Introduction to R

Open Source Tools for Intelligent Systems
summer semester 2017

Rami Yaari (ramiyaari@gmail.com)
Naama Kopelman (naama.kopelman@gmail.com)

Outline for the R Segment of the Course

The R segment of the course will include the following topics:

- Introduction to R
- Data visualization in R
- Data transformation in R
- Handling Time Series in R
- Modeling in R
- Applying machine learning algorithms using R
- Building dynamic documents with R
- Web application framework for R

List of Sources

The presentations for this segment of the course were made with the help of the following sources:

- The book ‘R for Data Science’ - available online at `http://r4ds.had.co.nz/`
- The book ‘R for Statistics’ (CRC Press)
- Slides from the course ‘Introduction to Statistical and Scientific Programming in R’ given by Dr. Yair Goldberg and Dr. Yuval Nov at the Haifa University - available online at http://stat.haifa.ac.il/~ygoldberg/Teaching/R/
- The book ‘An introduction to statistical learning with applications in R’ - available online at `http://www-bcf.usc.edu/~gareth/ISL/`
- Online material and blogs about R

Background on R

- An open source programming language
- Developed for statistical computing (since 1995)
- Freely distributed by CRAN (Comprehensive R Archive Network)
- Runs on Windows/MAC OS X/Linux
- Popular among statisticians and data analysts

R Features

- High-level language: procedural, elements of OOP
- Supports matrix arithmetics
- Strong graphical support
- Interpreted language (no compilation)
- Easily extended and shared using packages
- Implementations of a wide variety of statistical procedures

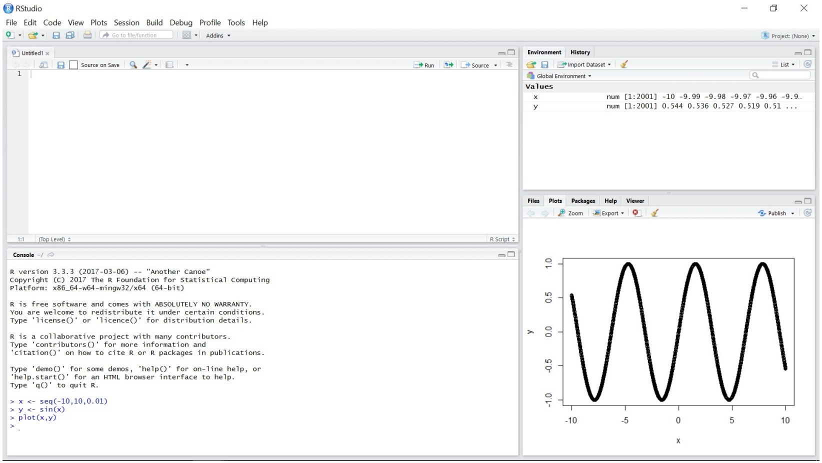
RStudio

- A free open source IDE for R
- Can be run on a desktop or using a browser connected to RStudio server
- Can be downloaded from `https://www.rstudio.com/products/rstudio/`
- Must install R before installing RStudio

Main panes

- Console - allows running commands one at a time
- Editor - allows writing and running R programs
- Environment - information on currently loaded R objects
- History - history of previous commands
- Files - content of working directory
- Plots - all open figures
- Packages - list of currently installed/loaded packages
- Help - help information

RStudio



```
Values
x                          num [1:2001] -10 -9.99 -9.98 -9.97 -9.96 -9.9…
y                          num [1:2001] 0.544 0.536 0.527 0.519 0.51 ...
```

```
R version 3.3.3 (2017-03-06) -- "Another Canoe"
Copyright (C) 2017 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 redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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.

> x <- seq(-10,10,0.01)
> y <- sin(x)
> plot(x,y)
> 
```

Figure 1: RStudio Environment

R Session

- To open a new R session select ‘Session->New Session’ from the menu.

R Session

- To open a new R session select ‘Session->New Session’ from the menu.
- Set the working directory from the menu using ‘Session->Set Working Directory’ or using the setwd() command.

R Session

- To open a new R session select ‘Session->New Session’ from the menu.
- Set the working directory from the menu using ‘Session->Set Working Directory’ or using the setwd() command.
- You can see the working directory using the getwd() command or by looking at the ‘Files’ pane.

R Session

- To open a new R session select ‘Session->New Session’ from the menu.
- Set the working directory from the menu using ‘Session->Set Working Directory’ or using the setwd() command.
- You can see the working directory using the getwd() command or by looking at the ‘Files’ pane.
- When quiting you can save the session from the menu using ‘Session->Save Workspace As. . . ’ or using the save.image() command. This will create an “.Rdata” file with the given name.

R Session

- To open a new R session select ‘Session->New Session’ from the menu.
- Set the working directory from the menu using ‘Session->Set Working Directory’ or using the setwd() command.
- You can see the working directory using the getwd() command or by looking at the ‘Files’ pane.
- When quiting you can save the session from the menu using ‘Session->Save Workspace As. . . ’ or using the save.image() command. This will create an “.Rdata” file with the given name.
- If you quit without saving - R will ask if you want to save the workspace to a file with an empty name.

R Session

- To open a new R session select ‘Session->New Session’ from the menu.
- Set the working directory from the menu using ‘Session->Set Working Directory’ or using the setwd() command.
- You can see the working directory using the getwd() command or by looking at the ‘Files’ pane.
- When quiting you can save the session from the menu using ‘Session->Save Workspace As...’ or using the save.image() command. This will create an “.Rdata” file with the given name.
- If you quit without saving - R will ask if you want to save the workspace to a file with an empty name.
- You can load a saved session from the menu using ‘Session->Load Workpace...’, using the load.image() command or by clicking on the relevant “.Rdata” file and opening it with RStudio (by default it will open with regular R console).

R Session

- To open a new R session select ‘Session->New Session’ from the menu.
- Set the working directory from the menu using ‘Session->Set Working Directory’ or using the setwd() command.
- You can see the working directory using the getwd() command or by looking at the ‘Files’ pane.
- When quiting you can save the session from the menu using ‘Session->Save Workspace As...’ or using the save.image() command. This will create an “.Rdata” file with the given name.
- If you quit without saving - R will ask if you want to save the workspace to a file with an empty name.
- You can load a saved session from the menu using ‘Session->Load Workpace...’, using the load.image() command or by clicking on the relevant “.Rdata” file and opening it with RStudio (by default it will open with regular R console).
- The loaded session will contain the saved environment including all previously created objects.

Help on R

- Type ?command to open a help page on the requested command in the 'Help' pane (i.e. ?seq).
- From the Help menu - links to online help, keyboard shortcuts and cheatsheets.
- The RSiteSearch() command opens a browser and searches for the requested term on a dedicated search engine.
- Numerous online forums and blogs on any subject (just Google it).

R Basics

- Arithmetic operators
- Logical operators
- R objects
- Control flows
- Functions
- R functions for probability distributions
- R packages/libraries

Code Chunks in the Presentation

In the following slides, there are code snipets (produced using the rmarkdownn package which we will get to later).

- The code is highlighted in a blue background
- Comments in the code in R begin with #
- The output of the code as would be seen in the RStudio console is marked with ## at the beginning of each row

```
print('welcome to R')  #printing to the console
```

```
## [1] "welcome to R"
```

Basic arithmetic operators

```r
1+1
```

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

```r
2*5 - 1 / 2 #spaces doesn't effect the result
```

```
## [1] 9.5
```

```r
3^2+1
```

```
## [1] 10
```

```r
3^(2+1)      #use parenthesis to define order of operations
```

```
## [1] 27
```

Basic Arithmetic Operators

```r
9%/%2       #integer division
```

```
## [1] 4
```

```r
9%%2        #remainder of integer division
```

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

```r
2^100       #power
```

```
## [1] 1.267651e+30
```

```r
2^(-100)   #large/small numbers are shown in floating point format
```

```
## [1] 7.888609e-31
```

Arithmetic Operators Using Built-in Functions

Examples of some built-in artithmetic functions:

```
sqrt(9)           #square root
```

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

```
abs(-5)           #absolute value
```

```
## [1] 5
```

```
exp(1)            #natural exponential function
```

```
## [1] 2.718282
```

```
log10(1e3)        #logarithm using base 10
```

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

Arithmetic Operators Using Built-in Functions

```r
round(9.2); floor(9.2); ceiling(9.2)
```

```
## [1] 9
```

```
## [1] 9
```

```
## [1] 10
```

```r
log(exp(2))          #functions can be combined together
```

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

```r
sin(pi); cos(pi)     #note the floating-point inaccuracy issue
```

```
## [1] 1.224606e-16
```

```
## [1] -1
```

Logical Operators

```r
1 == 1      #equal
```

```
## [1] TRUE
```

```r
2 != 1      #not equal
```

```
## [1] TRUE
```

```r
2 < 1       #less than
```

```
## [1] FALSE
```

```r
2 >= 2      #equal or greater than
```

```
## [1] TRUE
```

Logical Operators

Use parenthesis to compose clear and accurate logical terms

```
(2==2) & (2==1)              #& means AND
```

```
## [1] FALSE
```

```
(2==2) | (2==1)              #| means OR
```

```
## [1] TRUE
```

```
(2==2) & !(2==1)             #! means NOT
```

```
## [1] TRUE
```

```
((2==2)==T) & ((2==1)==F)   #T is TRUE and F is FALSE
```

```
## [1] TRUE
```

Creating New Objects

New objects are created by assigning values to a variable using the assignment operators <-, ->, or =. Common practice is to use just the operator <-. The keyboard shortcut for <- is Alt + - (no space allowed in between <-).

```
a <- 99.9    # 99.9 is assigned to a
b <- a*2     # existing variables can be used in assigning new ones
b -> c       # b is assigned to c
c = a        # a is assigned to c, old value of c is replaced
```

Variable Names:

- can include letters, digits, dash, underscore and a period
- must begin with a letter (or period if the 2nd character isn't a digit)
- are case sensitive

Displaying and Deleting Objects

To display the value of object x use the print(x) function or type the object name:

```
print(a);
```

```
## [1] 99.9
```

```
b
```

```
## [1] 199.8
```

- R saves all the objects created in the session data.
- The list of objects are shown in the Environment pane or use the ls() or objects() functions.
- To remove object x use the rm(x) function.
- To remove all objects use rm(list=ls()).

Type of Objects

Atomic Object Types:

- null (empty object): NULL
- logical (boolean): TRUE and FALSE (or T and F)
- numeric (real number): 1, 2.32, 1e-5, pi (includes both integer and double)
- complex (complex number): 2+0i, 2i
- character (chain of characters); ‘A’, ‘hi’, “Hello”

Testing the Type of an Object

Use the mode(x) or typeof(x) functions to see the mode/type of object x. Use the functions is.null(x), is.logical(x), is.numeric(x), is.complex(x), is.character(x) in order to test the type of object x:

```r
x <- 1; mode(x)
```

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

```r
x <- 'hi'; mode(x)
```

```
## [1] "character"
```

```r
is.numeric(x)
```

```
## [1] FALSE
```

```r
is.character(x)
```

```
## [1] TRUE
```

Converting the Type of an Object

Use the functions as.logical(x), as.numeric(x), as.complex(x), as.character(x) in order to explicity convert the type of object x. R will convert the value of the object accordingly:

```
z <- 1; z <- as.logical(z); print(mode(z)); print(z)
```

```
## [1] "logical"

## [1] TRUE
```

```
c <- as.character(z); print(mode(c)); print(c)
```

```
## [1] "character"

## [1] "TRUE"
```

Special Values

Notable special values in R include:

- NA (not available) - signifies missing information
- NaN (not a number) - can be obtained from a calculation problem
- Inf/-Inf (infinite)

Use the functions is.na(x), is.nan(x), and is.infinite(x) to test whether object x holds one of these special values:

```
x <- NA; is.na(x)
```

```
## [1] TRUE
```

```
y <- log(-1); is.nan(y)
```

```
## Warning in log(-1): NaNs produced
```

```
## [1] TRUE
```

```
z <- 1/0; is.infinite(z)
```

```
## [1] TRUE
```

Exercises - Objects and Operators

1. Create a variable named x with the value 123.
2. Convert x into a character variable. Convert x into a logical variable. What values did you get?
3. List all existing variables and verify x exist, then delete x and verify its gone.
4. Create variables x and y with assigned values, then switch their values using a third variable.
5. Calculate the roots of x^2+2*x-3

Object Collections

R supports the following basic object collections:

- vector - a 1-D indexed collection of atomic objects of a given type
- matrix - a 2-D indexed collection of atomic objects of a given type
- array - an n-D indexed collection of atomic objects of a given type
- list - an ordered collection of components (atomic object or a collection), possibly of different types
- data frame - a list whose components are vectors of a given length

Use the functions is.atomic(x), is.vector(x), is.matrix(x), is.array(x), is.list(x) and is.data.frame(x) to test the collection type of object x.

Vectors

- A vector is a 1-D indexed collection of atomic objects of a given type.
- Vectors of a particlar type can be constructed using the logical(), numeric(), complex(), character() functions or the vector() function with a specified type.
- The c() function constructs a vector using concatenated values.

```r
vector(mode='logical',length=3)
```

```
## [1] FALSE FALSE FALSE
```

```r
numeric(length=3)
```

```
## [1] 0 0 0
```

```r
c('a','b','c','d')
```

```
## [1] "a" "b" "c" "d"
```

The : Operator

The : operator constructs numeric vectors, increasing or decreasing, whose step size is 1/-1:

```r
1.5:5.5
```

```
## [1] 1.5 2.5 3.5 4.5 5.5
```

```r
5:1
```

```
## [1] 5 4 3 2 1
```

```r
1:5+2
```

```
## [1] 3 4 5 6 7
```

```r
c(1:3,3:1)
```

```
## [1] 1 2 3 3 2 1
```

The Seq() Function

The seq() function expands on the : operator by allowing to define a step size or the total number of elements in the vector:

```
seq(0,10)          #creates a vector from 0 to 10 with step-size=1
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

```
seq(0,10,by=2)     #creates a vector from 0 to 10 with step-size=2
```

```
## [1]  0  2  4  6  8 10
```

```
seq(0,10,length=4) #creates an evenly spaced vector with 4 elements
```

```
## [1]  0.000000  3.333333  6.666667 10.000000
```

The Length of a Vector

Use length(x) to obtain the length of vector x. When printing a vector, the number inside [] at the beginning of each row shows the index of the first element of the vector in this row:

```r
vec <- 100:133
length(vec)
```

```
## [1] 34
```

```r
vec
```

```
##  [1] 100 101 102 103 104 105 106 107 108 109
## [11] 110 111 112 113 114 115 116 117 118 119
## [21] 120 121 122 123 124 125 126 127 128 129
## [31] 130 131 132 133
```

The [ ] Operator

The operator [ ] alows us to select a particular index in a vector. Used in combination with the : operator we can access selected indices.

```
(vec <- -5:5)
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

```
vec[1]
```

```
## [1] -5
```

```
vec[3:8]
```

```
## [1] -3 -2 -1  0  1  2
```

```
vec[-(3:8)]  #selecting all indices besides 3:8
```

```
## [1] -5 -4  3  4  5
```

```
vec[12]       #attempting to access a non-existent index
```

```
## [1] NA
```

Manipulating a Vector

```
(vec <- 1:5)
```

```
## [1] 1 2 3 4 5
```

```
vec <- c(vec,100,101); vec      #add elements to the vector
```

```
## [1]   1   2   3   4   5 100 101
```

```
vec[1:2] <- vec[5]*10; vec      #update values of selected elements
```

```
## [1]  50  50   3   4   5 100 101
```

```
vec <- vec[-(1:2)]; vec         #remove selected elements
```

```
## [1]   3   4   5 100 101
```

The which() Function

Use the which() function to obtain the indices of elements of a vector that follow a given logical condition:

```
vec <- c('Male','Male','Female',NA,'Female'); vec
```

```
## [1] "Male"   "Male"   "Female" NA       "Female"
```

```
which(vec=='Female')
```

```
## [1] 3 5
```

```
which(is.na(vec))
```

```
## [1] 4
```

Useful Functions for Numeric Vectors

```
(x <- c(1:4,10))
```

```
## [1]  1  2  3  4 10
```

```
c(sum(x),prod(x))          #sum and product of the vector elements
```

```
## [1]  20 240
```

```
c(min(x),max(x))           #minimum and maximum of elements
```

```
## [1]  1 10
```

```
c(mean(x),median(x))       #mean and median of elements
```

```
## [1] 4 3
```

```
c(sd(x),var(x))            #standard deviation and variance
```

```
## [1]  3.535534 12.500000
```

Useful Functions for Numeric Vectors

```r
(x <- c(3:1,0:3))
```

```
## [1] 3 2 1 0 1 2 3
```

```r
(x <- sort(x))      #sort ascending
```

```
## [1] 0 1 1 2 2 3 3
```

```r
cumsum(x)           #cumulative sum
```

```
## [1]  0  1  2  4  6  9 12
```

```r
unique(x)           #set of unique values
```

```
## [1] 0 1 2 3
```

```r
rep(x,2)            #replicate x 2 times
```

```
##  [1] 0 1 1 2 2 3 3 0 1 1 2 2 3 3
```

Useful Functions for Logical Vectors

Use the any(x) and all(x) functions to test whether a logical condition applies to any or all of the elements of a logical vector x:

```
(num_vec <- -3:3)
```

```
## [1] -3 -2 -1  0  1  2  3
```

```
(log_vec <- (num_vec>0))
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
any(log_vec)
```

```
## [1] TRUE
```

```
all(log_vec)
```

```
## [1] FALSE
```

The Recycling Rule for Vectors

When mixing vectors of different sizes in a calculation, the shorter vector will be replicated (recycled) to match the longer vector's length. If the length of the longer vector's length is not an integer multiplicative of the shorter vector, the recyclying will still be performed, but a warning will be given:

```r
(x <- rep(1,6))
```

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

```r
x + 1      #similar to x+rep(1,6)
```

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

```r
x + 1:3    #similar to x+rep(1:3,2)
```

```
## [1] 2 3 4 2 3 4
```

```r
x + 1:4
```

```
## Warning in x + 1:4: longer object length is not a multiple of shorter object
```

```
## [1] 2 3 4 5 2 3
```

Exercises - Vectors

1. Create a vector with only the odd numbers from 1 to 15 using: i) the c() function, ii) the seq() function, iii) the operator : and arithmetic operations.
2. Calculate the the mean and standard deviation of the square root of the numbers 1 to 100 excluding 33.
3. Create vector with the values 1 to 10. Create a second vector with the same length whose values are all 5. Compute the vectors with the memberwise sum, product and max of the two vectors.
4. How would you remove all negative numbers from a vector? set all negtive numbers to zero?
5. How would you remove all NA values from a vector?
6. How would you reverse the order of a vector?
7. How would you sort a vector in a descending order? (look at the help for sort())

Factors

Factors are vectors that can hold just a number of defined values. They are helpful in representing qualitative data, in particular as part of a data frame (later in the presentation). Factors can be non-ordinal (male, female) or ordinal (small,medium,large).

```
(sex <- factor(c('F','F','M','F')))
```

```
## [1] F F M F
## Levels: F M
```

```
(sizes <- ordered(c('large','large','small','medium','small'),
                  levels=c('small','medium','large','huge')))
```

```
## [1] large  large  small  medium small
## Levels: small < medium < large < huge
```

Factor Attributes

Use levels(f) to obtain the levels of factor f. nlevels(f) returns the number of levels in f. Use table(f) or summary(f) to obtain a count of the number of instances for each possible level in f.

```
levels(sex)
```

```
## [1] "F" "M"
```

```
table(sex)
```

```
## sex
## F M
## 3 1
```

```
summary(sizes)
```

```
##  small medium  large   huge
##      2      1      2      0
```

Converting a Numeric Vector into a Factor

A numeric vector can be coerced into a factor using the as.factor() function. The levels() method can be used to replace the values with new labels:

```
num_vec <- c(1,2,1,2,3,2,1,2)
fact <- as.factor(num_vec)
fact
```

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

```
levels(fact) <- c('yes','no','maybe')
fact
```

```
## [1] yes   no    yes   no    maybe no    yes   no
## Levels: yes no maybe
```

Converting a Numeric Vector into a Factor

Sometimes we want to convert quantitative data that can hold a range of numeric values into qualitative data, by dividing the range of values into several categories. For this task we can use the cut() function. The breaks argument defines the categories. If a values is not inside one of the defined categories it receives the value NA:

```
ages <- c(4,99,18,1.2,44,4,66,-1);
(age_groups <- cut(ages,breaks=c(0,18,65,Inf),
                   labels=c('child','adult','elderly')))
```

```
## [1] child   elderly child   child   adult   child   elderly <NA>
## Levels: child adult elderly
```

Matrices

A matrix enhances the vector concept to 2-D. We can create a matrix by reshaping a vector into a matrix form. By default the vector is unfolded into a matrix column after column unless it is specified to do it by rows. When printing a matrix the numbers inside [] on top show the columns indices and the numbers inside [] on the left show the rows indices:

```
matrix(1:6,2,3)               #unfolded by columns
```

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

```
matrix(1:6,2,3,byrow=T)       #unfolded by rows
```

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

Matrix Dimensions

In place of the length attribute a matrix is defined by the number of rows and columns:

```
(mat <- matrix(1:6,2,3))
```

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

```
nrow(mat)
```

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

```
ncol(mat)
```

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

```
dim(mat)
```

```
## [1] 2 3
```

Accessing Matrix Members

As with a vector, the [ ] operator is used to access and select specific matrix members, only this time inside the operator two numbers are given separated by comma - the first number indicating the row index(indices) and the second indicating the column index(indices):

```
(mat <- matrix(c('a','b','c','d'),2,2))
```

```
##      [,1] [,2]
## [1,] "a"  "c"
## [2,] "b"  "d"
```

```
mat[1,2]
```

```
## [1] "c"
```

```
mat[2,1:2]
```

```
## [1] "b" "d"
```

Accessing Matrix Members

Leaving out the matrix rows/columns index inside the [ ] operator indicates selecting all the rows/columns:

```
(mat <- matrix(1:12,3,4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
mat[1,]
```

```
## [1]  1  4  7 10
```

```
mat[,2]
```

```
## [1] 4 5 6
```

cbind() and rbind()

The cbind() and rbind() functions can be used to combine two matrices together. cbind() combines the matrices along the columns (the matrices must have the same number of rows). rbind() combines them along the rows (the matrices must have the same number of columns).

```r
x <- matrix(1:4,2,2);
y <- matrix(5:8,2,2);
cbind(x,y)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

```r
rbind(x,y)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [3,]    5    7
## [4,]    6    8
```

Arithmetic Operations with Matrices

Basic arithmetic operations with a matrix operate on each member of the matrix. To perform matrix multiplication use the operator %*%:

```
x
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
x^2 + 2*x - 1
```

```
##      [,1] [,2]
## [1,]    2   14
## [2,]    7   23
```

```
x%*%x
```

```
##      [,1] [,2]
## [1,]    7   15
## [2,]   10   22
```

Matrix Algebra Functions

Some examples of functions used with matrix algebra:

```r
x
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```r
t(x)        #transpose
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```r
diag(x)     #diagonal
```

```
## [1] 1 4
```

```r
det(x)      #determinant
```

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

Calculations with Matrices

Using calculating functions with a matrix operate on the matrix as a whole, as if it was a vector:

```
x
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
max(x)
```

```
## [1] 4
```

```
sum(x)
```

```
## [1] 10
```

```
mean(x)
```

```
## [1] 2.5
```

The apply() Function

To perform a calculation on each row/column of a matrix use the apply() function. The function receives three arguments - the matrix, the dimension on which to perform the calculation (1 for rows, 2 for columns), the calculating function to perform:

```
(x <- matrix(1:6,2,3))
```

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

```
apply(x,1,sum)  # calculate the sum of each row
```

```
## [1]  9 12
```

```
apply(x,2,sum)  # calculate the sum of each column
```

```
## [1]  3  7 11
```

```
apply(x,2,mean) # calculate the mean of each column
```

```
## [1] 1.5 3.5 5.5
```

“Vectorizing” a Matrix

Use the as.vector(x) or c(x) functions to coerce a matrix x into a vector. The functions transform the matrix into a vector by columns. To do so by rows - transform the matrix first:

```
x
```

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

```
as.vector(x)
```

```
## [1] 1 2 3 4 5 6
```

```
c(t(x))
```

```
## [1] 1 3 5 2 4 6
```

Arrays

An array is a generalization of a matrix from 2-D to n-D. Arrays are constructed and manipulated in a similar way to matrices:

```
(x <- array(1:16,c(2,4,2)))  # a 3-D array with dimensions 2-4-2
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
```

```
x[1,3,2]        # accessing a selected index in the array
```

```
## [1] 13
```

```
apply(x,3,sum)  # sum of the elements along the 3rd dimension of the array
```

```
## [1]  36 100
```

Exercises - Matrices and Arrays

1. Create matrix x - a 5x5 matrix whose columns are the number 1:5. Create matrix y - 5x5 matrix whose columns are the number 1:5 in two ways: i) using the matrix() function and ii) by transposing the first matrix
2. Compute the memberwise product of x and y and the matrix product of x and y.
3. Compute memberwise sum of the second row of x with the third column of y.
4. Combine the two matrices into one so that y is to the right of x. Combine the matrices so that y is below x.
5. Compute the sum of each row of the matrix x using apply(). Compute the product of each column in y.

Lists

So far we have seen atomic variables or collections of atomic variables of one type. Lists can hold together objects of different types. Each member of a list can be an atomic variable, collection of atomic variables (vector/matrix/array) or another list (in a recursive manner).

```
(myList <- list(c(T,F,T), 'abc', matrix(1:4,2,2)))
```

```
## [[1]]
## [1]  TRUE FALSE  TRUE
##
## [[2]]
## [1] "abc"
##
## [[3]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
length(myList)
```

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

Naming List Members

List members can be given names at the construction of the list. The names of the list members can be obtained using the names() function. The names function can also be used to set the names of a list members if not given in the construction of the list (see next slide for example).

```
addr <- list(Name=c("Dan","Ran"),
             Street=c("Kaplan","Pinkas"),
             StreetNum=c(144,33));
names(addr)
```

```
## [1] "Name"      "Street"    "StreetNum"
```

```
addr
```

```
## $Name
## [1] "Dan" "Ran"
##
## $Street
## [1] "Kaplan" "Pinkas"
##
## $StreetNum
## [1] 144  33
```

Accessing List Members

Accessing list members can be done using the operator [[]] with the index of the list member or using the operator $ with the name of the list member - in case it has a name:

```
addr <- list(c("Dan","Ran"),c("Kaplan","Pinkas"),c(144,33));
names(addr) <- c("Name","Street","StreetNum");
addr[[1]]
```

```
## [1] "Dan" "Ran"
```

```
addr$Street
```

```
## [1] "Kaplan" "Pinkas"
```

```
addr$StreetNum[2]
```

```
## [1] 33
```

Extracting a Sub-List

We can extract a sub-list out of the list using the [ ] operator with the selected indices:

```
addr[2:3] # returns a list with two members
```

```
## $Street
## [1] "Kaplan" "Pinkas"
##
## $StreetNum
## [1] 144  33
```

```
addr[1]   # returns a list with one member
```

```
## $Name
## [1] "Dan" "Ran"
```

```
addr[[1]] # returns just the vector of names
```

```
## [1] "Dan" "Ran"
```

Adding a Member to a List

New members can be added to a list by setting values to the list using a new member name, or by concatenating a new member using the c() function (can also be used to concatenate another list):

```r
addr$AptNum <- c(7,1);
(addr <- c(addr,LastUpdated=date()))
```

```
## $Name
## [1] "Dan" "Ran"
##
## $Street
## [1] "Kaplan" "Pinkas"
##
## $StreetNum
## [1] 144  33
##
## $AptNum
## [1] 7 1
##
## $LastUpdated
## [1] "Wed Aug 02 12:41:15 2017"
```

unlist()

The unlist() function is used to create a single vector out of the list members.

```
(myList <- list(matrix(1:4,2,2), F, c("hi","bye")))
```

```
## [[1]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## [[2]]
## [1] FALSE
##
## [[3]]
## [1] "hi"  "bye"
```

```
unlist(myList)
```

```
## [1] "1"     "2"     "3"     "4"     "FALSE" "hi"    "bye"
```

Since vectors can only be of a single type, the list members will be coerced into a single type. In this case - since the list included strings, all objects in the returned vector were coerced into strings.

lapply()

The lapply() function is used to perform a function on each member of a list (similar to how apply() performs a function on each row/column of a matrix). It returns a new list with the results of the operations:

```
(myList <- list(1:10,11:20))
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
(sumList <- lapply(myList,sum))
```

```
## [[1]]
## [1] 55
##
## [[2]]
## [1] 155
```

```
unlist(sumList)
```

```
## [1]  55 155
```

Exercises - Lists

1. Create a list with 3 members: 3 band names, 2 tv series, the number pi. Give a name to each member.
2. Rename the list's members and print the names of the list members.
3. Extract the list of the band names as another list and as a vector. Check in each case whether extracted collection is a list or not.
4. Replace the third band with a new band.
5. Add another tv series to the two in the list.
6. Remove the third member (the number pi) from the list.

Data Frames

A data frame is a special type of list, where each member of the list is a vector, and all the vectors are of the same length (but not necesarily of the same type). Essentially, a data frame is a table (like in an excel sheet). It is the typical way a data set is stored and handled, where each row would represent an observation and each column would represent a particular variable.

```r
vec1 = c('John','Jim','Jack','Jil','Jane')
vec2 = c('M','M','M','F','F')
vec3 = c(80,100,70,65,77)
vec4 = c(170, 180, 170, 160, 172)
(df <- data.frame(Name=vec1,Sex=vec2,Weight=vec3,Height=vec4))
```

```
##   Name Sex Weight Height
## 1 John   M     80    170
## 2  Jim   M    100    180
## 3 Jack   M     70    170
## 4  Jil   F     65    160
## 5 Jane   F     77    172
```

Adding a New Variable to a Data Frame

All the features and methods related to a list apply also to a data frame because it is a list.

```
# adding new variable BMI
df$BMI <- round(df$Weight/((df$Height/100)^2),2)
df
```

```
##   Name Sex Weight Height   BMI
## 1 John   M     80    170 27.68
## 2  Jim   M    100    180 30.86
## 3 Jack   M     70    170 24.22
## 4  Jil   F     65    160 25.39
## 5 Jane   F     77    172 26.03
```

```
dim(df)  # The dimensions of the data frame
```

```
## [1] 5 5
```

Extracting/Removing Observations/Variables from a Data Frame

```
df[1,]               # extracting the first observation (row)
```

```
##   Name Sex Weight Height   BMI
## 1 John   M     80    170 27.68
```

```
(df2 <- df[-2,])     # removing the 2nd observation (row)
```

```
##   Name Sex Weight Height   BMI
## 1 John   M     80    170 27.68
## 3 Jack   M     70    170 24.22
## 4  Jil   F     65    160 25.39
## 5 Jane   F     77    172 26.03
```

```
(df2 <- df[,-5])     # removing the 5th variable (column)
```

```
##   Name Sex Weight Height
## 1 John   M     80    170
## 2  Jim   M    100    180
## 3 Jack   M     70    170
## 4  Jil   F     65    160
## 5 Jane   F     77    172
```

Selecting a Subset of the Data Frame

```
df
```

```
##   Name Sex Weight Height   BMI
## 1 John   M     80    170 27.68
## 2  Jim   M    100    180 30.86
## 3 Jack   M     70    170 24.22
## 4  Jil   F     65    160 25.39
## 5 Jane   F     77    172 26.03
```

```
(df2 <- df[df$Weight>75,]) # selecting rows whose weight > 75
```

```
##   Name Sex Weight Height   BMI
## 1 John   M     80    170 27.68
## 2  Jim   M    100    180 30.86
## 5 Jane   F     77    172 26.03
```

Selecting a Subset of the Data Frame

The subset() function can also be used to select a specific subset of a data frame.

```
subset(df,subset=df$Sex=='F')       #selecting only females
```

```
##   Name Sex Weight Height   BMI
## 4  Jil   F     65    160 25.39
## 5 Jane   F     77    172 26.03
```

```
subset(df,select=c(Height,Weight)) #selecting Weight and Height
```

```
##   Height Weight
## 1    170     80
## 2    180    100
## 3    170     70
## 4    160     65
## 5    172     77
```

Splitting a Data Frame

The split() function can be we used to split a data frame into several parts according to some condition. It returns a list, where each component is a data frame containing part of the original data frame:

```
split(df,df$Sex)  #splitting the data frame into two by Sex
```

```
## $F
##   Name Sex Weight Height   BMI
## 4  Jil   F     65    160 25.39
## 5 Jane   F     77    172 26.03
##
## $M
##   Name Sex Weight Height   BMI
## 1 John   M     80    170 27.68
## 2  Jim   M    100    180 30.86
## 3 Jack   M     70    170 24.22
```

The attach()/detach() Functions

Use the attach(df) function with data frame df in order to access df variables using their names without specifying they belong to df:

```
df
```

```
##   Name Sex Weight Height   BMI
## 1 John   M     80    170 27.68
## 2  Jim   M    100    180 30.86
## 3 Jack   M     70    170 24.22
## 4  Jil   F     65    160 25.39
## 5 Jane   F     77    172 26.03
```

```
df$Weight  # Weight cannot be used by itself only in the df context
```

```
## [1]  80 100  70  65  77
```

```
attach(df)
Weight     # Now Weight is defined globally
```

```
## [1]  80 100  70  65  77
```

Use detach(df) to remove df variables from the global scope.

Conversion Between a Matrix and a Data Frame

Use the function as.data.frame(x) to obtain a data frame from matrix x. Use as.matrix(df) to obtain a matrix from the data frame df. Since a matrix can only contain one type of atomic variable, all varaibles will be coerced into one type:

```
(df1 <- as.data.frame(matrix(1:4,2,2)))
```

```
##   V1 V2
## 1  1  3
## 2  2  4
```

```
(mat1 <- as.matrix(data.frame(v_1=c('a','b'),v_2=1:2)))
```

```
##      v_1 v_2
## [1,] "a" "1"
## [2,] "b" "2"
```

Note that the data frame is set with default var names V1..Vn. The matrix retains the variable names v_1..v_n for its columns. It is possible to set column/row names for a matrix using the dimnames() function.

Exporting/Reading Data to/from a File

Use write.table() or write.csv() functions to export a data frame into a text file. The text file will include a row for each observation with a special character (e.g. comma,space,tab) used to separate the columns. The write functions are given the data frame and the path to the file to be created, as well as some other optional parameters such as whether to save a header with the column names and what character to use to separate the columns.

```
write.table(df,'./Physical.txt',col.names=T,row.names=F,sep='\t');
```

Text files containing data in this format (whether exported from R or obtained from a different source) can be read into a data frame using the read.table() or read.csv() functions.

```
df <- read.table('./Physical.txt',header=T,sep='\t');
df
```

```
##   Name Sex Weight Height   BMI
## 1 John   M     80    170 27.68
## 2  Jim   M    100    180 30.86
## 3 Jack   M     70    170 24.22
## 4  Jil   F     65    160 25.39
## 5 Jane   F     77    172 26.03
```

Helpful Functions for Examining/Editing a Data Frame

Use head(df)/tail(df) to print just the first/last observations in a data frame. Use View(df) or fix(df) to open a window with the values of data frame df that can be manually edited. Use summary(df) with data frame df to present summary information regarding each variable of df. The summary function presents different information for different types of variables:

```
summary(df)
```

```
##    Name    Sex       Weight          Height           BMI       
##  Jack:1   F:2   Min.   : 65.0   Min.   :160.0   Min.   :24.22  
##  Jane:1   M:3   1st Qu.: 70.0   1st Qu.:170.0   1st Qu.:25.39  
##  Jil :1         Median : 77.0   Median :170.0   Median :26.03  
##  Jim :1         Mean   : 78.4   Mean   :170.4   Mean   :26.84  
##  John:1         3rd Qu.: 80.0   3rd Qu.:172.0   3rd Qu.:27.68  
##                 Max.   :100.0   Max.   :180.0   Max.   :30.86  
```

Setting the Type of a Variable

As can be seen by the printout of the summary call in the previous slide, character vectors are automatically treated as factors when inserted into a data frame. This makes sense for the Sex variable but does not makes sense for the Name variable (there is no pre-defined number of possible names). We can coerce a variable to a particular type using one of the as.<type>() functions. In this case we can coerce the Name variable to a character type:

```r
df$Name <- as.character(df$Name)
summary(df)
```

```
##      Name               Sex       Weight           Height           BMI
##  Length:5           F:2   Min.   : 65.0   Min.   :160.0   Min.   :24.22
##  Class :character   M:3   1st Qu.: 70.0   1st Qu.:170.0   1st Qu.:25.39
##  Mode  :character         Median : 77.0   Median :170.0   Median :26.03
##                           Mean   : 78.4   Mean   :170.4   Mean   :26.84
##                           3rd Qu.: 80.0   3rd Qu.:172.0   3rd Qu.:27.68
##                           Max.   :100.0   Max.   :180.0   Max.   :30.86
```

Exercises - Data Frames

1. The iris data set is part of the base R package. Try out the following commands: a) ?iris, b) iris, c) summary(iris), d) fix(iris) e) is.data.frame(iris), f) is.list(iris), g) is.matrix(iris), h) dim(iris).
2. Based on the result of the summary() command - what is the type of each variable in iris? Use is.numeric() and is.factor() with the variables to verify your answer.
3. Try printing one of the variables of iris without using the iris prefix. Now try the command attach(iris) and see what happens.
4. Add a new variable to iris called Petal.Color assigning a different color to each specie. Make sure the new variable is a factor.
5. Select a subset of the iris data with the observations in which the Sepal.Length>5 and Sepal.Width>3.
6. Save the iris data to a file called iris.csv on your working directory, where the variables are separated by a comma. Make sure the file was saved properly. Read the file into a new object called iris2. Compare the summary of iris and iris2 to see that they are the same.

Control Flows

R uses similar control flows elements as most programming languages. It includes two basic components:

- Conditions
  - if
  - if . . . else . . .
  - ifelse
  - switch
- Loops
  - for
  - while
  - repeat

Conditions - if Statement

Use an **if** statement to perform a set of operations only under a certain condition being true. Its syntax is:

```
if(condition) {
    expr1
    expr2
}
```

The curled brackets group together all the expression we want to be performed under the **if** clause. If there is only one expression to be performed under the **if** clause then the brackets are not necessary but it is good practice to always use them.

Conditions - if . . . else . . . Statement

Use an **if. . . else. . .** statement to perform one set of operations if a certain condition is true and another set if it is not true. It is possible to chain together several **if. . . else. . .** statements in order to handle multiple conditions scenario:

```
if(condition1) {
    expr1
    expr2
}
else if(condition2) {
    expr3
    expr4
}
else {
    expr5
    expr6
}
```

There is a shorter notation in case there are just two options to select from, with a simple operation to perform in each case. If the condition is true expr1 will be done and if not expr2 will be performed:

```
ifelse(condition, expr1, expr2);
```

Conditions - switch statement

You can use a **switch** statement instead of multiple **if...else...** statements.

switch(expr,
    val1=$\{$expr1$\}$,
    val2=$\{$expr2$\}$,
    ...
    valn=$\{$exprn$\}$
)

```
x <- 'b'
switch(x,
       'a' = { y <- 10 },
       'b' = { y <- 20 },
       'c' = { y <- 30 })
y
```

```
## [1] 20
```

Conditions - & and | versus && and ||

The condition used within an **if** statement can be a complex logical expression. The operators & and | signifying AND and OR which were discussed before have another version which is && and ||. The difference between the two versions is in expressions such as v1 AND/OR v2 where v1 or v2 are vectors. The &/| operators will operate on each member of the vector and return a logical vector while the &&/|| operators will return a single TRUE/FALSE value which is essentially any(v1) AND/OR any(v2).

```
x <- c(1,2)
y <- c(2,2)
(x == 1) & (y==2)
```

```
## [1]  TRUE FALSE
```

```
(x == 1) && (y==2)
```

```
## [1] TRUE
```

```
(x == 1) | (y==2)
```

```
## [1] TRUE TRUE
```

```
(x == 1) || (y==2)
```

```
## [1] TRUE
```

Loops - for Statement

We use a loop in order to repeat a certain piece of code with a different parameter each time. The **for** statement is the most commonly used loop statement. Its syntax is:

for(var in vector) {
expr1
expr2
}

The variable var is incremented with each loop to hold the next member of the vector, until reaching the end of the vector. For example:

```
for (i in 1:3) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

```
for (fruit in c('apple','banana','melon')) {
  print(fruit)
}
```

```
## [1] "apple"
## [1] "banana"
## [1] "melon"
```

Loops - while Statement

The **while** statement is another way to implement a loop. Its syntax is:

while(condition) {
   expr1
   expr2
}

In this case, the code within the **while** brackets is repeated until the condition is met:

```r
i <- 3
while (i > 0) {
  print(i)
  i <- i-1
}
```

```
## [1] 3
## [1] 2
## [1] 1
```

Loops - repeat Statement

A third way to implement a loop in R is using the **repeat** statement whose syntax is:

```
repeat {
   expr1
   expr2
}
```

It will essentially repeat the code within the brackets forever. In order to exit the loop we need to have an **if** statement within the loop that checks for a certain condition and if it occurs - calls the command **break** - which will exit the loop.

```r
i <- 3
repeat {
  print(i)
  i <- i-1
  if(i<=0) {
    break;
  }
}
```

```
## [1] 3
## [1] 2
## [1] 1
```

Loops - break and next

The **break** command seen in the previous example within the **repeat** statement can also be used in the same manner within a **for** or a **while** statement. The **next** command can be used within a loop to skip the rest of the code for the current iteration and move on to the next iteration of the loop.

```
for (fruit in c('apple','banana','melon','peach')) {
  if(fruit == 'melon') {
    next;
  }
  print(fruit)
}
```

```
## [1] "apple"
## [1] "banana"
## [1] "peach"
```

However the use of the commands **break** and **next** is not considered good coding and it is best to avoid them by using a different implementation to achieve the same results.

Alternatives to Loops

Loops can be costly in terms of run time, and in R there are many instances we can avoid them using vector calculations. For example, suppose we want to calculate the sum of squares of all members of a vector, we can do it in two ways:

```r
vec <- 1:100;
#option 1 - calculating ssq using a loop
ssq1 <- 0
for (i in vec) {
  ssq1 <- ssq1 + i^2;
}
ssq1
```

```
## [1] 338350
```

```r
#option 2 - calculating ssq using vector calculations
ssq2 <- sum(vec^2); ssq2
```

```
## [1] 338350
```

The second option is more efficient in both code length and run time, which will become more significant as the length of the vector increases. We can also use apply()/lapply() functions to perform operations on vectors instead of loops.

Functions in R

- Functions in R are similar to functions in most programming languages. We already seen the use of many pre-defined functions in R but did not formally explain what a function is.

Functions in R

- Functions in R are similar to functions in most programming languages. We already seen the use of many pre-defined functions in R but did not formally explain what a function is.
- A function is an encapsulation of some code performing an operation which we would want to call many times. Instead of writing the same code every time we want to use it, we define it in a function once and then call the function to perform the operation each time we want it done.

Functions in R

- Functions in R are similar to functions in most programming languages. We already seen the use of many pre-defined functions in R but did not formally explain what a function is.

- A function is an encapsulation of some code performing an operation which we would want to call many times. Instead of writing the same code every time we want to use it, we define it in a function once and then call the function to perform the operation each time we want it done.

- A function can be called from within another function. The most basic functions in R, which are frequently used, are called primitive functions and are implementd in the C language in order to have a shorter run time.

Functions in R

- Functions in R are similar to functions in most programming languages. We already seen the use of many pre-defined functions in R but did not formally explain what a function is.
- A function is an encapsulation of some code performing an operation which we would want to call many times. Instead of writing the same code every time we want to use it, we define it in a function once and then call the function to perform the operation each time we want it done.
- A function can be called from within another function. The most basic functions in R, which are frequently used, are called primitive functions and are implementd in the C language in order to have a shorter run time.
- A function can be defined to receive any number of arguments (including none), which are its input. An input argument can be an atomic variable of any type or a collection of any type (vector,matrix,list. . .). Input arguments can be manatory or optional. An optional argument will have a default value defined for it.

Functions in R

- Functions in R are similar to functions in most programming languages. We already seen the use of many pre-defined functions in R but did not formally explain what a function is.
- A function is an encapsulation of some code performing an operation which we would want to call many times. Instead of writing the same code every time we want to use it, we define it in a function once and then call the function to perform the operation each time we want it done.
- A function can be called from within another function. The most basic functions in R, which are frequently used, are called primitive functions and are implementd in the C language in order to have a shorter run time.
- A function can be defined to receive any number of arguments (including none), which are its input. An input argument can be an atomic variable of any type or a collection of any type (vector,matrix,list. . .). Input arguments can be manatory or optional. An optional argument will have a default value defined for it.
- A function returns one object of any type, which is its output. The returned object can be a collection of some type which holds many values.

Functions in R

- Functions in R are similar to functions in most programming languages. We already seen the use of many pre-defined functions in R but did not formally explain what a function is.
- A function is an encapsulation of some code performing an operation which we would want to call many times. Instead of writing the same code every time we want to use it, we define it in a function once and then call the function to perform the operation each time we want it done.
- A function can be called from within another function. The most basic functions in R, which are frequently used, are called primitive functions and are implementd in the C language in order to have a shorter run time.
- A function can be defined to receive any number of arguments (including none), which are its input. An input argument can be an atomic variable of any type or a collection of any type (vector,matrix,list. . .). Input arguments can be manatory or optional. An optional argument will have a default value defined for it.
- A function returns one object of any type, which is its output. The returned object can be a collection of some type which holds many values.
- Functions in R are themselves R objects and can be passed as an argument to another function or returned as the return value of a function.

Calling Pre-defined Functions

Lets look for example on the pre-defined R function called **sample**. Writing ?sample in the RStudio console opens the help page on the **sample** function. According to the help page, "**sample** takes a sample of the specified size from the elements of **x**, using either with or without replacement".

The usage of **sample** is defined in the help page as follows:
sample(x, size, replace=FALSE, prob=NULL)

The function recieves 2-4 parameters (two mandatory and two optional):

- x - a vector from which the function will sample
- size - the number of samples required
- replace - whether or not to sample with replacement (default is no replacement)
- prob - probability weights for each element in the vector (default is uniform sampling)

```
(samples1 <- sample(1:10,10))
```

```
##  [1]  5  8  1  2  3  4 10  6  9  7
```

```
(samples2 <- sample(1:10,10,replace=T))
```

```
##  [1]  6  4  2  4  2  6  1 10  2  1
```

Calling Pre-defined Functions

We can change the order of the arguments given in a call to a function by explicity specifying the name of the arguments in the call:

```
sample(replace=T,size=5,x=c('apple','banana'))
```

```
## [1] "apple"  "banana" "apple"  "apple"  "apple"
```

A function can print a warning or can stop with an error in case it was given some illegal arguments. For example, with **sample** we cannot call the function with the arguments size>length(x) and replace=F:

```
sample(1:10,100)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample la
```

Some functions can recive no arguments. In this case when calling the function, we still need to have the parenthesis after the function name, with nothing inside them.

User-defined Functions

We can write our own functions in R and use them in the same manner pre-defined function are used. The syntax for defining a function looks like this:

```
function_name <- function(m_arg1,...,m_argN,o_arg1=val1,...,o_argN=valN)
{
   function_body
   return (return_value)
}
```

The arguments m_arg1...m_argN are mandatory arguments. The arguments o_arg1...o_argN are optional arguments with default values val1...valN. It is good practice to define the optional arguments after the mandatory arguments.

The call **return** can appear more than once within the function (see next slide). It can also not appear at all (not recommended) in which case the function will return the value of the last term that appeared in it.

In order to be able to use a newly defined function we need to load it into memory by running the code containing the function definition. In order for the function to be available the next time RStudio is started we need to save the workspace and load it at the start of the new session.

User-defined Functions - Example

As and example of a user-defined function, here is a definition of a function called **temp.converter** that converts tempratures between celsius and fahrenheit. By default, it converts from celsius to fahrenheit but can be used to convert the other way around.

```
temp.converter <- function(temp,to_fahrenheit=T)
{
  if(to_fahrenheit)
    return (temp*9/5 + 32)
  else
    return (5/9*(temp - 32))
}
temp.converter(40)
```

```
## [1] 104
```

```
temp.converter(104,to_fahrenheit=F)
```

```
## [1] 40
```

User-defined Functions - Error Handling

What happens if somebody tries to call our **temp.converter** with a non-numeric value as the **temp** argument? In the current implementation the mistake will be found out during the calculation and an error will be returned.

```
temp.converter('40C')
```

```
## Error in temp * 9: non-numeric argument to binary operator
```

In some cases we would like to check the arguments ourselves before using them and produce a warning or an error if something is not right. Here it is done in a new **temp.converter2** function that checks if **temp** is numeric, and calls stop() if not, to exit the function. If **temp** is numeric it calls the original **temp.converter** function:

```
temp.converter2 <- function(temp,to_fahrenheit=T)
{
  if(!is.numeric(temp))
    stop('temp must be of numeric type')

  return (temp.converter(temp,to_fahrenheit))
}
temp.converter2('40C')
```

```
## Error in temp.converter2("40C"): temp must be of numeric type
```

The Scope of Variables

Variables that were defined inside a function, will not be available outside of the function as their scope is limited to the function. Attempting to access such a variable will generate an error.

```r
print.y <- function()
{
  y <- 10
  print(y)
}
print.y()
```

```
## [1] 10
```

```r
y+5
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```

The Scope of Variables

If we define and set a value to a variable inside a function which has the same name as a variable outside of the function, it will not effect the variable outside of the function once the function was completed.

```
x <- 10;
add5 <- function(y)
{
  x <- 5
  return (x+y)
}
add5(7)
```

```
## [1] 12
```

```
x
```

```
## [1] 10
```

The Scope of Variables

However, R objects that were defined outside of a function, **are** available within the function (unlike the default behavior in most programming languages). If R cannot find a variable that is used within a function, it will search for it outside the scope of the function. If the function was called from another function, it will start looking for it in the scope of the calling function, and if it cannot find it there, it will look for it in the encompasing scope, until reaching the global scope.

Nevertheless, it is a very bad practice to use a variable within a function assuming it is defined outside the scope of the function, since at some point someone can call the function while this variable is not defined. Instead, define the variable as an argument to the function so that it must be passed to the function while calling it.

```r
x <- 10;
sum.xy <- function(y) { return (x+y) }
sum.xy(5)
```

```
## [1] 15
```

```r
rm(x)
sum.xy(5)
```

```
## Error in sum.xy(5): object 'x' not found
```

R Functions for Probability Distributions

As a programming language for statisticians, R has built-in functions for generating values from known distributions. For each supported distribution, there are 4 different functions in R starting with the letters ‘p’,‘q’,‘d’,‘r’:

- d for ‘density’ - the density function
- p for ‘probablity’ - the cumulative distribution function
- q for ‘quantile’ - the inverse of the cumulative distribution function
- r for ‘random’ - a random variable from the specified distribution

Here is a table with some notable distributions and the related R functions:

Distribution	Density	Probability	Quantile	Random
Beta	dbeta	pbeta	qbeta	rbeta
Binomial	dbinom	pbinom	qbinom	rbinom
Chi-Square	dchisq	pchisq	qchisq	rchisq
Exponential	dexp	pexp	qexp	rexp
Gamma	dgamma	pgamma	qgamma	rgamma
Negative-Binomial	dnbinom	pnbinom	qnbinom	rnbinom
Normal	dnorm	pnorm	qnorm	rnorm
Poisson	dpois	ppois	qpois	rpois
Student t	dt	pt	qt	rt
Uniform	dunif	punif	qunif	runif

R Functions for Probability Distributions

Here, for example, is the use of the four functions for the Normal distribution with mean=0 and s.d.=1 (the standard normal distribution):

```
dnorm(0,mean=0,sd=1)   # calculates the probability for x=0
```

```
## [1] 0.3989423
```

```
pnorm(0,mean=0,sd=1)   # calculates the c.d.f. for x=0
```

```
## [1] 0.5
```

```
qnorm(0.5,mean=0,sd=1) # calculates the inverse c.d.f. for x=0.5
```

```
## [1] 0
```

```
rnorm(5,mean=0,sd=1)   # generates 5 random numbers
```

```
## [1]  0.2643954 -1.2871555  0.2973873 -2.1364136 -0.9323639
```

Random Number Generation

We saw already two examples in R for functions that generate random values. We saw the **sample** function that generates random samples from a given vector with or without replacement, and we now saw the random ('r') functions of for the different distributions (e.g. **rnorm**) that generate random numbers from a given distribution.

As in all programming languages, the random number generating (RNG) functions use pseudo-random number generators. This means that the functions rely on the randomness of an intial seed value. With the same seed, calling an RNG function will result with the same sequence of random values each time. Typically, the seed is created the first time it is required using the current time and process ID. However, we can manually set the seed using the set.seed() function. This is useful for debugging a code that uses RNG because we can expect the same result each time we run the program and we can compare the results obtained from different implementations of the same code.

Random Number Generation

```r
set.seed(123)
# generates 5 random numbers
rnorm(5,mean=0,sd=1)
```

```
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
```

```r
# generates another 5 random numbers
rnorm(5,mean=0,sd=1)
```

```
## [1]  1.7150650  0.4609162 -1.2650612 -0.6868529 -0.4456620
```

```r
set.seed(123)
# generates 5 random numbers (same 5 as in first call)
rnorm(5,mean=0,sd=1)
```

```
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
```

```r
# generates another 5 random numbers (same 5 as in second call)
rnorm(5,mean=0,sd=1)
```

```
## [1]  1.7150650  0.4609162 -1.2650612 -0.6868529 -0.4456620
```

R Packages

- An R package is a collection of related R programs that enhance the functions of R.
- The R installation includes several basic R packages.
- Thousands of additional R packages on a variety of topics are available for free at the CRAN website.
- To download and install a package use 'Tools->Install Packages...' from the menu or the install.package() command.
- Packages evolve and may need to be updated. Updating packages can be done from the menu using 'Tool->Check for Package Updates...' or using the update.packages() command
- We will go over various packages in this course:

List of packages

- ggplot2 - create elegant data visualisations
- tidyr - easily tidy data with 'spread()' and 'gather()' functions
- dplyr - a fast, consistent tool for working with data frame like objects
- zoo - S3 infrastructure for regular and irregular time series
- caret - classification and regression training
- rmarkdown - dynamic documents for R
- shiny - web application framework for R

R Libraries

- An installed package is saved on the computer as a library.
- To use function f() that is part of library x - call x::f()
- To use a library functions without explicitly specifying the library it is necessary to load the library using the **library()** command (i.e., library(x)).
- A package needs to be installed only once, but the library needs to be loaded again each time RStudio is restarted.
- To see information on library x type - library(help='x')
- The 'Packages' pane shows the list of installed libraries. Loaded libraries have their checkbox marked. By checking/unchecking the box next to a library you can load/unload it.