" and "data.frame" are other possible values.
  - The class of the object is used to allow for an object-oriented style of programming in R

The dim attribute allows R to treat z as a matrix

Ordered and unordered factors

- A **Factor** is a vector object used to specify a discrete classifcation (grouping) of the components of other vectors of the same length. A factor can be created using function factor( ) and the levels of a factor can be identified using function levels ( )

```
> state <- c("tas", "sa", "qld", "nsw", "nsw", "nt", "wa", "wa",
+            "qld", "vic", "nsw", "vic", "qld", "qld", "sa", "tas",
+            "sa", "nt", "wa", "vic", "qld", "nsw", "nsw", "wa",
+            "sa", "act", "nsw", "vic", "vic", "act")
> statef <- factor(state)
> statef
[1] tas sa qld nsw nsw nt wa wa qld vic nsw vic qld qld sa tas sa nt wa vic qld nsw
nsw wa sa act nsw vic vic act
Levels: act nsw nt qld sa tas vic wa
levels(statef)
[1] "act" "nsw" "nt" "qld" "sa" "tas" "vic" "wa"
```

Ordered and unordered factors

- tapply( ): Used to apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
+              61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
+              59, 46, 58, 43)
> incmeans <- tapply(incomes, statef, mean) #calculate the sample mean income for each 
level of statef
> incmeans
       act      nsw       nt      qld       sa      tas      vic       wa 
  44.50000 57.33333 55.50000 53.60000 55.00000 60.50000 56.00000 52.25000
```

tapply() is used to apply a function, here mean(), to each group of components of the first argument, here incomes, defined by the levels of the second component, here statef

Ordered and unordered factors

- ordered( ): Used to create such ordered factors but is otherwise identical to factor.

```
> stateo <- ordered(state)
> stateo
[1] tas sa qld nsw nsw nt wa wa qld vic nsw vic qld qld sa tas sa nt wa vic qld nsw
nsw wa sa act nsw vic vic act
Levels: act < nsw < nt < qld < sa < tas < vic < wa
> levels(stateo)
[1] "act" "nsw" "nt" "qld" "sa" "tas" "vic" "wa"
```

Levels of stateo are ordered alphabetically.

Arrays and matrices

- **Arrays:** Arrays are just vectors with *dim* attribute. It can be created by using function array(data_vector, dim=dimension_vector)

  where data_vector is a vector containing data

  dimension_vector is a vector of non-negative integers with length *k* which means the array has *k* dimensions

```
> z <- 1:12
> dim(z) <- c(2,3,2) #set dimensions to vector z; z becomes the array
> z
, , 1
    [,1] [,2] [,3]
[1,]   1    3    5
[2,]   2    4    6
, , 2
    [,1] [,2] [,3]
[1,]   7    9   11
[2,]   8   10   12
> zz <- array(1:12, dim=c(2,3,2)) #create array using function array()
```

Arrays and matrices

- **Array indexing:** Use [ ] to specify location

```
> zz[2,3,2]  #data in row 2, column 3, and layer 2 of zz
[1] 12
> zz[2,3,]  #data in row 2, column 3, and all layers of zz
[1] 6 12
> zz[2,,]  #data in row 2, all columns and all layers of zz
    [,1] [,2]
[1,]   2    8
[2,]   4   10
[3,]   6   12
> zz[2,2:3,1] #data in row 2, column 2 and 3, and layer 1 of zz
[1] 4  6
```

Arrays and matrices

- **Matrix:** Matrix can be generated by using function matrix(data_vector, num_rows, num_columns)

  where num_rows is number of rows

  num_columns is number of columns

  data_vector is a vector of size num_rows * num_columns

```
> x <- matrix(1:20,4,5) #4 = number of rows, 5 = number of columns
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Arrays and matrices

- **Index matrices:** Index matrix can be used in order either to

  - Assign a vector of quantities to an irregular collection of elements in the array

  - Extract an irregular collection as a vector

```
> i <- matrix(c(1:3,3:1),3,2) #Generate a 3 by 2 index matrix
> i
    [,1] [,2]
[1,]   1    3
[2,]   2    2
[3,]   3    1
> x[i] #Extract those elements
[1] 9 6 3
> x[i] <- 0 #Replace those elements by 0
> x
    [,1] [,2] [,3] [,4] [,5]
[1,]   1    5    0   13   17
[2,]   2    0   10   14   18
[3,]   0    7   11   15   19
[4,]   4    8   12   16   20
```

Arrays and matrices

- **Matrix facilities:** Finding dimension, number of rows and column, length, transpose

```
> dim(x)
[1] 4 5
> nrow(x)
[1] 4
> ncol(x)
[1] 5
> length(x)
[1] 20
> t(x)
     [,1] [,2] [,3] [,4]
[1,]    1    2    0    4
[2,]    5    0    7    8
[3,]    0   10   11   12
[4,]   13   14   15   16
[5,]   17   18   19   20
```

Arrays and matrices

- **Named rows and columns:**

```
> y <- matrix(1:6,2,3)
> y
[,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
> rownames(y) <- c("Michael","Peter")
> y
       [,1] [,2] [,3]
Michael   1    3    5
Peter     2    4    6
> colnames(y) <- c("Age","Weight","Height")
> y
        Age Weight Height
Michael   1      3      5
Peter     2      4      6
> dimnames(y)
[[1]]
[1] "Michael" "Peter"
[[2]]
[1] "Age" "Weight" "Height"
```

Arrays and matrices

- **Forming partition matrices:** A matrix could be generated from vectors or matrices by using functions cbind( ) and rbind( )
  - cbind( ) forms matrices by binding together matrices horizontally or column-wise
  - rbind( ) forms matrices by binding together matrices vertically or row-wise

```
> m1 <- matrix(1,2,2)
> m2 <- matrix(2,2,2)
> cm <- cbind(m1,m2)
> cm
     [,1] [,2] [,3] [,4]
[1,]    1    1    2    2
[2,]    1    1    2    2
> rm <- rbind(m1,m2)
> rm
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    2    2
[4,]    2    2
> crm <- rbind(cm,cbind(m2,m1))
> crm
     [,1] [,2] [,3] [,4]
[1,]    1    1    2    2
[2,]    1    1    2    2
[3,]    2    2    1    1
[4,]    2    2    1    1
```

Lists and data frames

- **Lists:** An list is an object consisting of an ordered collection of objects known as its components.

```
> lst <- list(name="Fred",wife="Mary",no.children="3",child.ages=c(4,8,9))
> lst
$name
[1] "Fred"
$wife
[1] "Mary"
$no.children
[1] "3"
$child.ages
[1] 4 8 9
> lst[[2]]
[1] "Mary"
> lst$wife
[1] "Mary"
> lst[[4]]
[1] 4 8 9
> lst[[4]][1]
[1] 4
> lst$child.ages[1]
[1] 4
```

Lists and data frames

- **Data frames:** A data frame is a list with class "data.frame“ that look like a matrix with mixed data types.

```
> df <- data.frame(name=c("Michael","Mark","Maggie"), children=c(2,0,2))
> df
     name children
1 Michael        2
2    Mark        0
3  Maggie        2
> df$name
[1] Michael Mark Maggie
Levels: Maggie Mark Michael
> df[1,]
     name children
1 Michael        2
> df[,1]
[1] Michael Mark Maggie
Levels: Maggie Mark Michael
```

Reading data from files

- Functions read.table( ) and write.table( ) can be used to read/write complete file from/to data.frames. The file format can be space or tab-separated.
- Functions read.csv( ) and write.csv( ) can also be used to read and write complete file from/to CSV file.

```
> write.table(df, file="df.dat", sep=",")
> df2 <- read.table("df.dat", sep=",")
> df2
     name children
1 Michael        2
2    Mark        0
3  Maggie        2
> write.csv(df, file="df.csv")
> df3 <- read.csv("df.csv")
> df3
  X    name children
1 1 Michael        2
2 2    Mark        0
3 3  Maggie        2
```

Grouping, loops and conditional execution

- **Conditional execution:** If **statements**

```
> x <- 12
> if (x > 10) {
+   cat("x is >10")
+ } else {
+   cat("x is <=10")
+ }
x is >10
> x <- c(12,16,3)
> if(all(x>10)) cat("All values in x is >10")
> if(any(x>10)) cat("There is at least one value >10")
There is at least one value >10
```

Grouping, loops and conditional execution

- **Repetitive execution:** for **loops, repeat** and **while**

```
> x <- 0
> for (i in 1:5){
+   x <- x+i
+ }
> x
[1] 15
```

```
> x <- 0
> c <- 1
> repeat{
+   x <- x+c
+   c <- c+1
+ if (c == 6) break
+ }
> x
[1] 15
> c
[1] 6
```

```
> x <- 0
> c <- 1
> while(c < 6){
+   x <- x+c
+   c <- c+1
+ }
> x
[1] 15
> c
[1] 6
```

Writing function

- R is a functional programming language. Functions are objects of mode “function”. Since functions are regular (first class) objects they can be passed on as arguments and returned by functions.

```
> inc <- function(x) {x <- x+1}
> inc
function(x) {x <- x+1}
> mode(inc)
[1] "function"
> x1 <- inc(5) #call function inc
> x1
[1] 6
> x2 <- inc(1:10)
> x2
[1] 2 3 4 5 6 7 8 9 10 11
> inc <- function(x, b=1) {x <- x+b} #setting defaults
> x1 <- inc(5)
> x1
[1] 6
> x2 <- inc(1:10,10) #change b from 1 to 10
> x2
[1] 11 12 13 14 15 16 17 18 19 20
```

lapply, sapply, apply

- lapply( ) is a function used to apply functions to lists
- sapply( ) is a user-friendly version and wrapper of lapply
- apply( ) is use for row operation or column operation

```
> lt <- list(1:3,6,7:3)
> lapply(lt, FUN=function(x) {rev(x)})
#apply reverse function to all element
[[1]]
[1] 3 2 1
[[2]]
[1] 6
[[3]]
[1] 3 4 5 6 7
> sapply(lt, length)
[1] 3 1 5
> m <- matrix(1:9,3) #create a matrix
> m
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
> apply(m, MARGIN=1, sum) #MARGIN=1 means row operation
[1] 12 15 18
> apply(m, MARGIN=2, sum) #MARGIN=1 means column operation
[1] 6 15 24
> rowSums(m)
[1] 12 15 18
> colSums(m)
[1] 6 15 24
```

Packages

- All R functions and datasets are stored in *packages*.
- To use a package, you have to install that package using install.packages(“name”) where name is name of package.
- To update a package, use the command update.packages(“name”)
- To load a particular package, use the command library(name)
- To see which packages are installed at your site, use the command library()
- To see which packages are currently loaded, use the command search()
- There are thousands of contributed packages for R, written by many different authors.
- Find available packages at

  https://cran.r-project.org/web/packages/available_packages_by_name.html

Getting help

- R comes with online help
  - To get help about something in particular, for example, get help about solve function, you can use the following functions

```
help(solve)

?solve
```

  - To launch a web browser that allows the help pages to be browsed with hyperlinks

```
help.start(solve)
```

  - To allow R searching for help in various ways about solve

```
??solve
```

Exercises

1. Create a vector with 10 numbers (3, 12, 6, -5, 0, 8, 15, 1, -10, 7) by you and assign it to x.
2. What is the “data type” of x? How can you find out?
3. Subtract 5 from the 2nd, 4th, 6th, etc. element in x.
4. Compute the sum and the average for x (there are functions for that).
5. Reverse the order of the elements in x.
6. Find out which numbers in x are negative.
7. Remove all entries with negative numbers from x.
8. How long is x now (use a function).
9. Remove x from the environment/workspace (session).
10. Create the a vector of strings containing “CSE 8001”, “CSE 8002", ...,”CSE 8100” using paste.