Data Analysis/R Software Training Workshop in Taita Taveta University, Ngerenyi.

Fundamentals of R programming

Noah Cheruiyot Mutai

02 September, 2019

Outline

What is R and why R?

- R is a language and environment for statistical computing and graphics.
- Like SAS, STATA, JMP, etc
- Allows statistical programming
- Statistical programming allows automation
- It's free and open source
- Massive packages.
- Cutting edge tools
- Language support for data analysis
- Fantastic community
- Powerful tools for communicating results.

Setting up

R

- R can be downloaded from one of the mirror sites in; https://cran.r-project.org/bin/windows/base/
- Install R with the standard settings, since later on we will work with RStudio (IDE).

RStudio

- RStudio can be downloaded from; https://www.rstudio.com/products/rstudio/download/
- Download the Desktop Version for Linux, Mac or Windows.
- Install RStudio. (Make sure you already installed R.)
- Start RStudio.

The R studio interface.

- 4 windows
- upper left- code editor: type, edit and run code
- upper right-environment: all functions, objects, data, etc created in active session
- clear by pressing the brush
- history: list of commands
- run command again-to console
- to source

...

- delete command with red star
- lower left: console - see output
- lower right: File: access files in your computer
- All list of files in the folder
- Plots: show plots
- Packages: all installed packages, install and activate packages
- Help: search for help
- Viewer: display charts using special plotting packages e.g. ggv
- *NOTE:* The layout can be re-arranged in Tools -> Global options -> Pane layout.

Other Data editors or IDEs for R

- There are several text editors one can choose from.
- Here are a few;
  - Vim -https://www.vim.org/
  - Emacs -http://www.gnu.org/software/emacs/
  - Eclipse -http://www.eclipse.org/eclipse/
  - Sublime -http://www.sublimetext.com/
  - Tinn-R -<https://sourceforge.net/p/tinn-r/wiki/Home/

Packages in R

What is a package?

- An R package includes a set of functions and datasets which is not included in the ‘base’ R System.
- Packages provide additional functionality.
- An exhaustive list of available packages is on CRAN.

  http://cran.r-project.org/web/packages/
- There are also many packages associated with the Bioconductor project

  http://bioconductor.org.

Other places to get packages

```
- GitHub
- BitBucket
- rForge
- Your friends and collaborators
- Write them yourself
```

How to install a package?

- Use install.packages() command to install packages.
- Need to specify other options e.g. lib, repos, dep.
- Activating R packages use library() or require()
- Detach packages with detach(package = package)
- To get the contents of a package use library(help = package) e.g.

```
# help(package = stats)
# OR
# library(help = MASS)
```

Functions inside a package

- We can access functions in a package.

```
# lsf.str("package:stats")
```

- To list all objects in a package.

```
# ls("package:stats")
```

- To see the list of currently loaded package use:

```
# library(dplyr)
# search()
```

Which package for which kind of analysis?

- Under https://cran.r-project.org/, go to packages,
- then CRAN Task Views.
- You will see packages per topic.
- For example under Experimental Design.
    - e.g agricolae: Statistical Procedures for Agricultural Research
    - desplot: Plotting Field Plans for Agricultural Experiments
    - agridat: Agricultural Datasets

Other set up issues and commands

- Check working directory with;

```
# getwd()
```

- Set working directory with;

```
# setwd()
```

- or

```
# setwd(choose.dir())
```

...

- list the files in the working directory

```
# list.files()
# or dir()
```

- Use;

```
# ls()
```

to list all objects. - Clear work space before any analysis with;

```
# rm(list=ls())
```

- Modify appearance of R studio in global options.

To see all data available in R by default

```
# data()
```

- Data in a specific package;

```
# data(package = "agridat")
# data(package="MASS")
# data(package="emdi")
```

Getting help in R

- There are several ways of getting help in R.
- Type question mark followed by function e.g.

```
# ? read.table or help("read.table")
# find("read.table")
# apropos("lm")
# help.start()
# help()
# help.search()
# demo()
# example()
# library()
# vignette()
# browseVignettes()
```

Online help

- The is a tremendous amount of information about R on the web, but your first port of call is likely to be CRAN at;

  http://cran.r-project.org/

Worked examples of functions

```
- example("lm")
- example("glm")
```

Demonstration of R functions

```
- demo(graphics)
- demo(persp)
- demo(graphics)
- demo(Hershey)
- demo(plotmath)
```

Good house keeping

- To see what variables you have created in the current session, type;

```
# objects()
# ls()
```

- To see which libraries and data frames are attached:

```
# search()
```

...

- At the end of a session in R, it is good practice
- to remove (rm) any variables names you have created (using, say, x <- 5.6) and to detach any data frames you have attached earlier in the session
  - rm()
  - detach()
- To get rid of everything, including all the data frames, type; rm(list=ls())
- NOTE: This fucntion should be used very carefully as it clears literally everything.

Citing R

- The R Development Core Team has done a huge
- amount of work and we, the R user community,
- should pay them due credit whenever we publish
- work that has used R.

```
citation()
```

```
##
## To cite R in publications use:
##
##   R Core Team (2019). R: A language and environment for
##   statistical computing. R Foundation for Statistical Compu
##   Vienna, Austria. URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
```

Operations in R

Arithmetic operators

- These operators are used to carry out mathematical operations like addition and multiplication.

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2

Examples

```
x <- 5
y <- 16
x+y
```

```
## [1] 21
```

```
x-y
```

```
## [1] -11
```

```
x*y
```

```
## [1] 80
```

Examples

```r
y/x
```

```
## [1] 3.2
```

```r
y%/%x
```

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

```r
y%%x
```

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

```r
y^x
```

```
## [1] 1048576
```

Relational Operators

- Relational operators are used to compare between values. Here is a list of relational operators available in R.

Operator	Description
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	exactly equal to
`!=`	not equal to
`!x`	Not x

Examples

```r
x <- 5
y <- 16
x<y
```

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

```r
x>y
```

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

```r
x<=5
```

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

Examples

```r
y>=20
```

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

```r
y == 16
```

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

```r
x != 5
```

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

Logical Operators

- Logical operators are used to carry out Boolean operations like AND, OR etc.

Operator	Description
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
|	Element-wise logical OR
||	Logical OR

Examples

```
x <- c(TRUE,FALSE)
y <- c(FALSE,TRUE)
!x
```

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

```
x&y
```

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

Examples

```
x&&y
```

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

```
x|y
```

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

```
x||y
```

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

Assignment Operators

- These operators are used to assign values to variables.

Operator	Description
`<-, <<-, =`	Leftwards assignment
`->, ->>`	Rightwards assignment

- The operators <- and = can be used, almost interchangeably, to assign to variable in the same environment.
- <- is recommended.

Examples

```r
x <- 5;x
```

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

```r
x = 9; x
```

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

```r
10 -> x; x
```

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

Getting help on operators in R

- Put the operator in between quotes after question mark.

```
# ?"!"
# ?"&"
# ?"<-"
```

Some functions in R

- R has got many inbuilt mathematical and statistical fucntions.
- Here we highlight just a few of them.
- One can get a list of all built in fucntions in R by running the command

```
# builtins()
# run this without the comment
```

Numeric Functions

Function	Description
abs(x)	absolute value
sqrt(x)	square root
ceiling(x)	ceiling(3.475) is 4
floor(x)	floor(3.475) is 3
trunc(x)	trunc(5.99) is 5
round(x, digits=n)	round(3.475, digits=2) is 3.48
signif(x, digits=n)	signif(3.475, digits=2) is 3.5
cos(x), sin(x), tan(x)	also acos(x), cosh(x), acosh(x), etc.
log(x)	natural logarithm
log10(x)	common logarithm
exp(x)	e^x

Statistical functions

Function	Description
`mean(x)`	mean of object x
`sd(x)`	standard deviation of object(x).
`var(x)`	variance and
`mad(x)`	median absolute deviation.
`median(x)`	median
`quantile(x, probs)`	quantiles
`range(x)`	range
`sum(x)`	sum
`min(x)`	minimum
`max(x)`	maximum
`cor.test()`	correlation test
`cumsum() cumprod()`	cumuluative functions for vectors
`sample()`	random samples

...

- Check https://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf for a more comprehensive list of R operators and fucntions.

Style Guide in R

- The goal of style rules is to make R code easier to read, share and verify.
- There is no unique standard.
- File Names: end in .R (have a meaningful name)
- Identifiers: should be meaningful; R is case sensitive
- variables: lowerCamelCase(dataFrame, someData)

. . . style guide in R

- functions: lowerCamelCase(someFunction, functionName)
- constants: lowerCamelCase(i,j, meanOfSomething)
- Line Length: maximum 80 characters
- Indentation: two spaces, no tabs Spacing
- Curly Braces: first on same line, last on own line
- else: Surround else with braces
- Assignment: use <-, not =
- Semicolons: don't use them
- Commenting: all comments begin with # followed by a space.

...

- More information on style guide;

  1. http://jef.works/R-style-guide/
  2. http://adv-r.had.co.nz/Style.html
  3. https://github.com/rdatsci/PackagesInfo/wiki/R-Style-Guide
  4. https://google.github.io/styleguide/Rguide.xml
  5. https://style.tidyverse.org/index.html

Data structures in R

- There are five data types commonly used in analysis in R.
  - Atomic vectors
  - Matrices
  - Data frames
  - Lists; special vectors; different data types.
  - Factors
  - Arrays; not very common.

Atomic vectors

- The basic data structure in R is a vector.
- Six types of atomic vectors
  - numeric/double
  - integer
  - character
  - logical
  - Others; complex and raw.
- Three properties of a vector: typeof(), length(), attributes()
- All elements of an atomic vector must have same data type.
- Atomic vectors are created with c() operator.
- We use <- or = for assignment.

Atomic vectors

Numeric/Double

- These are basically numbers.

```
dbl_var <- c(1, 2.5, 4.5)
class(dbl_var)
```

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

Integer

- Specify the L suffix to get an integer

```
int_var <- c(1L, 6L, 10L)
class(int_var)
```

```
## [1] "integer"
```

Atomic vectors

Logical

- Use TRUE and FALSE (or T and F) to create logical vectors

```
log_var <- c(TRUE, FALSE, T, F)
class(log_var)
```

```
## [1] "logical"
```

Character

```
chr_var <- c("these are", "some strings")
class(chr_var)
```

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

Complex and raw

- Doubles, integers, characters, and logicals are the most common types of atomic vectors in R, but R also recognizes two more types: complex and raw.
- It is doubtful that you will ever use these to analyze data, but here they are for the sake of thoroughness.
- Complex vectors store complex numbers.
- To create a complex vector, add an imaginary term to a number with i:

```
comp <- c(1 + 1i, 1 + 2i, 1 + 3i)
class(comp)
```

```
## [1] "complex"
```

Raw

- Raw vectors store raw bytes of data.
- Making raw vectors gets complicated, but you can make an empty raw vector of length n with raw(n).
- See the help page of raw for more options when working with this type of data;

```
raw(3)
```

```
## [1] 00 00 00
```

```
class(raw(3))
```

```
## [1] "raw"
```

Dates and Times

- The attribute system lets R represent more types of data than just doubles, integers, characters, logicals, complexes, and raws.
- The time looks like a character string when you display it, but its data type is actually "double", and its class is "POSIXct" "POSIXt" (it has two classes):

```
now <- Sys.time()
now
```

```
## [1] "2019-09-02 11:02:48 EAT"
```

```
class(now)
```

```
## [1] "POSIXct" "POSIXt"
```

....

- POSIXct is a widely used framework for representing dates and times.
- In the POSIXct framework, each time is represented by the number of seconds that have passed between the time and 12:00 AM January 1st 1970 (in the Universal Time Coordinated (UTC) zone).
- For example, the time above occurs 1,395,057,600 seconds after then.
- So in the POSIXct system, the time would be saved as 1395057600.

...

- R creates the time object by building a double vector with one element, 1553527480.
- You can see this vector by removing the class attribute of now, or by using the unclass function, which does the same thing:

```
unclass(now)
```

```
## [1] 1567411369
```

More information on dates and times in R on;
https://www.r-bloggers.com/using-dates-and-times-in-r/

There is also a course by Charlotte Wickham on;
https://www.datacamp.com/courses/working-with-dates-and-times-in-r?tap_a=5644-dce66f&tap_s=10907-287229

Coercion in R

- Objects can be explicitly coerced from one class to another using as. functions, if available
- R's coercion behavior may seem inconvenient, but it is not arbitrary.
- R always follows the same rules when it coerces data types.
- Once you are familiar with these rules, you can use R's coercion behavior to do surprisingly useful things.

So how does R coerce data types?

- If a character string is present in an atomic vector, R will convert everything else in the vector to *character strings*.
- If a vector only contains logicals and numbers, R will convert the logicals to numbers; every TRUE becomes a 1, and every FALSE becomes a 0.
- You can convert a factor to a character string with the as.character function.
- R will retain the display version of the factor, not the integers stored in memory:

...

```r
x<- 0:6
x
```

```
## [1] 0 1 2 3 4 5 6
```

```r
class(x)
```

```
## [1] "integer"
```

```r
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```r
as.logical(x)
```

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

```r
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```r
as.complex(x)
```

```
## [1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Nonsensical coercion

Nonsensical coercion results in NAs e.g.

```
x<- c("a","b","c","d")
as.numeric(x)
```

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

```
as.logical(x)
```

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

Matrices

- A matrix is a collection of data elements of the same type.
- Arranged in a two-dimensional rectangle.
- To create a matrix we must indicate the elements,
- As well as the number of rows (nrow) and columns (ncol)

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

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

- By default, any matrix is created column-wise
- To change that we set the byrow option to TRUE

Matrices

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

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

```r
#or
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

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

Matrices

- It is not necessary to specify both the number of rows and columns.
- The number of elements must be a multiple of the number of rows or columns.

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

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

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

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

Marices

- To get the matrix dimensions

```r
dim(m)
```

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

```r
nrow(m)
```

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

```r
ncol(m)
```

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

Creating matrices with rbind() and cbind()

- rbind() and cbind() allow us to bind vectors
- in order to create a matrix
- these vectors must have the same length

```
x <- c(1,2,3,4)
y <- c(10,11,12,13)
z <- c(20,30,40,50)
```

- if we use rbind(), our vectors will be rows

```
m <- rbind(x, y, z); m
```

```
##   [,1] [,2] [,3] [,4]
## x    1    2    3    4
## y   10   11   12   13
## z   20   30   40   50
```

Any order

- we can bind the vectors in any order

```
m <- rbind(y, z, x) ;m
```

```
##   [,1] [,2] [,3] [,4]
## y   10   11   12   13
## z   20   30   40   50
## x    1    2    3    4
```

- we can also bind the same vector several times.

```
m <- rbind(x, y, x, z); m
```

```
##   [,1] [,2] [,3] [,4]
## x    1    2    3    4
## y   10   11   12   13
## x    1    2    3    4
## z   20   30   40   50
```

Rbind() function directly.

- it is not necessary to create the vectors first
- we can enter them directly in the rbind() function

```
m <- rbind(c(1, 2, 3), c(7, 8, 9), c(21, 22, 23))
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    7    8    9
## [3,]   21   22   23
```

Columns from cbind

- if we use cbind() the vectors will be columns

```
m <- cbind(x, y, z)
m
```

```
##      x  y  z
## [1,] 1 10 20
## [2,] 2 11 30
## [3,] 3 12 40
## [4,] 4 13 50
```

```
class(m)
```

```
## [1] "matrix"
```

Naming matrix rows and columns

- we can name rows and columns when we create the matrix
- using the dimnames option in the matrix() function

```
m <- matrix(c(1,2,3,4,5,6), nrow = 2,
            dimnames = list(c("row1", "row2"),
                            c("col1", "col2", "col3")))
m
```

```
##      col1 col2 col3
## row1    1    3    5
## row2    2    4    6
```

Another example

```r
m <- matrix(c(1,2,3,4,5,6), nrow = 2,
            dimnames = list(c("goats", "cows"),
                            c("a", "b", "c")))
m
```

```
##       a b c
## goats 1 3 5
## cows  2 4 6
```

```r
# ?dimnames
```

Rownames() and colnames()

- alternatively, we can name rows and columns at any time
- after creating the matrix
- using the functions rownames() and colnames()

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

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

Rownames and column names

```
rownames(m) <- c("row1", "row2")
m
```

```
##      [,1] [,2] [,3]
## row1    1    3    5
## row2    2    4    6
```

```
colnames(m) <- c("col1", "col2", "col3")
m
```

```
##      col1 col2 col3
## row1    1    3    5
## row2    2    4    6
```

Removing names

- Assign to NULL

```r
rownames(m) <- NULL
m
```

```
##      col1 col2 col3
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
colnames(m) <- NULL
m
```

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

Indexing matrices

- indexing means accessing one or several matrix elements
- indices must be put between square brackets
- we must use two indices: one for the row
- and the other one for the column

```
m <- matrix(1:16, nrow = 4, byrow = TRUE)
m
```

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

Element on row 2, column 3

- access the element on row 2, column 3

```
m[2,3]
```

```
## [1] 7
```

- access the element on row 4, column 1

```
m[4,1]
```

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

- access the element on row 2, column 2
- and the element on row 4, column 3

```r
c(m[2,2], m[4,3])
```

```
## [1]  6 15
```

- access the row 2

```r
m[2,]
```

```
## [1] 5 6 7 8
```

Further examples

- we can put the elements in this row in a vector

```
x <- m[2,] ; x
```

```
## [1] 5 6 7 8
```

- access column 3

```
m[,3]
```

```
## [1]  3  7 11 15
```

- access the elements on row 2, columns 2,3 and 4

```
m[2,2:4]
```

```
## [1] 6 7 8
```

...

- access the elements on column3, rows 1 and 4

```r
m[c(1,4),3]
```

```
## [1]  3 15
```

- access the elements on rows 2 and 4, columns 2 and 4

```r
m[c(2,4), c(2,4)]
```

```
##      [,1] [,2]
## [1,]    6    8
## [2,]   14   16
```

...

- access the elements on rows 2, 3 and 4, columns 3 and 4

```
m[2:4, 3:4]
```

```
##      [,1] [,2]
## [1,]    7    8
## [2,]   11   12
## [3,]   15   16
```

- access the elements at the intersection of rows 1 and 2 with columns 1 and 2

- and the elements at the intersection of rows 3 and 4 with columns 3 and 4 (the result will be a vector)

```
c(m[1:2, 1:2], m[3:4, 3:4])
```

```
## [1]  1  5  2  6 11 15 12 16
```

...

- Access the fifth element, in column-wise order.

```
m[5]
```

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

- access the fifth and the seventh element, in column-wise order.

```
m[c(5, 7)]
```

```
## [1]  2 10
```

- access the fifth, the sixth and the seventh element, in column-wise order.

```
m[5:7]
```

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

Remove elements

- to remove elements we use negative indices
- access the row 2 less the element on the third column

```
m
```

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

```
m[2, -3]
```

```
## [1] 5 6 8
```

Some more

- access the row 4 less the elements on the second and fourth column

```
m[4, c(-2, -4)]
```

```
## [1] 13 15
```

- access the rows 2, 3 and 4 less the element on the first column

```
m[2:4,-1]
```

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

Filtering matrices

- filtering means accessing elements that meet a certain condition
- this condition must be put between square brackets
- create a 4x4 matrix of 16 discrete random numbers between 1 to 100

```
x <- sample(100, 16, replace = TRUE)
x
```

```
##  [1] 73  6 13 41 24 57 68 97  3  3 45 60 92 21 61 47
```

Another matrix

```
m <- matrix(sample(100, 16, replace = TRUE),
            nrow = 4, byrow = TRUE);m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   12   13   26   33
## [2,]    9   28   22   30
## [3,]   34   70   35   49
## [4,]   96   44   95   10
```

```
m <- matrix(x, nrow = 4, byrow = TRUE);m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   73    6   13   41
## [2,]   24   57   68   97
## [3,]    3    3   45   60
## [4,]   92   21   61   47
```

...

- select the elements that are greater than 50

```
m[m>50]
```

```
## [1] 73 92 57 68 61 97 60
```

- select the elements that are smaller than 70

```
m[m <70]
```

```
##  [1] 24  3  6 57  3 21 13 68 45 61 41 60 47
```

- select the elements that are smaller than 70 and greater than 30.

```
m[m < 70 & m >30]
```

```
## [1] 57 68 45 61 41 60 47
```

- select the elements that are greater than 70 or smaller than 20

```
m[m>70|m<20]
```

```
## [1] 73  3 92  6  3 13 97
```

...

- select the elements that are equal to a given value

```
m[m==99]
```

```
## integer(0)
```

- select the elements that are equal to a given value or lower than 30

```
m[m==99|m<30]
```

```
## [1] 24  3  6  3 21 13
```

```
m[c(m==99|m<30)]
```

```
## [1] 24  3  6  3 21 13
```

Indices of the elements that meet a condition

- to find out the indices of the elements that meet a condition
- we use the which() function

```
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   73    6   13   41
## [2,]   24   57   68   97
## [3,]    3    3   45   60
## [4,]   92   21   61   47
```

```
which(m==14)
```

```
## integer(0)
```

...

```r
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   73    6   13   41
## [2,]   24   57   68   97
## [3,]    3    3   45   60
## [4,]   92   21   61   47
```

```r
which(m==29)
```

```
## integer(0)
```

- the indices are returned in column-wise order

```
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   73    6   13   41
## [2,]   24   57   68   97
## [3,]    3    3   45   60
## [4,]   92   21   61   47
```

```
which(m>50)
```

```
## [1]  1  4  6 10 12 14 15
```

```
which(m == 60)
```

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

Editing elements in a matrix

- we can edit elements in matrices by assigning values to them directly

```
m <- matrix(1:16, nrow = 4, byrow = TRUE);m
```

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

- assign the value 100 to the element on row 3, column 4

```
m[3,4] <- 100
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11  100
## [4,]   13   14   15   16
```

...

- assign the value 100 to the seventh element, in column-wise order

```r
m[7] <- 100
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9  100   11  100
## [4,]   13   14   15   16
```

- assign the value 100 to the elements on row 1, columns 2, 3 and 4

```
m[1, 2:4] <- 100
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1  100  100  100
## [2,]    5    6    7    8
## [3,]    9  100   11  100
## [4,]   13   14   15   16
```

- assign the value 0 to the entire second row

```
m[2,] <- 0
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1  100  100  100
## [2,]    0    0    0    0
## [3,]    9  100   11  100
## [4,]   13   14   15   16
```

- we can also assign multiple values at once

- assign the values 31, 32 and 33 to the elements on row 1, columns 2, 3 and 4

```
m[1, 2:4] <- c(31, 32, 33); m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   31   32   33
## [2,]    0    0    0    0
## [3,]    9  100   11  100
## [4,]   13   14   15   16
```

...

- assign the values 51 to 54 to the entire third column

```r
m[,3] <- 51:54
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   31   51   33
## [2,]    0    0   52    0
## [3,]    9  100   53  100
## [4,]   13   14   54   16
```

- assign the values 1000 and 2000 to the seventh and the nineth elements in column-wise order

```
m[c(7, 9)] <- c(1000, 2000) ; m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   31 2000   33
## [2,]    0    0   52    0
## [3,]    9 1000   53  100
## [4,]   13   14   54   16
```

Adding and deleting rows and columns from a matrix

```r
m <- matrix(1:16, nrow = 4, byrow = TRUE)
m
```

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

- to add rows we use the rbind() function

```
m <- rbind(m, c(50, 60, 70, 80))
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   50   60   70   80
```

- the vector length must be equal to the number of columns in the matrix.

- Alternatively

```
x <- c(8, 10, 12, 14)
m <- rbind(m, x) ;m
```

```
##   [,1] [,2] [,3] [,4]
##      1    2    3    4
##      5    6    7    8
##      9   10   11   12
##     13   14   15   16
##     50   60   70   80
## x    8   10   12   14
```

...

- in this case the new row will have the vector name
- if we don't want that, we can remove the name

```r
rownames(m) <- NULL
```

- we can also use rbind() to bind two or more matrices
- these matrices must have the same number of columns

```
m2 <- matrix(21:28, nrow = 2, byrow = TRUE)
m2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   21   22   23   24
## [2,]   25   26   27   28
```

...

```r
m <- rbind(m, m2)
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   50   60   70   80
## [6,]    8   10   12   14
## [7,]   21   22   23   24
## [8,]   25   26   27   28
```

- to add columns in a matrix we use the cbind() function

```r
m <- matrix(1:16, nrow = 4, byrow = TRUE)
m
```

...

```r
m <- cbind(m, c(100, 101, 102, 103))
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4  100
## [2,]    5    6    7    8  101
## [3,]    9   10   11   12  102
## [4,]   13   14   15   16  103
```

...

- the vector length must be equal to the number of rows in the matrix
- we can also use cbind() to bind two or more matrices
- these matrices must have the same number of rows

...

```r
m2 <- matrix(51:58, nrow = 4, byrow = TRUE)
m2
```

```
##      [,1] [,2]
## [1,]   51   52
## [2,]   53   54
## [3,]   55   56
## [4,]   57   58
```

```r
m <- cbind(m, m2)
m
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    2    3    4  100   51   52
## [2,]    5    6    7    8  101   53   54
## [3,]    9   10   11   12  102   55   56
## [4,]   13   14   15   16  103   57   58
```

- to remove rows and column we simply use negative indices

```
m <- matrix(1:16, nrow = 4, byrow = TRUE) ; m
```

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

- remove the second row (and create a new matrix, m1)

```
m1 <- m[-2,] ; m1
```

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

Remove the first column

```
m1 <- m[,-1]
m1
```

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

- remove the first and the third row

```
m1 <- m[c(-1, -3),]
m1
```

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

...

- remove the first and the third column

```r
m1 <- m[,c(-1, -3)]
m1
```

```
##      [,1] [,2]
## [1,]    2    4
## [2,]    6    8
## [3,]   10   12
## [4,]   14   16
```

- Remove the first, the second and the third row

```r
m1 <- m[-1:-3,]
m1
```

```
## [1] 13 14 15 16
```

. . .

- remove the first, the second and the third column

```
m1 <- m[,-1:-3]
m1
```

```
## [1]  4  8 12 16
```

Minima and minima

- create a 4x5 matrix of 20 random numbers

```
i <- sample(100, 20)
m <- matrix(i, nrow = 4, byrow = TRUE)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   18   21   88   24   15
## [2,]   70   29   22   32    4
## [3,]   89   28   48   27   10
## [4,]   90   20   86   14   67
```

Get the minimum and maximum value, overall

```r
min(m)
```

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

```r
max(m)
```

```
## [1] 90
```

- get the minimum value in the third row

```r
min(m[3,])
```

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

- get the maximum value in the fourth column

```r
max(m[,4])
```

```
## [1] 32
```

Indices of the minimum and maximum values

- to get the indices of the minimum and maximum values
- we use the functions which.min() and which.max()
- the indices of the overall minimum and maximum values

```r
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   18   21   88   24   15
## [2,]   70   29   22   32    4
## [3,]   89   28   48   27   10
## [4,]   90   20   86   14   67
```

```r
which.min(m)
```

```
## [1] 18
```

```r
which.max(m)
```

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

Applying functions to matrices 1

- to perform operations on the matrix rows and columns
- we can use the apply() function
- the arguments of the apply() function are:
  - the matrix name
  - the dimension we apply the function to (1 for rows, 2 for columns)
  - the function to apply
- create a 4x4 matrix

```
m <- matrix(1:16, nrow = 4)
m
```

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

Sum of elements on each row and column

- compute the sum of the elements on each row and column, respectively.

```
apply(m, 1, sum)
```

```
## [1] 28 32 36 40
```

```
apply(m, 2, sum)
```

```
## [1] 10 26 42 58
```

. . .

- compute the product of the elements on each row and column, respectively.

```
apply(m, 1, prod)
```

```
## [1]  585 1680 3465 6144
```

```
apply(m, 2, prod)
```

```
## [1]    24  1680 11880 43680
```

...

- compute the mean for each row and column, respectively

```
apply(m, 1, mean)
```

```
## [1]  7  8  9 10
```

```
apply(m, 2, mean)
```

```
## [1]  2.5  6.5 10.5 14.5
```

...

- compute the standard deviation for each row and column, respectively.

```
apply(m, 1, sd)
```

```
## [1] 5.163978 5.163978 5.163978 5.163978
```

```
apply(m, 2, sd)
```

```
## [1] 1.290994 1.290994 1.290994 1.290994
```

Applying functions to matrices 2

- create a 4x4 matrix

```
m <- matrix(1:16, nrow = 4, byrow = TRUE) ; m
```

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

...

- compute the cumulative sums for the data values in each row.

```
apply(m, 1, cumsum)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    3   11   19   27
## [3,]    6   18   30   42
## [4,]   10   26   42   58
```

- the cumulative sums are computed by row,
- BUT the matrix is built column-wise (the default way in R)
- to built the same matrix row-wise
- we have to use the matrix() function

```
m1 <- matrix(apply(m, 1, cumsum),
             nrow = 4, byrow = TRUE)
m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    6   10
## [2,]    5   11   18   26
## [3,]    9   19   30   42
## [4,]   13   27   42   58
```

...

- compute the cumulative sums for each column

```
m
```

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

```
apply(m, 2, cumsum)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    6    8   10   12
## [3,]   15   18   21   24
## [4,]   28   32   36   40
```

...

- now everything is OK: the cumulative sums are computed by columns
- and the matrix is built column-wise
- the same happens when we use the cumprod function
- that computes the cumulative products (verify by yourself)

...

- and the same happens when we use other functions
- and apply the function by row (using the 1 argument)

...

- compute the square roots by row

```
m
```

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

```
apply(m, 1, sqrt)
```

```
##          [,1]     [,2]     [,3]     [,4]
## [1,] 1.000000 2.236068 3.000000 3.605551
## [2,] 1.414214 2.449490 3.162278 3.741657
## [3,] 1.732051 2.645751 3.316625 3.872983
## [4,] 2.000000 2.828427 3.464102 4.000000
```

- compute the natural logarithms by row

```r
apply(m, 1, log)
```

```
##           [,1]     [,2]     [,3]     [,4]
## [1,] 0.0000000 1.609438 2.197225 2.564949
## [2,] 0.6931472 1.791759 2.302585 2.639057
## [3,] 1.0986123 1.945910 2.397895 2.708050
## [4,] 1.3862944 2.079442 2.484907 2.772589
```

...

- compute the antilogarithms by row

```
apply(m, 1, exp)
```

```
##           [,1]      [,2]       [,3]      [,4]
## [1,]  2.718282  148.4132   8103.084  442413.4
## [2,]  7.389056  403.4288  22026.466 1202604.3
## [3,] 20.085537 1096.6332  59874.142 3269017.4
## [4,] 54.598150 2980.9580 162754.791 8886110.5
```

...

- to get a row-wise matrix using the sqrt function

```
m
```

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

```
m1 <- matrix(apply(m, 1, sqrt),
             nrow = 4, byrow = TRUE)
m1
```

```
##          [,1]     [,2]     [,3]     [,4]
## [1,] 1.000000 1.414214 1.732051 2.000000
## [2,] 2.236068 2.449490 2.645751 2.828427
## [3,] 3.000000 3.162278 3.316625 3.464102
```

...

- to get a row-wise matrix using the log function

```
m
```

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

```
m1 <- matrix(apply(m, 1, log), nrow = 4, byrow = TRUE)
m1
```

```
##          [,1]      [,2]     [,3]     [,4]
## [1,] 0.000000 0.6931472 1.098612 1.386294
## [2,] 1.609438 1.7917595 1.945910 2.079442
## [3,] 2.197225 2.3025851 2.397895 2.484907
## [4,] 2.564949 2.6390573 2.708050 2.772589
```

...

```r
f <- function (x) { 2*x + 3 }
m
```

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

```r
apply(m, 1, f)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    5   13   21   29
## [2,]    7   15   23   31
## [3,]    9   17   25   33
## [4,]   11   19   27   35
```

...

- to get a row-wise matrix

```
m1 <- matrix(apply(m, 1, f), nrow = 4, byrow = TRUE)
m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    5    7    9   11
## [2,]   13   15   17   19
## [3,]   21   23   25   27
## [4,]   29   31   33   35
```

Summary

- when we compute the cumulative sum, cumulative product
- square root, logarithm, exponential, sin, cos etc. by COLUMN, no problem arises
- however, when we compute the same functions by ROW
- the resulted matrix is transposed
- (because, by default, R builds the matrices column-wise)
- so to get the resulted matrix row-wise we have to use the matrix() function

Applying functions to matrices 3

- the sweep() function is useful when we have to
- perform different operations on various matrix rows and columns
- create the matrix

```
m <- matrix(1:12, nrow = 3, byrow = TRUE)
m
```

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

- For the sweep() function we must specify:
- the data source (our matrix)
- the dimension (1 for rows, 2 for columns)
- the vector of values (its length must be equal to the number of columns/rows)
- a binary operator between quotation marks: “+”, “-”, "*" or “/”

...

- add 10, 20 and 30 to each row, respectively

```r
sweep(m, 1, c(10, 20, 30), "+")
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   11   12   13   14
## [2,]   25   26   27   28
## [3,]   39   40   41   42
```

- substract 10, 20 and 30 from each row, respectively

```r
sweep(m, 1, c(10, 20, 30), "-")
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   -9   -8   -7   -6
## [2,]  -15  -14  -13  -12
## [3,]  -21  -20  -19  -18
```

...

- multiply each row by 10, 20 and 30, respectively

```r
sweep(m, 1, c(10, 20, 30), "*")
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   20   30   40
## [2,]  100  120  140  160
## [3,]  270  300  330  360
```

- divide each row by 10, 20 and 30, respectively

```r
sweep(m, 1, c(10, 20, 30), "/")
```

```
##      [,1]      [,2]      [,3] [,4]
## [1,] 0.10 0.2000000 0.3000000  0.4
## [2,] 0.25 0.3000000 0.3500000  0.4
## [3,] 0.30 0.3333333 0.3666667  0.4
```

...

- add 10, 20, 30 and 40 to each column, respectively

```
sweep(m, 2, c(10, 20, 30, 40), "+")
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   11   22   33   44
## [2,]   15   26   37   48
## [3,]   19   30   41   52
```

- substract 10, 20, 30 and 40 from each column, respectively

```
sweep(m, 2, c(10, 20, 30, 40), "-")
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   -9  -18  -27  -36
## [2,]   -5  -14  -23  -32
## [3,]   -1  -10  -19  -28
```

...

- multiply each column by 10, 20, 30 and 40, respectively

```
sweep(m, 2, c(10, 20, 30, 40), "*")
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   40   90  160
## [2,]   50  120  210  320
## [3,]   90  200  330  480
```

- divide each column by 10, 20, 30 and 40, respectively

```
sweep(m, 2, c(10, 20, 30, 40), "/")
```

```
##      [,1] [,2]      [,3] [,4]
## [1,]  0.1  0.1 0.1000000  0.1
## [2,]  0.5  0.3 0.2333333  0.2
## [3,]  0.9  0.5 0.3666667  0.3
```

Adding and multiplying matrices

- we can add or multiply two matrices of the same dimensions element-wise
- create two 3x3 matrices

```
m1 <- matrix(1:9, nrow = 3, byrow = TRUE)
m2 <- matrix(101:109, nrow = 3, byrow = TRUE)
m <- m1 + m2
m <- m1 * m2
```

...

- to perform real matrix multiplication we use the %*% operator
- the number of columns in the first matrix must be equal to the number of rows in the second matrix
- the resulted matrix will have the number of rows of the first matrix
- and the number of columns of the second matrix

...

- let's create a 3x5 matrix...

```
m1 <- matrix(1:15, nrow = 3, byrow = TRUE) ;m1
```

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

- Create a 5x4 matrix

```
m2 <- matrix(1:20, nrow = 5, byrow = TRUE) ;m2
```

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

...

- these two matrices can be multiplied
- the result will be a 3x4 matrix

```
m <- m1 %*% m2
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  175  190  205  220
## [2,]  400  440  480  520
## [3,]  625  690  755  820
```

Other matrix operations

Transpose of a matrix

- to transpose a matrix, we use the t() function

```
m <- matrix(1:20, nrow = 5, byrow = TRUE) ; m
```

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

```
t(m)
```

```
##      [,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
```

Determinant

```
m <- matrix(c(2, 4, 8, 12, 5, 7, 9, 15, 10),
            nrow = 3, byrow = TRUE) ;m
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    8
## [2,]   12    5    7
## [3,]    9   15   10
```

```
det(m)
```

```
## [1] 742
```

- to compute the inverse of a quadratic matrix
- we use the solve() function
- it only works if the determinant is different from zero

Inverse

```
mi <- solve(m)
mi
```

```
##              [,1]         [,2]        [,3]
## [1,] -0.07412399  0.107816712 -0.01617251
## [2,] -0.07681941 -0.070080863  0.11051213
## [3,]  0.18194070  0.008086253 -0.05121294
```

```
m %*% mi
```

```
##              [,1]         [,2]         [,3]
## [1,] 1.000000e+00 2.775558e-17 5.551115e-17
## [2,] 0.000000e+00 1.000000e+00 0.000000e+00
## [3,] 2.220446e-16 8.326673e-17 1.000000e+00
```

Diagonal of a matrix

- to extract the elements on the main diagonal of a quadratic matrix we use the diag() function

```
m
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    8
## [2,]   12    5    7
## [3,]    9   15   10
```

```
x <- diag(m)
x
```

```
## [1]  2  5 10
```

```
class(x)
```

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

...

- we can apply the diag() function to a vector as well
- in this case we get a quadratic matrix that contains the vector components
- in the main diagonal and zero everywhere else

```
x <- c(10, 12, 14, 16, 18)
diag(x)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   10    0    0    0    0
## [2,]    0   12    0    0    0
## [3,]    0    0   14    0    0
## [4,]    0    0    0   16    0
## [5,]    0    0    0    0   18
```

...

- we can use the diag() function to create an identity matrix
- this will create a 5*5 identity matrix

```
diag(rep(1, 5))
```

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

Arrays

Multidimensional arrays

- A multidimensional array stores several two-dimensional
- data structures (i.e matrices)
- the matrices must have the SAME dimensions
- suppose that we have three brands that sell in two supermarkets
- create a matrix that contain the brands codes and prices
- in the first supermarket

```
market1 <- matrix(c(22,44,66,9,11,5), nrow=3)
rownames(market1) <- c("brand1", "brand2", "brand3")
colnames(market1) <- c("code", "price")
market1
```

```
##        code price
## brand1   22     9
## brand2   44    11
## brand3   66     5
```

...

- create another matrix that contain the brands
- codes and prices in the second supermarket

```
market2 <- matrix(c(55,77,99,10,14,20), nrow=3)
rownames(market2) <- c("brand1", "brand2", "brand3")
colnames(market2) <- c("code", "price")
market2
```

```
##        code price
## brand1   55    10
## brand2   77    14
## brand3   99    20
```

array() function

- create an array with these matrices
- using the array() function
- in the array() function we have to specify:
  - the data sources (i.e. matrices)
  - the dimensions
- the order of the dimensions is: rows, columns, layers
- each matrix has three rows and two columns
- and the array has two layers (the two matrices)

```
markets <- array(data=c(market1, market2),
                 dim=c(3,2,2))
```

...

```
markets
```

```
## , , 1
##
##      [,1] [,2]
## [1,]   22    9
## [2,]   44   11
## [3,]   66    5
##
## , , 2
##
##      [,1] [,2]
## [1,]   55   10
## [2,]   77   14
## [3,]   99   20
```

...

- if we don't specify the dimensions, the result will be a vector, not an array

```r
markets2 <- array(data=c(market1, market2))
markets2
```

```
##  [1] 22 44 66  9 11  5 55 77 99 10 14 20
```

print the array

```
markets
```

```
## , , 1
##
##      [,1] [,2]
## [1,]   22    9
## [2,]   44   11
## [3,]   66    5
##
## , , 2
##
##      [,1] [,2]
## [1,]   55   10
## [2,]   77   14
## [3,]   99   20
```

specify the dimension names

```
markets <- array(data=c(market1, market2),
                 dim=c(3,2,2),
      dimnames = list(c("brand1", "brand2", "brand3"),
                      c("code", "price"),
                      c("smark1", "smark2")))
```

...

```r
markets
```

```
## , , smark1
##
##        code price
## brand1   22     9
## brand2   44    11
## brand3   66     5
##
## , , smark2
##
##        code price
## brand1   55    10
## brand2   77    14
## brand3   99    20
```

get the dimensions

```r
dim(markets)
```

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

```r
dimnames(markets)
```

```
## [[1]]
## [1] "brand1" "brand2" "brand3"
##
## [[2]]
## [1] "code"  "price"
##
## [[3]]
## [1] "smark1" "smark2"
```

Idexing arrays

- the indices must be put between square brackets
- we have to use three indices:
- the first index is for the rows in the matrices
- the second index is for the columns in the matrices
- the third index is for the layers # ....

```
markets
```

```
## , , smark1
##
##        code price
## brand1   22     9
## brand2   44    11
## brand3   66     5
##
## , , smark2
##
```

access the first layer (matrix)

```
markets[,,1]
```

```
##        code price
## brand1   22     9
## brand2   44    11
## brand3   66     5
```

access the second layer (matrix)

```
markets[,,2]
```

```
##        code price
## brand1   55    10
## brand2   77    14
## brand3   99    20
```

access the second column of the first matrix

```r
markets[,2,1]
```

```
## brand1 brand2 brand3
##      9     11      5
```

access the first column of the second matrix

```
markets[,1,2]
```

```
## brand1 brand2 brand3
##     55     77     99
```

access the third row, second column in the first matrix

```r
markets[3,2,1]
```

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

access the second row, second column in the second matrix

```r
markets[2,2,2]
```

```
## [1] 14
```

access the first row in the first matrix

```
markets[1,,1]
```

```
##  code price
##    22     9
```

access the third row in the second matrix

```
markets[3,,2]
```

```
##  code price
##    99    20
```

access the second row, first column in both matrices

```
markets[2,1,]
```

```
## smark1 smark2
##     44     77
```

access the third row in both matrices

```
markets[3,,]
```

```
##        smark1 smark2
## code       66     99
## price       5     20
```

access the first column in both matrices

```r
markets[,1,]
```

```
##        smark1 smark2
## brand1     22     55
## brand2     44     77
## brand3     66     99
```

Data frames

Creating data frames

- like matrices, data frames are collections of objects
- the objects in a data frame must have the SAME length
- to create a data frame we use the data.frame() function
- create two vectors of the same length (10)

```
x <- 1:10
y <- rnorm(10)
dt <- data.frame(x, y)
```

...

```r
str(dt)
```

```
## 'data.frame':    10 obs. of  2 variables:
##  $ x: int  1 2 3 4 5 6 7 8 9 10
##  $ y: num  -1.107 -1.296 0.891 -0.963 0.412 ...
```

Character data frames

- the objects in a data frame may be also of
- character or logical type

```
z <- c("a","b","c","d","e","f","g",
       "h","i","j")
w <- c(TRUE,TRUE,TRUE,TRUE,TRUE,FALSE,
       FALSE,FALSE,FALSE,FALSE)
dt <- data.frame(x,y,z,w,
        stringsAsFactors = FALSE)
head(dt)
```

```
##   x          y z     w
## 1 1 -1.1066195 a  TRUE
## 2 2 -1.2963218 b  TRUE
## 3 3  0.8911410 c  TRUE
## 4 4 -0.9633931 d  TRUE
## 5 5  0.4121375 e  TRUE
```

...

- the objects (columns) in a data frame are also called variables.
- the rows are also called entries or observations
- by default, the data frames rows don't have names
- the row.names option in the data.frame() function is set to NULL)
- however, we can name the rows if we want to.

...

```r
dt <- data.frame(x,y,z,w,
      row.names = c("row1","row2","row3","row4",
                    "row5","row6","row7"),
                 stringsAsFactors = FALSE)
head(dt)
```

```
##      x          y z     w
## row1 1 -1.1066195 a  TRUE
## row2 2 -1.2963218 b  TRUE
## row3 3  0.8911410 c  TRUE
## row4 4 -0.9633931 d  TRUE
## row5 5  0.4121375 e  TRUE
## row6 6 -0.4912129 f FALSE
```

Get data frame dimensions

```r
dim(dt)
```

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

```r
nrow(dt)
```

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

```r
ncol(dt)
```

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

```r
str(dt)
```

```
## 'data.frame':    10 obs. of  4 variables:
##  $ x: int  1 2 3 4 5 6 7 8 9 10
##  $ y: num  -1.107 -1.296 0.891 -0.963 0.412 ...
##  $ z: chr  "a" "b" "c" "d" ...
##  $ w: logi  TRUE TRUE TRUE TRUE TRUE FALSE ...
```

Lists

- A list is a data structure that can contain objects of different types
- let's create a list of four employees in a company
- Mark, Tom, Laura and Sandra
- for each employee we have the following information
- name, age, gender, annual salary, whether they are managers or not;

```
employees <- list(names = c("Mark", "Tom", "Laura",
            "Sandra"), age = c(49, 28, 35, 25),
                  gender = c("m", "m", "f", "f"),
                  salary = c(75000, 62000, 55000, 46000),
                  manager = c(TRUE, FALSE, FALSE, FALSE))
```

...

```
employees
```

```
## $names
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
##
## $age
## [1] 49 28 35 25
##
## $gender
## [1] "m" "m" "f" "f"
##
## $salary
## [1] 75000 62000 55000 46000
##
## $manager
## [1]  TRUE FALSE FALSE FALSE
```

....

```r
str(employees)
```

```
## List of 5
##  $ names  : chr [1:4] "Mark" "Tom" "Laura" "Sandra"
##  $ age    : num [1:4] 49 28 35 25
##  $ gender : chr [1:4] "m" "m" "f" "f"
##  $ salary : num [1:4] 75000 62000 55000 46000
##  $ manager: logi [1:4] TRUE FALSE FALSE FALSE
```

...

- we can create a list without object names as well
- (but it is preferable to have names)

```
employees2 <- list(c("Mark", "Tom", "Laura", "Sandra"),
                   c(49, 28, 35, 25),
                   c("m", "m", "f", "f"),
                   c(75000, 62000, 55000, 46000),
                   c(TRUE, FALSE, FALSE, FALSE))

head(employees2,2)
```

```
## [[1]]
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
##
## [[2]]
## [1] 49 28 35 25
```

...

- if there are no names,the objects are referred using indices between double brackets
- for example, [[1]]
- getting object class and type

```
class(employees)
```

```
## [1] "list"
```

```
typeof(employees)
```

```
## [1] "list"
```

Print list structure

```r
str(employees)
```

```
## List of 5
##  $ names  : chr [1:4] "Mark" "Tom" "Laura" "Sandra"
##  $ age    : num [1:4] 49 28 35 25
##  $ gender : chr [1:4] "m" "m" "f" "f"
##  $ salary : num [1:4] 75000 62000 55000 46000
##  $ manager: logi [1:4] TRUE FALSE FALSE FALSE
```

- print the objects names

```r
names(employees)
```

```
## [1] "names"   "age"     "gender"  "salary"  "manager"
```

...

- the objects in a list DO NOT have to be of the same length
- let's make a list with that contains the dishes ordered
- by three friends at a restaurant, as well as their total bill their names are Fred, Jack and Peter

...

```r
lunch <- list(Fred = c("omelette", "fried potatos",
              "chicken", "icecream"),
              Jack = c("salad", "beef steak"),
              Peter = c("salad", "lasagna", "pancakes"),
              bill = 100)
```

```r
lunch
```

```
## $Fred
## [1] "omelette"      "fried potatos" "chicken"       "icecre
##
## $Jack
## [1] "salad"      "beef steak"
##
## $Peter
## [1] "salad"    "lasagna"  "pancakes"
##
```

...

```r
str(lunch)
```

```
## List of 4
##  $ Fred : chr [1:4] "omelette" "fried potatos" "chicken" "i
##  $ Jack : chr [1:2] "salad" "beef steak"
##  $ Peter: chr [1:3] "salad" "lasagna" "pancakes"
##  $ bill : num 100
```

Vector function

- we will create the same list of employees as in the previous lecture
- we create an empty list by setting the mode parameter to list

```
employ <- vector(mode = "list")
employ
```

```
## list()
```

```
class(employ)
```

```
## [1] "list"
```

```
names(employ)
```

```
## NULL
```

```
attributes(employ)
```

```
## NULL
```

Adding objects to our list

- now we can add objects to our list

```
employ[["names"]] <- c("Mark", "Tom", "Laura", "Sandra")
employ[["age"]] <- c(49, 28, 35, 25)
employ[["gender"]] <- c("m", "m", "f", "f")
employ[["salary"]] <- c(75000, 62000, 55000, 46000)
employ[["manager"]] <- c(TRUE, FALSE, FALSE, FALSE)
head(employ,2)
```

```
## $names
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
##
## $age
## [1] 49 28 35 25
```

Indexing lists using brackets

- to access objects we use double brackets
- to access individual elements we use simple brackets

```
employees <- list(names=c("Mark", "Tom", "Laura", "Sandra"),
                  age=c(49, 28, 35, 25),
                  gender=c("m", "m", "f", "f"),
                  salary=c(75000, 62000, 55000, 46000),
                  manager=c(TRUE, FALSE, FALSE, FALSE))
```

- To extract the vector of names

```
employees[["names"]]
```

```
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
```

```
employees[["gender"]]
```

```
## [1] "m" "m" "f" "f"
```

```
x <- employees[["names"]]
x
```

```
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
```

```
class(x)
```

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

```
typeof(x)
```

To extract the vector of ages

```
employees[["age"]]
```

```
## [1] 49 28 35 25
```

- if we don't have names, we can use the object indices

```
employees[[1]]
```

```
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
```

```
employees[[5]]
```

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

Accessing individual element

- Access an individual element, we put its index betwen simple brackets
- To get Laura's name

```
employees[["names"]][3]
```

```
## [1] "Laura"
```

```
employees[["names"]][1]
```

```
## [1] "Mark"
```

```
employees[["gender"]][3]
```

```
## [1] "f"
```

```
employees[[1]][3]
```

```
## [1] "Laura"
```

Alternatively

```r
employees[[c(1,3)]]
```

```
## [1] "Laura"
```

Other examples

- get Tom's salary

```
employees[["names"]]
```

```
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
```

```
employees[["salary"]][2]
```

```
## [1] 62000
```

```
employees[[4]][2]
```

```
## [1] 62000
```

```
employees[[c(4,2)]]
```

```
## [1] 62000
```

- To get Mark's, Tom's and Laura's salaries

```
employees[["salary"]][1:3]
```

```
## [1] 75000 62000 55000
```

```
employees[["salary"]][1:2]
```

```
## [1] 75000 62000
```

...

- To get Mark's and Sandra's salaries.

```r
employees[["salary"]][c(1,4)]
```

```
## [1] 75000 46000
```

- to remove elements we use negative indices

```r
employees[["salary"]][-2]
```

```
## [1] 75000 55000 46000
```

```r
employees[["age"]][-1:-3]
```

```
## [1] 25
```

Indexing lists using the objects names.

```
employees <- list(names=c("Mark", "Tom", "Laura", "Sandra"),
                  age=c(49, 28, 35, 25),
                  gender=c("m", "m", "f", "f"),
                  salary=c(75000, 62000, 55000, 46000),
                  manager=c(TRUE, FALSE, FALSE, FALSE))
head(employees,2)
```

```
## $names
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
##
## $age
## [1] 49 28 35 25
```

. . .

- Access the vectors of names, gender, salary

```
employees$names
```

```
## [1] "Mark"   "Tom"    "Laura"  "Sandra"
```

```
employees$gender
```

```
## [1] "m" "m" "f" "f"
```

```
employees$salary
```

```
## [1] 75000 62000 55000 46000
```

...

- To get Tom's salary

```
employees$salary[2]
```

```
## [1] 62000
```

```
employees$manager[3]
```

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

- To get Tom's, Laura's and Sandra's salaries

```
employees$salary[2:4]
```

```
## [1] 62000 55000 46000
```

...

- we can also remove elements (Mark's salary, in this case)

```
employees$salary[-1]
```

```
## [1] 62000 55000 46000
```

- create a sub-list with ages and salaries only

```
emp2 <- list(age=employees$age, salary=employees$salary)
emp2
```

```
## $age
## [1] 49 28 35 25
##
## $salary
## [1] 75000 62000 55000 46000
```

Factors

- Factors are categorical variables
- They take on a limited number of distinct values called levels.
- To create a factor we use the factor() function

```
x <- c(4, 4, 6, 5, 6, 6, 6, 4, 4, 5, 4, 5, 6, 4)
x
```

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

```
f <- factor(x)
f
```

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

```
# Factor of characters
y <- c("a", "b", "c", "b", "a", "c",
       "b", "a", "a", "c")
ff <- factor(y)
ff
```

```
##  [1] a b c b a c b a a c
## Levels: a b c
```

Factor levels

- to get the factor levels

```
levels(f)
```

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

```
levels(ff)
```

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

- we can assign labels to factor values
- suppose that in the x vector the codes 4, 5 and 6
- are actually car brands:
- Ford, Toyota and Mercedes, respectively
- so let's label the factor levels accordingly

...

```r
x
```

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

```r
f <- factor(x,
    labels = c("Ford", "Toyota", "Mercedes"))
f
```

```
##  [1] Ford     Ford     Mercedes Toyota   Mercedes Mercedes
##  [8] Ford     Ford     Toyota   Ford     Toyota   Mercedes
## Levels: Ford Toyota Mercedes
```

...

- to change the codes we can use the levels option
- suppose that for some reason we have to change the codes as follows:
- 4 becomes Mercedes
- 6 becomes Toyota
- 5 becomes Ford

```
f <- factor(x,
     levels= c(5,6,4),
     labels = c("Ford", "Toyota", "Mercedes"))
x
```

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

```
f
```

```
##  [1] Mercedes Mercedes Toyota   Ford     Toyota   Toyota
##  [8] Mercedes Mercedes Ford     Mercedes Ford     Toyota
## Levels: Ford Toyota Mercedes
```

...

- if the factor levels are ordered, we will use
- the ordered() function to create it
- suppose that in the vector x below the codes 1, 2 and 3
- represent respondents' education levels:
- elementary, middle and high, respectively

```
x <- c(1,1,3,2,2,1,3,3,2,1,1,2,3)
f <- ordered(x,
levels=c(1,2,3),
labels = c("elementary", "middle", "high"))
f
```

```
##  [1] elementary elementary high       middle     middle
##  [7] high       high       middle     elementary elementary
## [13] high
## Levels: elementary < middle < high
```

...

- even if we change the coding, the order stays the same

```
f <- ordered(x,
levels=c(3,2,1),
labels = c("elementary", "middle", "high"))
f
```

```
##  [1] high        high        elementary middle      middle
##  [7] elementary elementary middle      high        high
## [13] elementary
## Levels: elementary < middle < high
```

- it is not absolutely necessary to specify the levels
- (the program will take them as we have specified in the labels option)

...

```r
f <- ordered(x,
labels = c("elementary", "middle", "high"))
f
```

```
##  [1] elementary elementary high       middle     middle
##  [7] high       high       middle     elementary elementary
## [13] high
## Levels: elementary < middle < high
```

Get a factor length

```r
length(f)
```

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

- index a factor (access the tenth value, for instance)

```r
f[10]
```

```
## [1] elementary
## Levels: elementary < middle < high
```

- add a new value to a factor

```r
f[14] <- "elementary"
f
```

```
##  [1] elementary elementary high       middle     middle
##  [7] high       high       middle     elementary elementary
## [13] high       elementary
```

Add a new level to a factor

- how to add a new level to a factor
- suppose we want to add the doctoral level
- to the factor f coded with 4
- in this case it is required to introduce
- the new level (4)

```
f <- ordered(x, levels=c(1,2,3,4),
  labels = c("elementary", "middle",
             "high", "doctoral"))
f
```

```
##  [1] elementary elementary high        middle      middle
##  [7] high        high        middle      elementary elementary
## [13] high
## Levels: elementary < middle < high < doctoral
```

- now we can add the doctoral value.

```r
f[14] <- "doctoral"
f
```

```
##  [1] elementary elementary high       middle     middle
##  [7] high       high       middle     elementary elementary
## [13] high       doctoral
## Levels: elementary < middle < high < doctoral
```

- we cannot add a new value to a factor without defining that level first.

```r
f[15] <- "unknown"
f
```

```
##  [1] elementary elementary high       middle     middle
##  [7] high       high       middle     elementary elementary
## [13] high       doctoral   <NA>
## Levels: elementary < middle < high < doctoral
```

Splitting a vector using a factor levels

- suppose we have a vector with the employees' salaries

```
sal <- c(1000, 1800, 2500, 1750, 1900, 2700, 2100, 1100)
```

- and a factor containing the same employee categories
- (worker - W, middle manager - MM, top manager - TM)

```
categ <- factor(c("W", "MM", "TM", "MM", "W",
                  "TM", "MM", "W"))
categ
```

```
## [1] W  MM TM MM W  TM MM W
## Levels: MM TM W
```

- the function split() returns the vector values by factor levels

```
split(sal, categ)
```

```
## $MM
## [1] 1800 1750 2100
##
## $TM
## [1] 2500 2700
##
## $W
## [1] 1000 1900 1100
```

- this function returns a list

```
s <- split(sal, categ)
class(s)
```

...

- the list names are the factor levels.

```
names(s)
```

```
## [1] "MM" "TM" "W"
```

- we can also split by several factors
- let's add a new factor called gender

```
gender <- factor(c("Male", "Female", "Male",
                   "Male", "Female", "Female",
                   "Male", "Female"))
```

```
str(gender)
```

```
##  Factor w/ 2 levels "Female","Male": 2 1 2 2 1 1 2 1
```

Split by category and gender

```r
s <- split(sal, list(categ, gender))
```

```r
s
```

```
## $MM.Female
## [1] 1800
##
## $TM.Female
## [1] 2700
##
## $W.Female
## [1] 1900 1100
##
## $MM.Male
## [1] 1750 2100
##
## $TM.Male
```

- the result is a list again

```
class(s)
```

```
## [1] "list"
```

The tapply() function

- the tapply() function applies an operation to a vector values broken down by factor levels.

```
sal <- c(1000, 1800, 2500, 1750, 1900, 2700, 2100, 1100)
categ <- factor(c("W", "MM", "TM", "MM", "W", "TM",
                  "MM", "W"))
```

- compute the mean salary by category

```
tapply(sal, categ, mean)
```

```
##       MM       TM        W
## 1883.333 2600.000 1333.333
```

...

- tapply() returns an array

```
t <- tapply(sal, categ, mean)

class(t)
```

```
## [1] "array"
```

- let's add a new factor: gender

```
gender <- factor(c("Male", "Female",
                   "Male", "Male", "Female",
                   "Female", "Male", "Female"))
```

Mean by both category and gender.

- compute the mean by both category and gender

```r
t <- tapply(sal, list(categ, gender), mean)
t
```

```
##    Female Male
## MM   1800 1925
## TM   2700 2500
## W    1500 1000
```

- this time tapply() returned a matrix

```r
class(t)
```

```
## [1] "matrix"
```

The by() function

- by() does a similar thing as tapply()
- it applies an operation to a vector values
- broken down by factor levels

```r
sal <- c(1000, 1800, 2500, 1750, 1900,
         2700, 2100, 1100)
categ <- factor(c("W", "MM", "TM", "MM",
                  "W", "TM", "MM", "W"))
```

...

- Compute the mean salary by category

```
by(sal, categ, mean)
```

```
## categ: MM
## [1] 1883.333
## ------------------------------------------------------------------------
## categ: TM
## [1] 2600
## ------------------------------------------------------------------------
## categ: W
## [1] 1333.333
```

```
b <- by(sal, categ, mean)
```

...

- the object b is of a special class called “by”

```
class(b)
```

```
## [1] "by"
```

```
typeof(b)
```

```
## [1] "double"
```

- we can index b as a vector

```
b[2]
```

```
##   TM
## 2600
```

...

- we can convert b into a list as well

```
b <- as.list(b)
class(b)
```

```
## [1] "list"
```

```
b
```

```
## $MM
## [1] 1883.333
##
## $TM
## [1] 2600
##
## $W
## [1] 1333.333
```

...

- now we can index the object b as a list

```
b$TM
```

```
## [1] 2600
```

Programming structures

For loops

- Loops are programming structures that help us repeat (replicate) commands or instructions
- The for loop allows us to iterate over a vector or a sequence
- Syntax:
  for (values in sequence) { block of instructions }

- So the for loop will repeat the block of instructions for each value in the sequence
- Here's a simple loop that squares the numbers from 1 to 10 and prints the results

```
for (i in 1:10) {
  print(i ^ 2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
```

...

- We can choose to store the squares in a vector
- We must pre-define (initialize) the vector outside the loop

```r
x <- c()
for (i in 1:10) {
  x <- c(x, i^2)
}
x
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Another way to do the same thing

```
x <- c()
for (i in 1:10) {
  x[i] <- i ^ 2
}
x
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Create a for loop that squares the components of a vector

```
x <- seq(1,10,length=20)
x
```

```
##  [1]  1.000000  1.473684  1.947368  2.421053  2.894737  3.3
##  [8]  4.315789  4.789474  5.263158  5.736842  6.210526  6.6
## [15]  7.631579  8.105263  8.578947  9.052632  9.526316 10.0
```

```
for (i in x) {
  print(i^2)
}
```

```
## [1] 1
## [1] 2.171745
## [1] 3.792244
## [1] 5.861496
## [1] 8.379501
## [1] 11.34626
```

Or

```
x <- seq(1,10,length=20)
x
```

```
##  [1]  1.000000  1.473684  1.947368  2.421053  2.894737  3.3
##  [8]  4.315789  4.789474  5.263158  5.736842  6.210526  6.6
## [15]  7.631579  8.105263  8.578947  9.052632  9.526316 10.0
```

```
for (i in seq_along(x)) {
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
```

Put the results in a new vector y

```
y <- c()
for (i in x) {
  y <- c(y, i^2)
}
y
```

```
##  [1]   1.000000   2.171745   3.792244   5.861496   8.37950
##  [7]  14.761773  18.626039  22.939058  27.700831  32.91135
## [13]  44.678670  51.235457  58.240997  65.695291  73.59833
## [19]  90.750693 100.000000
```

Using brackets

- If you want to use brackets to create the y vector components it's a bit more complicated.

```
y <- c()
ind <- 1
for (i in x) {
    y[ind] <- i^2
    ind <- ind + 1
}
y
```

```
##  [1]   1.000000   2.171745   3.792244   5.861496   8.37950
##  [7]  14.761773  18.626039  22.939058  27.700831  32.91135
## [13]  44.678670  51.235457  58.240997  65.695291  73.59833
## [19]  90.750693 100.000000
```

important functions with for loops

```
# ?seq
# ? seq_along
# ? seq_len
```

Next statement

- The next statement skips the current iteration of the loop if a condition is met.
- The following loop will square the numbers from 1 to 10 except 4

```r
x <- c()
for (i in 1:10) {
    if (i == 4) next
    x <- c(x, i^2)
}
x
```

```
## [1]   1   4   9  25  36  49  64  81 100
```

Break statement

- The break statement ends the loop if a condition is met.
- The following loop will square the numbers from 1 to 3

```
x <- c()
for (i in 1:10) {
    if (i == 4) break
    x <- c(x, i^2)
}
x
```

```
## [1] 1 4 9
```

While loops

- The while loop executes a block of commands while a condition is satisfied.
- (when the condition is not satisfied any longer, it stops)
- Syntax:

  while (condition) { block of instructions }

Example.

- Create a while loop that takes the square root from the numbers 1-10 and stores the results in a vector

```r
i <- 0
x <- c()
while (i<10) {
    i <- i + 1
    x <- c(x, sqrt(i))
}
x
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490
##  [8] 2.828427 3.000000 3.162278
```

- N.B. if the condition is always true the loop will go on infinitely (you'll have to stop it by force)

...

```
# i <- 0
# while (i<10) {
#   sqrt(i)
# }
```

- the next statement skips the current iteration of the loop if a condition is met

- the following loop will take the square root of numbers from 1 to 10 except 4

```
i <- 0
x <- c()
while (i<10) {
  i <- i +1
  if (i==4) next
  x <- c(x, sqrt(i))
}
x
```

```
## [1] 1.000000 1.414214 1.732051 2.236068 2.449490 2.645751 2
## [9] 3.162278
```

- The break statement ends the loop if a condition is met.

- The following loop will take the square root of numbers from 1 to 3

```
i <- 0
x <- c()
while (i<10) {
  i <- i +1
  if (i==4) break
  x <- c(x, sqrt(i))
}
x
```

```
## [1] 1.000000 1.414214 1.732051
```

Repeat loops

- Repeat loop replicates a block of instructions for an indefinite number of times.
- Syntax:

repeat { block of instructions }

- this loop has no condition. So we have to use a break statement to make it stop at a given point otherwise, it will go on infinitely.

...

- Create a repeat loop that prints the even numbers from 2 to 20
- it stops when our variable is equal to 10

```
i <- 0
repeat {
  i <- i + 1
  print(i * 2)
  if ( i==10 ) break
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
```

Without break statement

- Let's see what happens without the break statement

```
# i <- 0
# repeat {
#   i <- i + 1
#   print(i * 2)
# }
```

Nested for loops

Example

```
x <- c(1, 2, 3)
y <- c(10, 20, 30, 40, 50)
```

- the loop below takes each component in x and multiplies it with each component in y then stores the results in a 3x5 matrix row-wise
- first we must pre-define the matrix as an empty object

...

```
m <- c()
### run the loop
for (i in 1:length(x)) {
  # initialize a vector that will be the matrix row
    rw <- c()
    for (j in 1:length(y))  {
    # inside the smaller loop i remains constant
    # so the x[i] component is multiplied with each element of
    # and the result is stored in the rw vector
        rw <- c(rw, x[i] * y[j])
    }
    # add the new row to the object m
    m <- rbind(m, rw)
}
m
```

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

Looping through a matrix

- the next code is a loop through a matrix
- it computes the sum of component squares for each row and stores the result in a vector

```
m <- matrix(1:12, nrow = 3, byrow = TRUE)
m
```

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

Initialize the vector of the sum of squares

```
vect_ssq <- c()
for (i in 1:nrow(m)) {
    ### here we loop in rows
    ### initialize the vector of squares
    sq <- c()
    for (j in 1:ncol(m))  {
        ### here we loop in columns
        ### square each element in the row
        ### and add the results to the vector of squares
        sq <- c(sq, m[i,j]^2)
    }
    ### add the sum of squares - sum(sq) - to the vector
    vect_ssq <- c(vect_ssq, sum(sq))
}
vect_ssq
```

```
## [1]  30 174 446
```

Conditional statements

- A conditional statement (if statements) executes a set of instructions only if a given condition is met
- Syntax:

  if (condition) {instructions to be executed if the condition is met} else {instructions to be executed if the condition is not met}
- the else statement is not mandatory if the else statement is missing and the condition is not satisfied the program will not execute anything

Example

- multiply a number by 5 if it is greater than zero

```
x <- 10
if (x>0) { x*5 }
```

```
## [1] 50
```

```
x <- -2
if (x>0) { x*5 }
```

Example

- multiply a number by 5 if it is greater than zero else, multiply it by 10

```
x <- 10

if (x>0) { x*5 } else { x*10 }
```

```
## [1] 50
```

```
x <- -7

if (x>0) { x*5 } else { x*10 }
```

```
## [1] -70
```

boolean expression

- the condition can be more complex (using boolean expression)
- check if two numbers are both strictly positive
- if yes, compute their sum
- if no, print the message "Stop code"

```
x <- 10
y <- 7
if (x>0 & y>0) { x + y } else { print("Stop code") }
```

```
## [1] 17
```

```
x <- 10
y <- 0
if (x>0 & y>0) { x + y } else { print("Stop code") }
```

```
## [1] "Stop code"
```

Example

- Check if at least one of two numbers is strictly positive if yes, add them if no, print the message “Stop code”

```
x <- 10
y <- -5
if (x>0 | y>0) { x + y } else { print("Stop code") }
```

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

```
x <- 0
y <- -2
if (x>0 | y>0) { x + y } else { print("Stop code") }
```

```
## [1] "Stop code"
```

. . .

- the blocks of instructions can be more complex
- if a number is positive or zero create a sequence of 10 components from 0 to that number sum the components and square the sum
- else change the sign of the number and do the same operations as above.

...

```r
x <- -10
if (x>=0)  {
    s <- seq(0, x, length = 10)
    sum(s)^2
} else {
    x <- -x
    s <- seq(0, x, length = 10)
    sum(s)^2
}
```

```
## [1] 2500
```

Nested conditional statements

- Check whether a number is lower than 100
- if yes, check whether it is lower than 50 then prints appropriate messages for each situation

```
x <- 900
if ( x<=100 ) {
    if ( x <= 50) {
        print("Your number is lower than or equal to 50")
    } else {
        print("Your number is between 50 and 100")
    }
} else {
    print("Your number is greater than 100")
}
```

```
## [1] "Your number is greater than 100"
```

Loops and conditional statements

- ifelse() is a function that combines a loop and a conditional statement

```
x <- c(8, 10, 15, 20, 23, 26, 31)
ifelse(x%%2==0, x/2, x)
```

```
## [1]  4  5 15 10 23 13 31
```

. . .

- we can do the same using a for loop

```
for (i in x) {

    if (i%%2==0) { print(i/2) } else {

        print(i)

    }

}
```

```
## [1] 4
## [1] 5
## [1] 15
## [1] 10
## [1] 23
```

. . . .

- A loop that goes through a matrix and separates the even components from the odd ones putting them in different vectors.

...

```r
m <- matrix(sample(100, 9), nrow = 3)
even <- c()
odd <- c()
### the i index will be used for the rows
### the j index will be used for the columns
for (i in 1:nrow(m))  {
    for (j in 1:ncol(m))   {
        if( m[i,j]%%2 == 0 )  { even <- c(even, m[i,j]) } else
            odd <- c(odd, m[i,j])
        }
    }
}
even
```

```
## [1] 14  6
```

```r
odd
```

User defined functions

- A function is a sequence of instructions that the programmer will likely use frequently
- that's why it is convenient to store these instructions in an object that can be easily called later on
- this object is a function
- A function can be viewed also as a sub-program or sub-routine.
- the R program has very many built-in functions
- the apply() family of functions, for example or the mathematical functions: sqrt(), exp(), log(), sin(), abs() etc. and many more
- In this section we talk about functions written by users
- Syntax: function (arguments) { block of instructions }

...

- A function that computes the following: x^2 + 3*x + 5

```
f <- function (x)  {
  x^2 + 3*x + 5
}
```

...

- call the function (apply the function to particular arguments)

```r
f(1)
```

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

```r
f(-5)
```

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

```r
f(1:10)
```

```
##  [1]   9  15  23  33  45  59  75  93 113 135
```

Get the class

```r
class(f)
```

```
## [1] "function"
```

- A function of two arguments, x and y, that computes the following: sin(x) + cos(y)

```
f <- function (x,y) {
  sin(x) + cos(y)
}

f(0,0)
```

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

```
f(190, 120)
```

```
## [1] 1.81198
```

...

- A function that computes x^2/(y-1) if y is different from 1

```
f <- function (x,y) {
  if (y!=1) { x^2/(y-1) }
}

f(10, 11)
```

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

```
f(3,1)
```

...

- The same function, a bit more developed

```
f <- function (x,y) {
  if (y!=1) { x^2/(y-1) } else {

    print("The y value must be different from 1.")

  }

}
```

Area of a rectangle

- A function that computes the area of a rectangle

```
area <- function (width, height)  {
  width * height
}
area(10,4)
```

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

- We can assign default values to arguments

```
area2 <- function (width, height=4)  {
  width * height
}
area2(10)
```

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

Function arguments

- to get the function arguments we use formals()

```r
formals(area)
```

```
## $width
##
##
## $height
```

- to get the block of statements we use body()

```r
body(area)
```

```
## {
##     width * height
## }
```

The return command

- By default, a function returns the last computed value
- however, sometimes we have to use the return command to get a returned value.

Syntax:

return (expression)

Example

```r
area <- function (width, height)  {
  a <- width * height
}
area(5,3)
# a
```

- the variable a is local to the function it cannot be found in the global environment
- to get a return, we have to use the return command

...

- Let's rewrite the function

```
area <- function (width, height)  {
    a <- width * height
    return(a)
}
area(5,3)
```

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

...

- We can make the function return other values too for example, the width argument.

```
area <- function (width, height)  {
    a <- width * height
    return(list(a, width))
}
area(5,3)
```

```
## [[1]]
## [1] 15
##
## [[2]]
## [1] 5
```

- So if we want to return more than one values we have to put them in a list to make the a variable global, we must use a special assignment symbol.
- <<-

```
# area <- function (width, height)  {
#   a <<- width * height    # special operator
#   return(list(a, width))
# }
# area(5,3)
# a
```

More complex function examples

- create a function that loops in two vectors, multiplies each component of the first vector with each component of the second vector and creates a matrix of the products.

```
f <- function (x,y) {
    m <- c()
    for (i in 1:length(x)) {
    rw <- c()
    for (j in 1:length(y))  {
    rw <- c(rw, x[i] * y[j])
}
m <- rbind(m, rw)
}
return(m)
}
```

Apply the function

```r
f(x = 1:5, y = 2:6)
```

```
##     [,1] [,2] [,3] [,4] [,5]
## rw     2    3    4    5    6
## rw     4    6    8   10   12
## rw     6    9   12   15   18
## rw     8   12   16   20   24
## rw    10   15   20   25   30
```

...

```r
f(c(1,2,3), c(10, 20, 30))
```

```
##      [,1] [,2] [,3]
## rw     10   20   30
## rw     20   40   60
## rw     30   60   90
```

...

- create a function that loops through a matrix,
- computes the sum of component squares for each row and stores the result in a vector

...

```
f <- function (m) {
     vect_ssq <- c()
   for (i in 1:nrow(m)) {
   sq <- c()
   for (j in 1:ncol(m))  {
   sq <- c(sq, m[i,j]^2)
}
vect_ssq <- c(vect_ssq, sum(sq))
}
  return(vect_ssq)
}
```

...

```r
mat <- matrix(1:9, nrow = 3, byrow = TRUE)
f(mat)
```

```
## [1]  14  77 194
```

Practical example

Checking whether a positive integer is a perfect square

- A number is a perfect square if its square root is an integer first, our function will check whether the number is positive
- if yes, it will check whether it is an integer finally, it will check whether it is a perfect square.

...

```
isperf <- function (x)  {
   if ( x<0 )  {
     print("The argument is a negative number!")
   } else {
       if ( round(x)!= x )   {
         print("The argument is not an integer!")
       }  else  {
         if (  round(sqrt(x)) == sqrt(x) )  {
           print("The argument is a perfect square.")
           return(sqrt(x))
         }   else  {
           print("The argument is not a perfect square.")
         }
       }
   }
}
```

Use the function

```r
isperf(64)
```

```
## [1] "The argument is a perfect square."
```

```
## [1] 8
```

```r
isperf(-5)
```

```
## [1] "The argument is a negative number!"
```

```r
isperf(3.5)
```

```
## [1] "The argument is not an integer!"
```

```r
isperf(42)
```

```
## [1] "The argument is not a perfect square."
```

Solving a quadratic equation

```
qd <- function (a, b, c)  {
      delta <- b^2 - 4*a*c
      if (delta<0) {
        print("The equation does not have real solutions.")
        return(delta)
      }  else  {
        if ( delta == 0)  {
         x1 <- (-b)/(2*a)
          print("The equation has one real solution.")
          return(list(delta, x1))
        }  else {
    x1 <- (-b+sqrt(delta))/(2*a)
    x2 <- (-b-sqrt(delta))/(2*a)
          print("The equation has two real solutions.")
          return(list(delta, x1, x2))}}}
```

Solve the equation: $2x\hat{}2 + 10x + 8 = 0$

```r
qd(2, 10, 8)
```

```
## [1] "The equation has two real solutions."

## [[1]]
## [1] 36
##
## [[2]]
## [1] -1
##
## [[3]]
## [1] -4
```

Solve the equation: -x^2 - 4*x - 4 = 0

```r
qd(-1, -4, -4)
```

```
## [1] "The equation has one real solution."
```

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

solve the equation: x^2 + x + 1 = 0

```r
qd(1, 1, 1)
```

```
## [1] "The equation does not have real solutions."
## [1] -3
```

Creating binarx operations using fucntions

- A binary operation is an operation that involves two terms
- We have already learned binary operations like %in% or %*%
- The users can create their own binary operations using functions
- These operations work well on either scalars, vectors or matrices
- To create a binary operation we must observe these rules:
    - 1. the function must have two arguments
    - 2. the function name must begin and end with a %
    - 3. the function name must be put between double quotes

...

- Create a binary operation that multiplies the squares of two numbers.

```
"%a2b2%" <- function (a,b)  { a^2*b^2}

2 %a2b2% 3
```

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

Use the same operation with two vectors

```
c(1,2) %a2b2% c(3,4)
```

```
## [1]  9 64
```

Use the same operation with two matrices

```r
m1 <- matrix(1:4, nrow = 2, byrow = TRUE)
m2 <- matrix(7:10, nrow = 2, byrow = TRUE)
m1 %a2b2% m2
```

```
##      [,1] [,2]
## [1,]   49  256
## [2,]  729 1600
```

- A binary operation that computes the logarithm of the sum of the inverses of two numbers (it will work only if the numbers are strictly positive)

```
"%logab%" <- function (a,b) { log(1/a+1/b) }
0.5 %logab% 0.1
```

```
## [1] 2.484907
```

```
0.5 %logab% 0
```

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

```
0.5 %logab% -0.5
```

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

The apply family of functions

- Writing for, while loops is useful when programming but not particularly easy when working interactively on the command line.
- There are some functions which implement looping to make life easier.
  - lapply: Loop over a list and evaluate a function on each element
  - sapply: Same as lapply but tries to simplify the result
  - mapply: Multivariate version of lapply
- An auxiliary function split is also useful, particularly in conjunction with lapply.

lapply

- lapply takes three arguments:
  - a list X,
  - a function (or the name of a function) FUN,
  - and other arguments via its ... argument.
- If X is not a list, it will be coerced to a list using as.list.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x00000000045be348>
## <environment: namespace:base>
```

- The actual looping is done internally in C code which makes it fast.

....

- Lapply always returns a list, regardless of the class of the input
- Example

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] -0.04607259
```

Example

```
x <- list(a = 1:4,
          b = rnorm(10),
          c = rnorm(20, 1),
          d = rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] 0.1883634
##
## $c
## [1] 0.9840625
##
## $d
```

Example

```r
x <- 1:4
lapply(x, runif)
```

```
## [[1]]
## [1] 0.4427155
##
## [[2]]
## [1] 0.7593092 0.8458438
##
## [[3]]
## [1] 0.1455904 0.2442342 0.9802017
##
## [[4]]
## [1] 0.7655873 0.3601417 0.4875675 0.4265774
```

Example

```
x <- 1:4
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 9.599585
##
## [[2]]
## [1] 6.741603 6.542366
##
## [[3]]
## [1] 6.538526 8.202612 1.168570
##
## [[4]]
## [1] 0.9117094 2.6788117 7.4949876 4.9368820
```

Anonymous functions

- lapply and friends make heavey use of anonymous fucntion
- Create a list of two matrices.

```
x <- list(a = matrix(data = 1:4,
                     nrow = 2, ncol = 2,
                     byrow = FALSE),
          b = matrix(data = 1:6, nrow = 3,
                     ncol = 2, byrow = TRUE))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $b
##      [,1] [,2]
```

Example

- An anonymous fucntion for extracting the first column of each matrix.

```
lapply(x, function(y) y[ ,1])
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 3 5
```

sapply

- sapply will try to simplify the result of lapply if possible.
- if the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length(>1), a matrix is returned.
- if it cant figure out things a list is returned.

...

```r
x <- list(a = 1:4, b = rnorm(10),
          c = rnorm(20, 1),
          d = rnorm(100, 5))
sapply(x, mean)
```

```
##          a          b          c          d
##  2.5000000 -0.5736388  1.0699685  4.9496856
```

```r
# mean(x)
```

split

- split takes a vector or other objects and splits it into groups determined by a vector or a list of factors.

```
str(split)
```

```
## function (x, f, drop = FALSE, ...)
```

- x is usually a data frame
- f is a factor(or coerced t one) or a list of factors drop indicates if empty factor levels should be dropped
- ... further potential arguments passed to methods

Splitting a data frame

```r
head(airquality, n=3)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
```

```r
s <- split(airquality[, c("Ozone","Solar.R", "Wind")],
           f = airquality$Month)
sapply(s, colMeans)
```

```
##                5         6          7        8        9
## Ozone         NA        NA         NA       NA       NA
## Solar.R       NA 190.16667 216.483871       NA 167.4333
## Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

```r
sapply(s, colMeans, na.rm = TRUE)
```

mapply

- mapply is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
str(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.N
```

mapply

```r
## the hard way
lHard <- list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
## the smart way
lSmart <- mapply(rep, 1:4, 4:1)
lSmart
```

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

Remark

- apply functions are faster than loops

String manipulation in R

Creating strings

- we can create a string variable using
- either double quotes or single quotes

```
x <- "Hello my friends"
x
```

```
## [1] "Hello my friends"
```

```
y <- 'Hello my friends'
y
```

```
## [1] "Hello my friends"
```

combine double quotes with single quotes

- we can combine double quotes with single quotes

```r
x <- "Hello 'my' friends"
x
```

```
## [1] "Hello 'my' friends"
```

```r
# y <- 'Hello 'my' friends'
# y
```

- but we cannot use double quotes or single quotes in the same statement more than one time.

Number of characters in the string

- count the number of characters in the string

```r
nchar(x)
```

```
## [1] 18
```

```r
### get class and type
class(x)
```

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

```r
typeof(x)
```

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

built-in vector letters

create sequences of letters using the built-in vector letters

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
#OR


LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

the first letter of the alphabet

```r
letters[1]
```

```
## [1] "a"
```

```r
### the first five letters of the alphabet

letters[1:5]
```

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

...

- the first, fifth and fourteenth letters of the alphabet

```r
letters[c(1, 5, 14)]
```

```
## [1] "a" "e" "n"
```

- create empty strings and empty character vectors
- create an empty string

```r
x <- ""
x
```

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

```r
nchar(x)
```

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

create a vector of two empty strings

```r
y <- character(2)

y
```

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

```r
length(y)
```

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

```r
nchar(y)
```

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

create an empty character vector

```r
z <- character(0)

z
```

```
## character(0)
```

```r
length(z)
```

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

```r
nchar(z)
```

```
## integer(0)
```

add a component to the vector

```r
z[1] <- "Tom"

length(z)
```

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

```r
nchar(z)
```

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

check whether a variable is of character type

```
x <- "The wheather is fine"
is.character(x)
```

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

convert a numeric vector in a character vector

```
x <- c(2, 3, 4)
typeof(x)
```

```
## [1] "double"
```

```
x <- as.character(x)
x
```

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

```
typeof(x)
```

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

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

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

Printing strings

- the most common way to print strings is to use the print() function

```
print("The weather is fine")
```

```
## [1] "The weather is fine"
```

- to remove the quotes

```
print("The weather is fine", quote = FALSE)
```

```
## [1] The weather is fine
```

Print without quotes

- to print without quotes we can also use
- the noquote() function

```r
noquote("The weather is fine")
```

```
## [1] The weather is fine
```

The format() function

- the format() function is used to print the
- strings of numbers in the desired format
- print the string retaining 3 digits only

```r
format(3.823564997, digits = 3)
```

```
## [1] "3.82"
```

- the nsmall option indicates the minimum number
- of decimal places

```r
format(5.8, nsmall = 4)
```

```
## [1] "5.8000"
```

...

- format() also converts numbers in strings

```r
x <- 5.45839
typeof(x)
```

```
## [1] "double"
```

```r
y <- format(x, digits = 3)
y
```

```
## [1] "5.46"
```

```r
typeof(y)
```

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

The sprintf() function

- the sprintf() function offers more advanced
- formatting options.
- syntax: sprintf(format, string)
- all the formats start with a % sign
- and they are put between double quotes
- %f is used for decimal numbers
- by default, it prints six decimals

...

```r
sprintf("%f", 0.725896956)
```

```
## [1] "0.725897"
```

```r
### to print 3 decimals only
sprintf("%.3f", 0.725896956)
```

```
## [1] "0.726"
```

```r
### round the number (print no decimals)
sprintf("%.f", 0.725896956)
```

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

print the + sign (for positive numbers)

```r
sprintf("%+f", 0.725896956)
```

```
## [1] "+0.725897"
```

```r
### print the + sign (but the first 3 decimals only)
sprintf("%+.3f", 0.725896956)
```

```
## [1] "+0.726"
```

...

- %e and %E print the number in exponential format

```r
sprintf("%e", 82.235691)
```

```
## [1] "8.223569e+01"
```

```r
sprintf("%E", 82.235691)
```

```
## [1] "8.223569E+01"
```

- %g prints six digits by default

```r
sprintf("%g", 82.235691)
```

```
## [1] "82.2357"
```

print the first 4 digits

```r
sprintf("%.4g", 82.235691)
```

```
## [1] "82.24"
```

- %s prints the desired number of characters in a string

```r
sprintf("%.4s", "Philadelphia")
```

```
## [1] "Phil"
```

%d is used to print integers

```r
sprintf("%d", 23755)
```

```
## [1] "23755"
```

advanced uses of the sprintf() function

- we want to print "This book costs 12.8 dollars"
- (the book price is 12.82)

```
sprintf("This book costs %.1f dollars", 12.82)
```

```
## [1] "This book costs 12.8 dollars"
```

- we want to print the following:
- "The sum of the numbers 7 and 3 is 10"

```
a <- 7
b <- 3
x <- sprintf("The sum of the numbers %d and %d is %d",
             a, b, a+b)
x
```

```
## [1] "The sum of the numbers 7 and 3 is 10"
```

- we want to print the following:
- "The sum of the numbers 4.5 and 10 is 14.5"

```r
a <- 4.5
b <- 10
x <- sprintf("The sum of the numbers %.1f and %d is %.1f",
             a, b, a+b)
x
```

```
## [1] "The sum of the numbers 4.5 and 10 is 14.5"
```

Concatenating strings

- we can concatenate string variables using the c() function
- however, the result may not be as expected

```
x <- "The weather"
y <- "is fine"
z <- c(x, y)
z
```

```
## [1] "The weather" "is fine"
```

...

- A more useful concatenating function for
- strings is paste()

```
z <- paste(x, y)
z
```

```
## [1] "The weather is fine"
```

- the default separator is the space
- we can indicate the separator using the sep option

```
z <- paste(x, y, sep = " ")
z
```

```
## [1] "The weather is fine"
```

to use no separator

```r
z <- paste(x, y, sep = "")
z
```

```
## [1] "The weatheris fine"
```

- to use the dash as a separator

```r
z <- paste(x, y, sep = "-")
z
```

```
## [1] "The weather-is fine"
```

...

- paste() can be used to concatenate character
- vectors as well it will concatenate them element-wise

```
x <- c("a", "b", "c", "d")
y <- c(1, 2, 3, 4)
w <- paste(x, y)
w
```

```
## [1] "a 1" "b 2" "c 3" "d 4"
```

- use a double dash as a separator (instead of the space)

```r
w <- paste(x, y, sep = "--")
w
```

```
## [1] "a--1" "b--2" "c--3" "d--4"
```

- to put a comma between the pairs of elements we use the collapse option

```r
w <- paste(x, y, sep = "--", collapse = ",")
w
```

```
## [1] "a--1,b--2,c--3,d--4"
```

put a comma and a space between the pairs

```
w <- paste(x, y, sep = "--", collapse = ", ")
w
```

```
## [1] "a--1, b--2, c--3, d--4"
```

- Another example of using collapse

```
x <- c("The weather", "we go to")
y <- c("is fine", "take a walk")
z <- paste(x, y, collapse = " and ")
z
```

```
## [1] "The weather is fine and we go to take a walk"
```

...

- in conclusion, we use sep to indicate
- the separator between the elements in a pair
- and we use collapse to indicate the separator between pairs
- paste0() is a version of paste
- that uses no separator by default

```
paste0("Port", "land")
```

```
## [1] "Portland"
```

...

- this is the same as writing

```r
paste("Port", "land", sep="")
```

```
## [1] "Portland"
```

- Another concatenating (and formatting)
- function is cat()

```r
cat("The weather is fine")
```

```
## The weather is fine
```

...

- the cat() function does not return a vector
- the line indicator ([1]) is missing
- the default separator of cat() is the space

```
cat("The weather", "is fine")
```

```
## The weather is fine
```

- we can modify the separator with the sep option

```
cat("The weather", "is fine", sep = "_")
```

```
## The weather_is fine
```

String manipulation (1)

- to change the case of a string, we can use the functions
- tolower(), toupper() and casefold()

```
x <- "Mark and Jenny went to New York"
### tolower() converts everything to lower case
tolower(x)
```

```
## [1] "mark and jenny went to new york"
```

```
# toupper() converts everything to upper case
toupper(x)
```

```
## [1] "MARK AND JENNY WENT TO NEW YORK"
```

casefold()

- by default, casefold() converts everything to lower case

```r
casefold(x)
```

```
## [1] "mark and jenny went to new york"
```

```r
### we can change this by setting the upper option to TRUE
casefold(x, upper = TRUE)
```

```
## [1] "MARK AND JENNY WENT TO NEW YORK"
```

chartr() function

- the chartr() function helps us change characters in a string
- suppose we wanted to write “Mary has a cat”
- but we erroneously wrote:

```
x <- "Mary has o cat"

##to change the o into a
chartr("o", "a", x)
```

```
## [1] "Mary has a cat"
```

...

- Another example to see how chartr() behaves

```
x <- "Mary has o dog"
chartr("o", "a", x)
```

```
## [1] "Mary has a dag"
```

- so chartr() changes ALL the specified characters
- in the following string a was replaced with *
- and r was replaced with $

```
x <- "B*rry h*s * $ed t$uck"
```

- to change each of them into the correct character

```
chartr("*$", "ar", x)
```

```
## [1] "Barry has a red truck"
```

String maniulation (2)

- how to extract a substring from a string

```
x <- "Philadelphia"
```

- we will use the substr() function
- we must specify: the string,
- the position of the first character
- and the position of the last character

- extract five characters from x
- form the fifth to the nineth

```
substr(x, 5, 9)
```

```
## [1] "adelp"
```

- try to extract characters in reverse order
- (for example, form the tenth to the fifth)

```
substr(x, 10, 5)
```

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

- substr() is a vectorized function

```
x <- c("Philadelphia", "Chicago", "Seattle")
### extract three characters from each
### component (2, 3 and 4)
substr(x, 2, 4)
```

```
## [1] "hil" "hic" "eat"
```

```
### in the vector above, replace the second character
### in each component with a $ sign
substr(x, 2, 2) <- "$"
x
```

```
## [1] "P$iladelphia" "C$icago"      "S$attle"
```

```r
x <- c("Philadelphia", "Chicago", "Seattle")
# Replace the characters 2, 3 and 4 with "$$$"
# (in each component)
substr(x, 2, 4) <- "$$$"
x
```

```
## [1] "P$$$adelphia" "C$$$ago"      "S$$$tle"
```

...

```
x <- c("Philadelphia", "Chicago", "Seattle")
```

- replace the second character in each component:
- with a $ in the first component
- with a * in the second component
- with a & in the third component

```
substr(x, 2, 2) <- c("$", "*", "&")
x
```

```
## [1] "P$iladelphia" "C*icago"      "S&attle"
```

```
x <- c("Philadelphia", "Chicago", "Seattle")
```

...

- replace the characters 2, 3 and 4 in each component
- with $*& in the first component
- with *&$ in the second component
- with &$* in the third component

```
substr(x, 2, 4) <- c("$*&", "*&$", "&$*")
x
```

```
## [1] "P$*&adelphia" "C*&$ago"       "S&$*tle"
```

String manipulation (3)

- how to split a string based on a substring
- we use the strsplit() function
- we must specify the string and the substring

```
x <- "1589-3558-0156-2079"
```

- let's split the string above by the dashes

```
strsplit(x, split="-")
```

```
## [[1]]
## [1] "1589" "3558" "0156" "2079"
```

Other splitting examples

```r
strsplit("Philadelphia", split="d")
```

```
## [[1]]
## [1] "Phila"  "elphia"
```

- the splititng substring is not considered

```r
strsplit("New York", split=" ")
```

```
## [[1]]
## [1] "New"  "York"
```

split by letters

```
strsplit("Detroit", split="")
```

```
## [[1]]
## [1] "D" "e" "t" "r" "o" "i" "t"
```

Functions to find patterns in strings

```
x <- c("Philadelphia", "Austin")
```

- finding a pattern in a vector of strings using grep()
- this function returns the index of component
- where you can find the pattern

```
grep(pattern = "del", x)
```

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

```
grep(pattern = "stin", x)
```

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

```
grep(pattern =  "w", x)
```

```
## integer(0)
```

```
grep(pattern =  "a", x)
```

to ignore the case

```r
grep(pattern =  "a", x, ignore.case = TRUE)
```

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

- to get the value instead of the index we set value = TRUE

```r
grep(pattern = "del", x, value = TRUE)
```

```
## [1] "Philadelphia"
```

```r
grep(pattern = "stin", x, value = TRUE)
```

```
## [1] "Austin"
```

The function grepl()

- the function grepl() returns logical values
- TRUE if the pattern is there and FALSE otherwise

```
grepl(pattern = "del", x)
```

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

```
grepl(pattern = "stin", x)
```

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

```
grepl(pattern = "w", x)
```

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

...

```r
grepl(pattern = "a", x)
```

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

```r
grepl(pattern = "a", x, ignore.case = TRUE)
```

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

- we can write these two functions in a simpler way

```r
grep("del", x)
```

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

```r
grepl("del", x)
```

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

The regexpr() function

- regexpr() returns the first position where
- the pattern can be found

```
# regexpr("hil", x)
# regexpr("stin", x)
# regexpr("w", x)
# regexpr("a", x)
# regexpr("a", x, ignore.case = TRUE)
```

...

- gregexpr() does the same thing as regexpr()
- only it returns a more complex list

```
# gregexpr("hil", x)
# gregexpr("stin", x)
# gregexpr("w", x)
# gregexpr("a", x)
# gregexpr("a", x, ignore.case = TRUE)
```

The function regexec()

- Another similar function is regexec()

```
# regexec("hil", x)
# regexec("stin", x)
# regexec("a", x, ignore.case = TRUE)
```

FUNCTIONS TO REPLACE PATTERNS IN STRINGS

- there are two important functions that find a pattern
- and replace it with another string
- sub() and gsub()
- sub() replaces the first occurence of the pattern in each component

```
x <- c("Massachussets", "Russel")
sub("ss", "dd", x)
```

```
## [1] "Maddachussets" "Ruddel"
```

```
sub("abc", "xyz", x)
```

```
## [1] "Massachussets" "Russel"
```

- gsub() replaces all the occurences of the pattern in each component

```
gsub("ss", "dd", x)
```

```
## [1] "Maddachuddets" "Ruddel"
```

REGULAR EXPRESSIONS

- A regular expression is a sequence of characters
- used to define a pattern
- these expressions are used in the functions
- that find or replace patterns in strings
- grep(), grepl(), regexpr(), gregexpr(), regexec()
- sub(), gsub()
- they can be also used in other functions like strsplit()
- find the components in a character vector
- that contain at least one of the letters l or d

...

```r
grep("[ld]", c("Philadelphia", "Milwaukee", "Boston"),
     value = TRUE)
```

```
## [1] "Philadelphia" "Milwaukee"
```

```r
grepl("[ld]", c("Philadelphia", "Milwaukee", "Boston"))
```

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

Replace the l and d letters with a $

```
gsub("[ld]", "$", c("Philadelphia", "Milwaukee",
                    "Boston"))
```

```
## [1] "Phi$a$e$phia" "Mi$waukee"    "Boston"
```

```
### replace the l and d letters with three asterisks

gsub("[ld]", "***", c("Philadelphia", "Milwaukee",
                      "Boston"))
```

```
## [1] "Phi***a***e***phia" "Mi***waukee"        "Boston"
```

...

- Find the components in a character vector
- that contain any other letters than l or d

```
grep("[^ld]", c("Philadelphia", "Milwaukee",
                "Boston"), value = TRUE)
```

```
## [1] "Philadelphia" "Milwaukee"    "Boston"
```

```
grepl("[^ld]", c("Philadelphia", "Milwaukee",
                 "Boston"))
```

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

```
grep("[^top]", c("stop", "pause", "top"),
     value = TRUE)
```

```
## [1] "stop"  "pause"
```

...

- find the components in a character vector
- that contain any of the characters 2 or 5

```
grep("[25]", c("as148", "tm254", "wd570"),
     value = TRUE)
```

```
## [1] "tm254" "wd570"
```

- find the components in a character vector
- that contain any other characters than 2 or 5

```
grep("[^25]", c("as148", "25", "wd570"),
     value = TRUE)
```

```
## [1] "as148" "wd570"
```

- find the components in a character vector
- that contain any of the characters in the 2-5 interval

```
grep("[2-5]", c("as148", "tm254", "wd189"),
     value = TRUE)
```

```
## [1] "as148" "tm254"
```

...

- find the components in a character vector
- that contain any other character than the
- characters in the 2-5 interval

```
grep("[^2-5]", c("as148", "234", "167"),
     value = TRUE)
```

```
## [1] "as148" "167"
```

...

- the period (.) replaces any character

```r
x <- c("target", "window", "store", "stairs")
```

- find the components that contain the sequence
- t - any character - r

```r
grep("t.r", x, value = TRUE)
```

```
## [1] "target" "store"
```

...

- find the components that contain the sequence
- t - any two characters - r

```
grep("t..r", x, value = TRUE)
```

```
## [1] "stairs"
```

- to represent a period we precede it with two backslashes (\\.)

```
x <- c("bnm", "as.d", "qwe.")
### find the components that contain a period
grep("\\.", x, value = TRUE)
```

```
## [1] "as.d" "qwe."
```

...

- find the components that contain at least a digit

```
x <- c("stop", "wait35", "4abc")
grep("\\d", x, value = TRUE)
```

```
## [1] "wait35" "4abc"
```

- find the components that contain other characters
- than digits

```
x <- c("stop", "wait35", "789")
grep("\\D", x, value = TRUE)
```

```
## [1] "stop"   "wait35"
```

...

- find the components that contain at least a space

```
x <- c("abc", "d ef", "ghi  ")
grep("\\s", x, value = TRUE)
```

```
## [1] "d ef"  "ghi  "
```

- find the components that contain other characters
- than spaces

```
x <- c("abc", "d ef", "    ")
grep("\\S", x, value = TRUE)
```

```
## [1] "abc"  "d ef"
```

...

- find the components that contain words
- a word is a string that contains at least
- one alpha-numeric character (letter, number and underscore)
- N.B. the underscore (_) is alpha-numeric character, while the dash (-) is not

```
x <- c("stop", "stop12", "456", "abc ",
       "abc-_","", "4$#", "$&#", "#@_", "#@-")
grep("\\w", x, value = TRUE)
```

```
## [1] "stop"   "stop12" "456"    "abc "   "abc-_"  "4$#"    "
```

...

```
# - "stop", "stop12", "456", "abc ", "abc-_" are words
# - "4$#" is a word (it contains an alpha-numeric character)
# - "" is not a word (it is an empty string)
# - "#@_" is a word (it contains the underscore, which is alph
# - "$&#" is not a word (it only contains special characters)
# - "#@-" is not a word (because the dash is not alpha-numeri
```

...

- find the components that contain non-words
- a non-word is a string that does not contain any
- alpha-numeric character (only spaces and special characters)

```
x
```

```
##  [1] "stop"    "stop12" "456"     "abc "    "abc-_"   ""
##  [8] "$&#"     "#@_"    "#@-"
```

```
grep("\\W", x, value = TRUE)
```

```
## [1] "abc "  "abc-_" "4$#"   "$&#"   "#@_"   "#@-"
```

...

```
# - "stop", "stop12" and "456" do NOT contain non-words
# - (all their characters are alpha-numeric)
# - "" is an empty string, so it does NOT contain non-words
# - "abc ", "abc-_" do contain non-words (space and dash, resp
# - "4$#", "$&#", "#@_" and "#@-" do contain non-words
# - (special characters)
```