# R Basics

# Why R?

- It's free!
- It runs on a variety of platforms including Windows, Unix and MacOS.
- It provides an unparalleled platform for programming new statistical methods in an easy and straightforward manner.
- It contains advanced statistical routines not yet available in other packages.
- It has state-of-the-art graphics capabilities.

# How to download?

- Google it using R or CRAN

(Comprehensive R Archive Network)

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

or

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

# R Overview

R is a comprehensive statistical and graphical programming language and is a dialect of the S language:

1988 - S2: RA Becker, JM Chambers, A Wilks
1992 - S3: JM Chambers, TJ Hastie
1998 - S4: JM Chambers

R: initially written by Ross Ihaka and Robert Gentleman at Dep. of Statistics of U of Auckland, New Zealand during 1990s.

Since 1997: international “R-core” team of 15 people with access to common CVS archive.

# R Overview

You can enter commands one at a time at the command prompt (>) or run a set of commands from a source file.

There is a wide variety of data types, including vectors (numerical, character, logical), matrices, data frames, and lists.

To quit R, use

>q()

# R Overview

Most functionality is provided through built-in and user-created functions and all data objects are kept in memory during an interactive session.

Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed

# R Interface

Start the R system, the main window (RGui) with a sub window (R Console) will appear

In the `Console' window the cursor is waiting for you to type in some R commands.

# Your First R Session

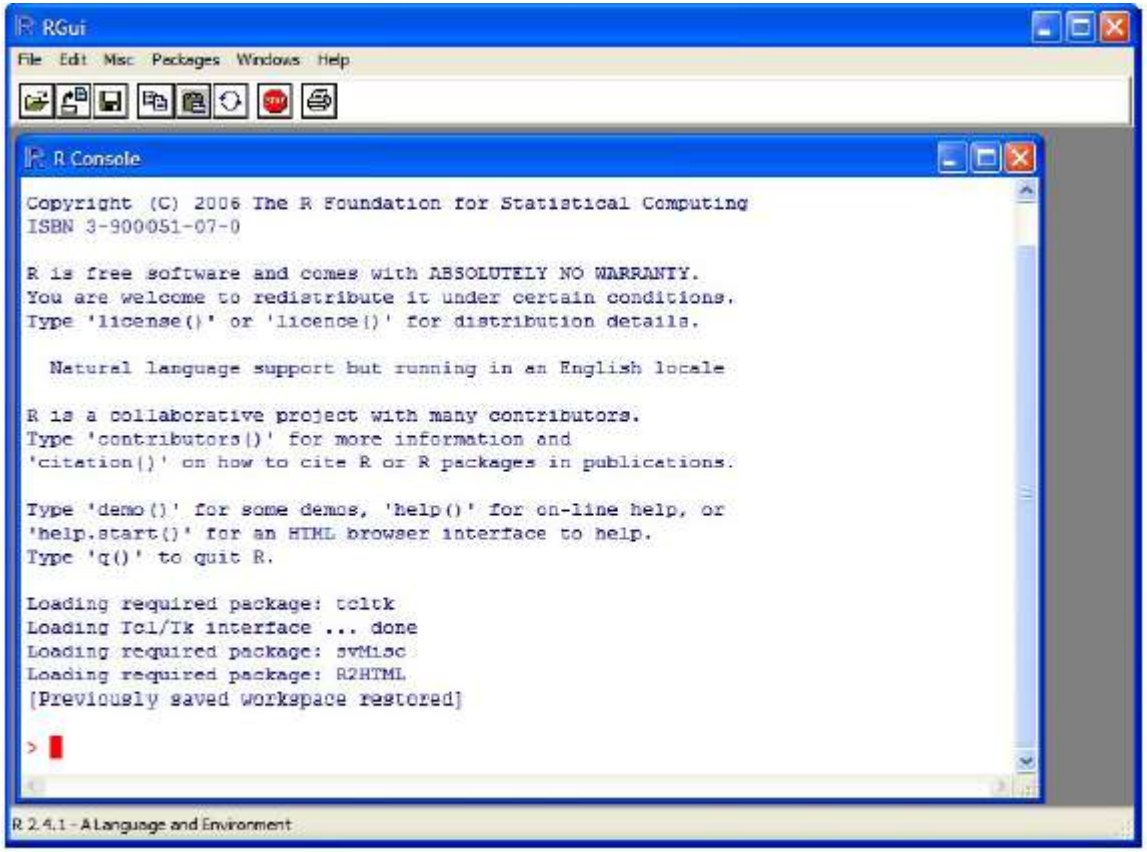
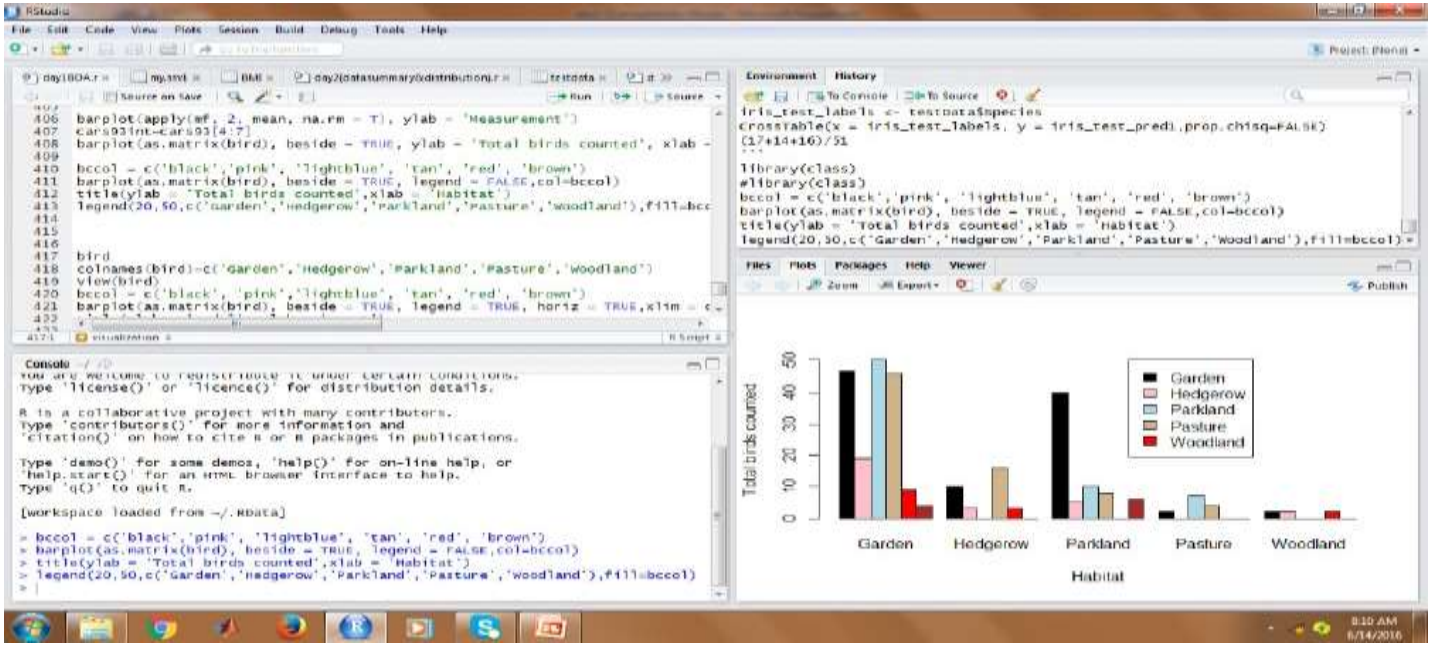



Figure 1.1: The R system on Windows

- **R Studio:** R Studio is an Integrated Development Environment (IDE) for R Language with advanced and more user-friendly GUI. R Studio allows the user to run R in a more user-friendly environment. It is open-source (i.e.free) and available at [http://www.rstudio.com/.](http://www.rstudio.com/)



# R Introduction

- Results of calculations can be stored in objects using the assignment operators:
  - An arrow (<-) formed by a smaller than character and a hyphen without a space!
  - The equal character (=).

# R Introduction

- These objects can then be used in other calculations. To print the object just enter the name of the object. There are some restrictions when giving an object a name:
  - Object names cannot contain `strange' symbols like !, +, -, #.
  - A dot (.) and an underscore ( ) are allowed, also a name starting with a dot.
  - Object names can contain a number but cannot start with a number.
  - R is case sensitive, X and x are two different objects, as well as temp and temP.

# An example

```
> # An example
> x <- c(1:10)
> x[(x>8) | (x<5)]
> # yields 1 2 3 4 9 10
> # How it works
> x <- c(1:10)
> x
>1 2 3 4 5 6 7 8 9 10
> x > 8
> F F F F F F F F T T
> x < 5
> T T T T F F F F F F
> x > 8 | x < 5
> T T T T F F F F T T
> x[c(T,T,T,T,F,F,F,F,T,T)]
> 1 2 3 4 9 10
```

# R Introduction

- To list the objects that you have in your current R session use the function ls or the function objects.

```
> ls()
[1] "x" "y"
```

- So to run the function ls we need to enter the name followed by an opening ( and a closing ). Entering only ls will just print the object, you will see the underlying R code of the the function ls. Most functions in R accept certain arguments. For example, one of the arguments of the function ls is pattern. To list all objects starting with the letter x:

```
> x2 = 9
> y2 = 10
> ls(pattern="x")
[1] "x2"
>x1 <- 1 # Create data objects
>x2 <- 2
>x3 <- 3
>ls()
[1] "x1" "x2" "x3"
```

# R Warning !

R is a case sensitive language.
FOO, Foo, and foo are three different objects

# R Introduction

```
> x = sin(9)/75
> y = log(x) + x^2
> x
[1] 0.005494913
> y
[1] -5.203902
> m <- matrix(c(1,2,4,1), ncol=2)
> m
> [,1] [,2]
[1,] 1 4
[2,] 2 1
> solve(m)
      [,1]         [,2]
[1,] -0.1428571 0.5714286
[2,] 0.2857143 -0.1428571
```

# R Workspace

```
getwd() # print the current working directory

ls()  # list the objects in the current workspace
setwd(mydirectory)   # change to mydirectory
setwd("c:/docs/mydir")
```

# R Help

Once **R** is installed, there is a comprehensive built-in help system. At the program's command prompt you can use any of the following:

help.start() # general help
help(*foo*) # help about function *foo*
?*foo* # same thing
apropos("*foo*") # list all function containing string foo
example(*foo*) # show an example of function *foo*

# R Datasets

**R** comes with a number of sample datasets that you can experiment with. Type

**> data( )**

to see the available datasets. The results will depend on which packages you have loaded. Type

**help(***datasetname***)**

for details on a sample dataset.

# R Packages

- One of the strengths of R is that the system can easily be extended. The system allows you to write new functions and package those functions in a so called `R package' (or `R library'). The R package may also contain other R objects, for example data sets or documentation. There is a lively R user community and many R packages have been written and made available on CRAN for other users. Just a few examples, there are packages for portfolio optimization, drawing maps, exporting objects to html, time series analysis, spatial statistics and the list goes on and on.

# R Packages

- When you download R, already a number (around 30) of packages are downloaded as well. To use a function in an R package, that package has to be attached to the system. When you start R not all of the downloaded packages are attached, only seven packages are attached to the system by default. You can use the function search to see a list of packages that are currently attached to the system, this list is also called the search path.

```
> search()
[1] ".GlobalEnv" "package:stats" "package:graphics"
[4] "package:grDevices" "package:datasets" "package:utils"
[7] "package:methods" "Autoloads" "package:base"
```

# R Packages

To attach another package to the system you can use the menu or the library function. Via the menu:

Select the `Packages' menu and select `Load package...', a list of available packages on your system will be displayed. Select one and click `OK', the package is now attached to your current R session. Via the library function:

```
> library(MASS)
> shoes
$A
[1] 13.2 8.2 10.9 14.3 10.7 6.6 9.5 10.8 8.8 13.3
$B
[1] 14.0 8.8 11.2 14.2 11.8 6.4 9.8 11.3 9.3 13.6
```

# R Packages

- The function library can also be used to list all the available libraries on your system with a short description. Run the function without any arguments

```
> library()
Packages in library 'C:/PROGRA~1/R/R-25~1.0/library':
base                    The R Base Package
Boot            Bootstrap R (S-Plus) Functions  (Canty)
class                   Functions for Classification
cluster                 Cluster Analysis Extended Rousseeuw  et al.
codetools               Code Analysis Tools for R
datasets                The R Datasets Package
DBI                     R Database Interface
foreign                 Read Data Stored by Minitab, S, SAS,
      SPSS, Stata, Systat, dBase, ...
graphics                The R Graphics Package
```

# Lecture 2: Data Input

# Data Types

**R** has a wide variety of data types including scalars, vectors (numerical, character, logical), matrices, dataframes, and lists.

# Vectors

- A vector in R is a one-dimensional array that can contain elements of the **same data type.**
- **To create a vector, you can use the c() function.**

For example:

- Numeric vector:1) my_vector <- c(1, 2, 3, 4, 5)

  2)a <- c(1,2,5.3,6,-2,4)
- Character vector:1) fruits <- c("apple", "banana", "cherry")

  2) b <- c("one","two","three")
- logical vector:c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE)
- Refer to elements of a vector using subscripts.

  a[c(2,4)] # 2nd and 4th elements of vector

- **any(): The any() function returns TRUE if at least one element in a logical vector is TRUE.**

**For example:**

```
logical_vector <- c(TRUE, FALSE, FALSE)
result <- any(logical_vector) # Returns TRUE
```

- **all(): The all() function returns TRUE if all elements in a logical vector are TRUE.**

**For example**

```
logical_vector <- c(TRUE, TRUE, TRUE)
result <- all(logical_vector) # Returns TRUE
```

- **which(): The which() function returns the indices of elements in a logical vector that are TRUE.**

**For example:**

```
logical_vector <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
indices <- which(logical_vector) # Returns indices 1, 3, 4
```

- To sort a vector in descending order in R, you can use the **sort() function with the decreasing parameter set to TRUE.**

For example:

1. vec <- c(5, 2, 8, 1, 3)
2. sorted_vec <- sort(vec, decreasing = TRUE) will give you a vector sorted in descending order.

- **length: The length() function is used to determine the length of a vector, which is the number of elements it contains.**

**For example:**

```
my_vector <- c(4, 7, 2, 9, 5)
vector_length <- length(my_vector) # Returns the length of the vector, which is 5
```

- **seq: The seq() function in R is used to create sequences of numbers. It can be used to generate a sequence from a starting value to an ending value, with a specified increment.**

For example:

sequence <- seq(1, 10, by = 2) # Creates a sequence: 1, 3, 5, 7, 9

- **rep: The rep() function is used to replicate elements in a vector.**

It takes two main arguments: the vector to be replicated and the number of times to repeat it.

For example:

vector <- c(1, 2, 3)

repeated_vector <- rep(vector, times = 3)

# Repeats the vector three times: 1, 2, 3, 1, 2, 3, 1, 2, 3

# Matrices

All columns in a matrix must have the same mode(numeric, character, etc.) and the same length.

**To create a matrix**:

The general format is

mymatrix <- **matrix(***vector*, **nrow=***r*, **ncol=***c*, **byrow=***FALSE*,**dimnames=list(***char_vector_rownames*, *char_vector_colnames***))**

**byrow=TRUE** indicates that the matrix should be filled by rows. **byrow=FALSE** indicates that the matrix should be filled by columns (the default).

**dimnames** provides optional labels for the columns and rows.

- cbind and rbind are functions in R used to combine vectors or matrices into a matrix.
- cbind combines objects **by columns,**
- rbind combines objects **by rows.**
- **For example:**

x <- c(1, 2, 3)

y <- c(4, 5, 6)

combined <- cbind(x, y) will create a 3x2 matrix with x and y as columns.

combined <- rbind(x, y) will create a 3x2 matrix with x and y as rows.

- The diag() function in R is used to create a diagonal matrix or extract the diagonal elements from a matrix.

For example:

- To create a diagonal matrix: diag_matrix <- diag(1:4)
- To extract the diagonal elements from a matrix:

```
mat<-matrix(1:9,nrow=3)
diagonal_elements <- diag(mat)
```

- To find the transpose and inverse of a matrix in R:

```
mat<-matrix(1:9,nrow=3)
```

- Transpose: t(mat) #where mat is your matrix.
- Inverse: solve(mat) #where mat is a square, invertible matrix.

- To find the dimension of a matrix in R, you can use the **dim() function.**
- **For example:**

```
mat <- matrix(1:6, nrow = 2, ncol = 3)
dim(mat) #will return the dimensions as 2 3, indicating it's a 2x3 matrix.
```

# Example

mat <- matrix(c(4.3, 3.1, 8.2, 8.2, 3.2, 0.9, 1.6, 6.5), nrow = 4, ncol = 2, **byrow = TRUE)**

Output

```
     [,1] [,2]
[1,]  4.3  3.1
[2,]  8.2  8.2
[3,]  3.2  0.9
[4,]  1.6  6.5
```

- To omit and overwrite elements in a matrix, you can simply assign new values to the elements you want to modify. For example, to omit the element in the second row and third column and overwrite an element:

```
mat <- matrix(1:9, nrow = 3)
mat[2, 3] <- NA # Omit the element
mat[1, 1] <- 99 # Overwrite an element
```

# Matrix Operations and Algebra

**1)MatrixTranspose**

For any m x n matrix A, its transpose, A ,is then m matrix obtained by writing either its columns as rows or its rows as columns

$$\text{If } A = \begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix}, \text{ then } A^{\top} = \begin{bmatrix} 2 & 6 \\ 5 & 1 \\ 2 & 4 \end{bmatrix}.$$

# Matrix Operations and Algebra

**2)** IdentityMatrix

The identity matrix written as Im is a particular kind of matrix used in mathematics. It's a square m X m matrix with ones on the dagonal and zeros elsewhere

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
R> A <- diag(x=3)
R> A
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

**3)** Scalar Multiple of a Matrix

A scalar value is just a single, univariate value. Multiplication of any matrix A by a scalar value a results in a matrix in which every individual element is multiplied by a.

$$2 \times \begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 10 & 4 \\ 12 & 2 & 8 \end{bmatrix}$$

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> a <- 2
R> a*A
     [,1] [,2] [,3]
[1,]    4   10    4
[2,]   12    2    8
```

**4)** Matrix addition and subtraction

Addition or subtraction of two matrices of equal size is also performed in an element-wise fashion. Corresponding elements are added or subtracted from one another, depending on the operation.

$$\begin{bmatrix} 2 & 6 \\ 5 & 1 \\ 2 & 4 \end{bmatrix} - \begin{bmatrix} -2 & 8.1 \\ 3 & 8.2 \\ 6 & -9.8 \end{bmatrix} = \begin{bmatrix} 4 & -2.1 \\ 2 & -7.2 \\ -4 & 13.8 \end{bmatrix}$$

```
R> A <- cbind(c(2,5,2),c(6,1,4))
R> A
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
R> B <- cbind(c(-2,3,6),c(8.1,8.2,-9.8))
R> B
     [,1] [,2]
[1,]   -2  8.1
[2,]    3  8.2
[3,]    6 -9.8
R> A-B
```

Matrice

```
     [,1] [,2]
[1,]    4 -2.1
[2,]    2 -7.2
[3,]   -4 13.8
```

**5)** Matrix Multiplication

In order to multiply two matrices A and B of size m X n and p X q, it must be true that n=p. The resulting matrix A . B will have the size m X q. The elements of the product are computed in a row-by-column fashion, where the value at position(AB)i,j is computed by element-wise multiplication of the entries in row i of A by the entries in column j of B, summing the result.

$$\begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & -3 \\ -1 & 1 \\ 1 & 5 \end{bmatrix}$$

$$= \begin{bmatrix} 2_{\times}3 + 5_{\times}(-1) + 2_{\times}1 & 2_{\times}(-3) + 5_{\times}(1) + 2_{\times}5 \\ 6_{\times}3 + 1_{\times}(-1) + 4_{\times}1 & 6_{\times}(-3) + 1_{\times}(1) + 4_{\times}5 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 9 \\ 21 & 3 \end{bmatrix}$$

## **6)** Matrix Inversion

Some square matrices can be inverted. The inverse of a matrix A is denoted A−1. An invertible matrix satisfies the following equation:

$$A\,A^{-1} = I_m$$

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & -0.5 \\ -2 & 1.5 \end{bmatrix}$$

```
R> A <- matrix(data=c(3,4,1,2),nrow=2,ncol=2)
R> A
     [,1] [,2]
[1,]    3    1
[2,]    4    2
R> solve(A)
     [,1] [,2]
[1,]    1 -0.5
[2,]   -2  1.5
```

# Arrays

R Arrays consist of all elements of the same data type. Vectors are supplied as input to the function and then create an array based on the number of dimensions.

You can create arrays in R using the array() function. An array is a multi-dimensional structure that can hold data of the same type.

- **Syntax:**

array(data, dim = (nrow, ncol, nmat), dimnames=names)

**where**

- nrow: Number of rows
- ncol : Number of columns
- nmat: Number of matrices of dimensions nrow * ncol
- dimnames : Default value = NULL.
- Here's an example of creating a 3x3x3 array:

```
data <- 1:27 # Vector with 27 elements
my_array <- array(data, dim = c(3, 3, 3))
```

# Multidimensional Arrays

- Just as a matrix (a “rectangle” of elements) is the result of increasing the dimension of a vector (a “line” of elements), the dimension of a matrix can be increased to get more complex data structures.
- three-dimensional array can be considered to be a collection of equally dimensioned matrices, providing you with a rectangular prism of elements.
- Figure 3-3 illustrates a three-row, four-column, two-layer (3 4 2) array.

# Multidimensional Arrays

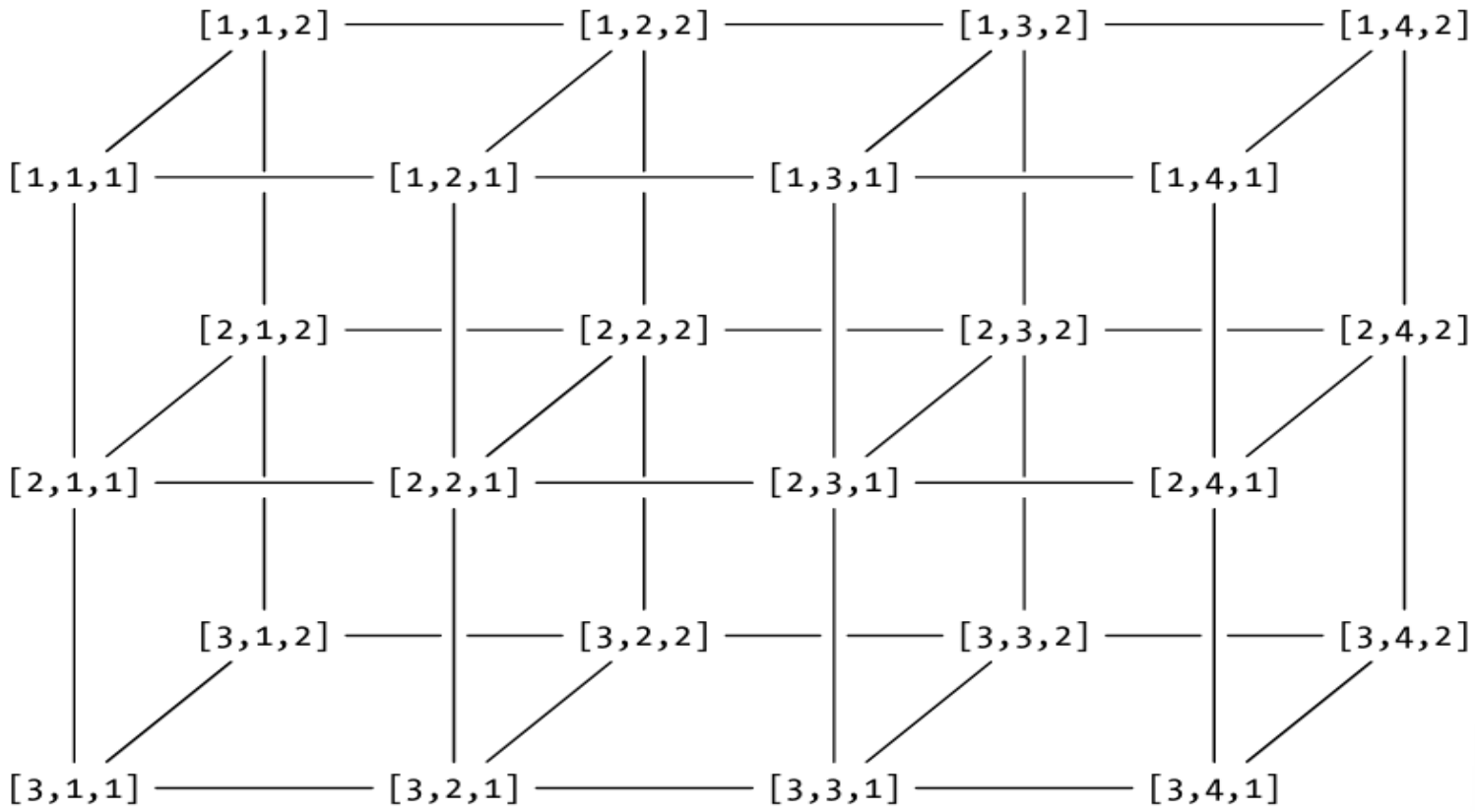



*Figure 3-3: A conceptual diagram of a 3 × 4 × 2 array. The index of each element is given at the corresponding position. These indexes are provided in the strict order of* `[row,column,layer]`.

# Multidimensional Arrays

**Definition**

To create these data structures in R, use the array function and specify the individual elements in the data argument as a vector. Then specify size in the dim argument as another vector with a length corresponding to the number of dimensions.

Note that array fills the entries of each layer with the elements in data in a strict column-wise fashion, starting with the first layer.

# Multidimensional Arrays

```
R> AR <- array(data=1:24,dim=c(3,4,2))
R> AR
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

## Subsets, Extractions, and Replacements

Suppose you want the second row of the second layer of the previously created array AR. You just enter these exact dimensional locations of AR in square brackets.

```
R> AR[2,,2]
 [1] 14 17 20 23
```

The desired elements have been extracted as a vector of length 4. If you want specific elements from this vector, say the third and first, in that order, you can call the following:

```
R> AR[2,c(3,1),2]
 [1] 20 14
```

## Subsets, Extractions, and Replacements

- To extract the first rows of both layers of AR, you enter this:

```
R> AR[1,,]


     [,1] [,2]
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

# Data frames

- In R, a data frame is a two-dimensional structure that stores data in rows and columns, similar to a table in a database or a spreadsheet. Each column in a data frame can have a different data type, and it can store various types of data such as numeric, character, logical, etc.
- # Creating the data frame

```
df <- data.frame(
Person = c("Peter", "Lois", "Meg", "Chris", "Stewie"),
Age = c(42, 40, 17, 14, 1),
Sex = c("M", "F", "F", "M", "M")
)
# Viewing the created data frame
print(df)
```

OUTPUT

```
  Person Age Sex
1  Peter  42   M
2   Lois  40   F
3    Meg  17   F
4  Chris  14   M
5 Stewie   1   M
```

There are a variety of ways to identify the elements of a dataframe

```
myframe[3:5] # columns 3,4,5 of dataframe
myframe[c("ID","Age")] # columns ID and Age from dataframe
myframe$X1 # variable x1 in the dataframe
```

# Data frames

# Lists

An ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

```
# example of a list with 4 components -
    # a string, a numeric vector, a matrix, and a scaler
    w <- list(name="Fred", mynumbers=a, mymatrix=y, age=5.3)

    # example of a list containing two lists
    v <- c(list1,list2)
```

# Lists

list in R is a data structure that can hold a collection of objects of different types.
For

- example:

```
my_list <- list(name = "John", age = 30, scores = c(85, 90, 78))
```

- Identify elements of a list using the [[]] convention.

```
mylist[[2]] # 2nd component of the list
```

- List slicing can be done using indexing and subsetting.
- For example:-

```
my_list <- list("apple", "orange", "banana", "grape", "melon")
subset <- my_list[2:4] # Extracting a subset of elements from
                                    index 2 to 4 (inclusive)
print(subset)
```

# Factors

A factor is a data structure in R used to represent categorical variables. Factors are used to store data that can take on a limited, fixed number of values, known as levels Tell **R** that a variable is **nominal** by making it a factor.

The factor stores the nominal values as a vector of integers in the range [ 1... k ] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers.

```
# variable gender with 20 "male" entries and # 30 "female" entries
gender <- c(rep("male",20), rep("female", 30))
gender <- factor(gender) # stores gender as 20 1s and 30 2s and associates#
1=female, 2=male internally (alphabetically)

# R now treats gender as a nominal variable
summary(gender)
```

OUTPUT

```
female   male
    30     20
```

- The levels() function in R is used with factors to get the levels distinct categories of a factor variable.
- For example:

```
gender <- factor(c("Male", "Female", "Male", "Male", "Female"))
gender_levels <- levels(gender) #will return a vector of levels: "Female" "Male".
```

- **cat: The cat() function in R is used to concatenate and print values together. It prints the values without any separators by default.**

```
cat("Hello", "World")
# Output: Hello World
```

- **paste: The paste() function is used to concatenate strings or other objects with a specified separator.**

```
result <- paste("Hello", "World", sep = ", ")
# Result: "Hello, World"
```

# Useful Functions

```
length(object) # number of elements or components
str(object)    # structure of an object
class(object)  # class or type of an object
names(object)  # names
c(object,object,...)  # combine objects into a vector
cbind(object, object, ...) # combine objects as columns
rbind(object, object, ...) # combine objects as rows
ls()       # list current objects
rm(object) # delete an object
newobject <- edit(object) # edit copy and save a newobject
fix(object)               # edit in place
```

# Viewing Data

**There are a number of functions for listing the contents of an object or dataset.**

```
# list objects in the working environment
    ls()
# list the variables in mydata
    names(mydata)
# list the structure of mydata
    str(mydata)
# list levels of factor v1 in mydata
    levels(mydata$v1)
# dimensions of an object
    dim(object)
```

# Viewing Data

**There are a number of functions for listing the contents of an object or dataset.**

```
# class of an object (numeric, matrix, dataframe, etc)
    class(object)
# print mydata
    mydata
# print first 10 rows of mydata
    head(mydata, n=10)
# print last 5 rows of mydata
    tail(mydata, n=5)
```

# Missing Data

- In **R**, missing values are represented by the symbol **NA** (not available) .
- Impossible values (e.g., dividing by zero) are represented by the symbol **NaN** (not a number).
- Unlike SAS(Statistical Analysis System), **R** uses the same symbol for character and numeric data.

**Testing for Missing Values**

```
is.na(x) # returns TRUE of x is missing
y <- c(1,2,3,NA)
is.na(y) # returns a vector (F F F T)
```

# Missing Data

**Recoding Values to Missing**

```
# recode 99 to missing for variable v1
# select rows where v1 is 99 and recode column v1
mydata[mydata$v1==99,"v1"] <- NA
```

**Excluding Missing Values from Analyses**

Arithmetic functions on missing values yield missing values.

```
x <- c(1,2,NA,3)
mean(x)         # returns NA
mean(x, na.rm=TRUE) # returns 2
```

# Missing Data

The function **complete.cases()** returns a logical vector indicating which cases are complete.

```
# list rows of data that have missing values
    mydata[!complete.cases(mydata),]
```

The function **na.omit()** returns the object with listwise deletion of missing values.

```
# create new dataset without missing data
    newdata <- na.omit(mydata)
```

# Date Values

**Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates.**

```
# use as.Date( ) to convert strings to dates
mydates <- as.Date(c("2007-06-22", "2004-02-13"))
# number of days between 6/22/07 and 2/13/04
    days <- mydates[1] - mydates[2]
```

**Sys.Date( ) returns today's date.**

**Date() returns the current date and time.**

# Date Values

**The following symbols can be used with the format( ) function to print dates.**

| Symbol | Meaning | Example |
|---|---|---|
| **%d** | day as a number (0-31) | 01-31 |
| **%a** | abbreviated weekday | Mon |
| **%A** | unabbreviated weekday | Monday |
| **%m** | month (00-12) | 00-12 |
| **%b** | abbreviated month | Jan |
| **%B** | unabbreviated month | January |
| **%y** | 2-digit year | 07 |
| **%Y** | 4-digit year | 2007 |

# Date Values

```
# print today's date
today <- Sys.Date()
format(today, format="%B %d %Y")
      "June 20 2007"
```

# Lecture 3: Data Manipulation

# Introduction

Each of these activities usually involve the use of **R**'s built-in operators (arithmetic and logical) and functions (numeric, character, and statistical). Additionally, you may need to use control structures (if-then, for, while, switch) in your programs and/or create your own functions. Finally you may need to convert variables or datasets from one type to another (e.g. numeric to character or matrix to dataframe).

# Arithmetic Operators

| 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 |

# Logical Operators

| 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 |
| **x \| y** | x OR y |
| **x & y** | x AND y |
| **isTRUE(x)** | test if x is TRUE |

# Control Structures

- **R** has the standard control structures you would expect. **expr** can be multiple (compound) statements by enclosing them in braces { }. It is more efficient to use built-in functions rather than control structures whenever possible.

# Control Structures

**if-else**

```
if (condition) {
  do_this
} else {
  do_that
}
```

Example

```
x <- 10

if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is 5 or less")
}
```

# Control Structures

**for**

```
for (variable in sequence) {
  do_this
}
```

Example

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

# Control Structures

**while**

```
while (condition) {
  do_this
}
```

Example

```
x <- 1
while (x <= 3) {
  print(x)
  x <- x + 1
}
```

# Control Structures

**switch**

```
switch(expression,
      case1 = result1,
      case2 = result2,
      ...)
```

Example

```
x <- 2
switch(x,
      "one",
      "two",
      "three")
```

# Control Structures

**ifelse**

ifelse(*test*,*yes*,*no*)

**Example**

```
x <- c(4, 6, 8)

result <- ifelse(x > 5, "big", "small")
print(result)
```

# Control Structures

- ```
  # transpose of a matrix
  # a poor alternative to built-in t() function

  mytrans <- function(x) {
    if (!is.matrix(x)) {
      warning("argument is not a matrix: returning NA")
      return(NA_real_)
    }
    y <- matrix(1, nrow=ncol(x), ncol=nrow(x))
    for (i in 1:nrow(x)) {
      for (j in 1:ncol(x)) {
        y[j,i] <- x[i,j]
      }
    }
  return(y)
  }
  ```
  ```

# Control Structures

- # try it
  z <- matrix(1:10, nrow=5, ncol=2)
  tz <- mytrans(z)

# R built-in functions

Almost everything in **R** is done through functions. Here I'm only referring to numeric and character functions that are commonly used in creating or recoding variables.

Note that while the examples on this page apply functions to individual variables, many can be applied to vectors and matrices as well.

# 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* |

# Character Functions

| Function | Description |
|---|---|
| **substr(***x***, start=***n1***, stop=***n2***)** | Extract or replace substrings in a character vector.<br>x <- "abcdef"<br>substr(x, 2, 4) is "bcd"<br>substr(x, 2, 4) <- "22222" is "a222ef" |
| **grep(***pattern***,** *x* **,**<br>**ignore.case**=FALSE**, fixed**=FALSE**)** | Search for *pattern* in *x*. If fixed =FALSE then *pattern* is a regular expression. If fixed=TRUE then *pattern* is a text string. Returns matching indices.<br>grep("A", c("b","A","c"), fixed=TRUE) returns 2 |
| **sub(***pattern***,** *replacement***,** *x***,**<br>**ignore.case** =FALSE**, fixed**=FALSE**)** | Find *pattern* in *x* and replace with *replacement* text. If fixed=FALSE then *pattern* is a regular expression.<br>If fixed = T then *pattern* is a text string.<br>sub("\\s",".","Hello There") returns "Hello.There" |
| **strsplit(***x***,** *split***)** | Split the elements of character vector *x* at *split*.<br>strsplit("abc", "") returns 3 element vector "a","b","c" |
| **paste(..., sep="")** | Concatenate strings after using *sep* string to seperate them.<br>paste("x",1:3,sep="") returns c("x1","x2" "x3")<br>paste("x",1:3,sep="M") returns c("xM1","xM2" "xM3")<br>paste("Today is", date()) |
| **toupper(***x***)** | Uppercase |
| **tolower(***x***)** | Lowercase |

# Stat/Prob Functions

- The following table describes functions related to probaility distributions. For random number generators below, you can use set.seed(1234) or some other integer to create reproducible pseudo-random numbers.

# Other Useful Functions

| Function | Description |
|---|---|
| **seq(***from* **,** *to***,** *by***)** | generate a sequence<br>indices <- seq(1,10,2)<br>#indices is c(1, 3, 5, 7, 9) |
| **rep(***x***,** *ntimes***)** | repeat *x n* times<br>y <- rep(1:3, 2)<br># y is c(1, 2, 3, 1, 2, 3) |
| **cut(***x***,** *n***)** | divide continuous variable in factor with *n* levels<br>y <- cut(x, 5) |

| Function | Description |
|---|---|
| **dnorm(***x***)** | normal density function (by default m=0 sd=1)<br># plot standard normal curve<br>x <- pretty(c(-3,3), 30)<br>y <- dnorm(x)<br>plot(x, y, type='l', xlab="Normal Deviate", ylab="Density", yaxs="i") |
| **pnorm(***q***)** | cumulative normal probability for q<br>(area under the normal curve to the right of q)<br>pnorm(1.96) is 0.975 |
| **qnorm(***p***)** | normal quantile.<br>value at the p percentile of normal distribution<br>qnorm(.9) is 1.28 # 90th percentile |
| **rnorm(***n***, m**=0**,sd**=1**)** | n random normal deviates with mean m<br>and standard deviation sd.<br>#50 random normal variates with mean=50, sd=10<br>x <- rnorm(50, m=50, sd=10) |
| **dbinom(***x***,** *size***,** *prob***)**<br>**pbinom(***q***,** *size***,** *prob***)**<br>**qbinom(***p***,** *size***,** *prob***)**<br>**rbinom(***n***,** *size***,** *prob***)** | binomial distribution where size is the sample size<br>and prob is the probability of a heads (pi)<br># prob of 0 to 5 heads of fair coin out of 10 flips<br>dbinom(0:5, 10, .5)<br># prob of 5 or less heads of fair coin out of 10 flips<br>pbinom(5, 10, .5) |
| **dpois(***x***,** *lamda***)**<br>**ppois(***q***,** *lamda***)**<br>**qpois(***p***,** *lamda***)**<br>**rpois(***n***,** *lamda***)** | poisson distribution with m=std=lamda<br>#probability of 0,1, or 2 events with lamda=4<br>dpois(0:2, 4)<br># probability of at least 3 events with lamda=4<br>1- ppois(2,4) |
| **dunif(***x***, min**=0**, max**=1**)**<br>**punif(***q***, min=0, max**=1**)**<br>**qunif(***p***, min**=0**, max**=1**)**<br>**runif(***n***, min**=0**, max**=1**)** | uniform distribution, follows the same pattern<br>as the normal distribution above.<br>#10 uniform random variates<br>x <- runif(10) |

| Function | Description |
|---|---|
| **mean(***x***, trim=**0**, na.rm=**FALSE**)** | mean of object x<br># trimmed mean, removing any missing values and<br># 5 percent of highest and lowest scores<br>mx <- mean(x,trim=.05,na.rm=TRUE) |
| **sd(***x***)** | standard deviation of object(x). also look at var(x) for variance and mad(x) for median absolute deviation. |
| **median(***x***)** | median |
| **quantile(***x***,** *probs***)** | quantiles where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0,1].<br># 30th and 84th percentiles of x<br>y <- quantile(x, c(.3,.84)) |
| **range(***x***)** | range |
| **sum(***x***)** | sum |
| **diff(***x***, lag=***1***)** | lagged differences, with lag indicating which lag to use |
| **min(***x***)** | minimum |
| **max(***x***)** | maximum |
| **scale(***x***, center=**TRUE**, scale=**TRUE**)** | column center or standardize a matrix. |

# Lecture 4: Non-Numeric Values

# Logical Values

Logical values (also simply called logicals) are based on a simple premise: a logical-valued object can only be either TRUE or FALSE. These can be inter preted as yes/no, one/zero, satisfied/not satisfied, and so on.

## True or false?

Logical values in R are written fully as TRUE and FALSE, but they are fre quently abbreviated as T or F. The abbreviated version has no effect on the execution of the code, so, for example, using decreasing=T is equivalent to decreasing=TRUE in the sort function.

# Logical Values

```
R> foo <- TRUE
R> foo
[1] TRUE
R> bar <- F
R> bar
[1] FALSE

R> baz <- c(T,F,F,F,T,F,T,T,T,F,T,F)
R> baz
 [1] TRUE FALSE FALSE FALSE TRUE FALSE TRUE
TRUE TRUE FALSE TRUE FALSE
```

# Logical Values

## A Logical Outcome: Relational Operators

Logicals are commonly used to check relationships between values. For example, you might want to know whether some number a is greater than a predefined threshold b. For this, you use the standard relational operators shown in Table 4-1, which produce logical values as results.

**Table 4-1:** Relational Operators

| Operator | Interpretation |
|---|---|
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

# Logical Values

Typically, these operators are used on numeric values:

```
R> 1==2
[1] FALSE
R> 1>2
[1] FALSE
R> (2-1)<=2
[1] TRUE
R> 1!=(2+3)
[1] TRUE
```

# Logical Values

Typically, these operators are used on numeric values:

```
R> foo <- c(3,2,1,4,1,2,1,-1,0,3)
R> bar <- c(4,1,2,1,1,0,0,3,0,4)
R> length(x=foo)==length(x=bar)
[1] TRUE
```

Nowconsider the following four evaluations:

```
R> foo==bar
 [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE
FALSE
R> foo<bar
[1] TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
R> foo<=bar
[1] TRUE FALSE TRUE FALSE TRUE FALSE FALSE TRUE TRUE TRUE
R> foo<=(bar+10)
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

# Logical Values

**Multiple Comparisons: Logical Operators**

Logicals are especially useful when you want to examine whether multiple conditions are satisfied. Often you'll want to perform certain operations only if a number of different conditions have been met.

**Table 4-2:** Logical Operators Comparing Two Logical Values

| Operator | Interpretation | Results |
|---|---|---|
| & | AND (element-wise) | TRUE & TRUE is TRUE<br>TRUE & FALSE is FALSE<br>FALSE & TRUE is FALSE<br>FALSE & FALSE is FALSE |
| && | AND (single comparison) | Same as & above |
| \| | OR (element-wise) | TRUE\|TRUE is TRUE<br>TRUE\|FALSE is TRUE<br>FALSE\|TRUE is TRUE<br>FALSE\|FALSE is FALSE |
| \|\| | OR (single comparison) | Same as \| above |
| ! | NOT | !TRUE is FALSE<br>!FALSE is TRUE |

# Logical Values

```
R> FALSE||((T&&TRUE)||FALSE)
 [1] TRUE
 R> !TRUE&&TRUE
 [1] FALSE
 R> (T&&(TRUE||F))&&FALSE
 [1] FALSE
 R> (6<4)||(3!=1)
 [1] TRUE
```

# Logical Values

**Logicals Are Numbers!**

Because of the binary nature of logical values, they're often represented with TRUE as 1 and FALSE as 0. In fact, in R, if you perform elementary numeric operations on logical values, TRUE is treated like 1, and FALSE is treated like 0.

```
R> TRUE+TRUE
 [1] 2
 R> FALSE-TRUE
 [1]-1
 R> T+T+F+T+F+F+T
 [1] 4

 R> 1&&1
 [1] TRUE
 R> 1||0
 [1] TRUE
 R> 0&&1
 [1] FALSE
```

# Logical Values

**Logical Subsetting and Extraction**

Logicals can also be used to extract and subset elements in vectors and other objects, in the same way as you've done so far with index vectors. Rather than entering explicit indexes in the square brackets, you can supply logical f lag vectors, where an element is extracted if the corresponding entry in the f lag vector is TRUE.

**R> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)**

If you wanted to extract the two negative elements, you could either enter myvec[c(2,10)], or you could do the following using logical flags:

**R> myvec[c(F,T,F,F,F,F,F,F,F,T)]**

**[1]-2.3-8.0**

**R> myvec<0**

**[1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE**

**R> myvec[myvec<0]**

**[1]-2.3-8.0**

# Characters

**Creating a String**

Character strings are indicated by double quotation marks, ". To create a string, just enter text between a pair of quotes.

```
R> foo <- "This is a character string!"
R> foo
[1] "This is a character string!"
R> length(x=foo)
[1] 1

#the total number of distinct strings
R> nchar(x=foo)
[1] 27

#Almost any combination of characters, including numbers, can be a valid character string.
R> bar <- "23.3"
R> bar
[1] "23.3"
```

# Characters

**Concatenation**

There are two main functions used to concatenate (or glue together) one or more strings: cat and paste. The difference between the two lies in how their contents are returned. The first function, cat, sends its output directly to the console screen and doesn't formally return anything. The paste function con catenates its contents and then returns the final character string as a usable R object.

```
R> qux <- c("awesome","R","is")
R> length(x=qux)
[1] 3
R> qux
[1] "awesome" "R“  "is“
```

```
R> cat(qux[2],qux[3],"totally",qux[1],"!")
R is totally awesome !
R> paste(qux[2],qux[3],"totally",qux[1],"!")
[1] "R is totally awesome !"
```

# Characters

**scape Sequences**

**R> cat("here is a string\nsplit\tto neww\b\n\n\tlines")**

here is a string
split	to new
	lines

**Table 4-3:** Common Escape Sequences for Use in Character Strings

| Escape sequence | Result |
|---|---|
| \n | Starts a newline |
| \t | Horizontal tab |
| \b | Invokes a backspace |
| \\ | Used as a single backslash |
| \" | Includes a double quote |

# Characters

Substrings and Matching

Pattern matching lets you inspect a given string to identify smaller strings within it.

The function substr takes a string x and extracts the part of the string between two character positions (inclusive), indicated with numbers passed as start and stop arguments.

```
R> foo <- "This is a character string!"
R> substr(x=foo,start=21,stop=27)
[1] "string!"
```