R programming

CHEN Xiao

Department of Mathematics

January 31, 2020

Overview

Introduction:

R is an integrated suite of software facilities for data manipulation, calculation and graphical display.

- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for interactive data analysis,
- graphical facilities for data analysis and display either directly at the computer or on hardcopy
- a well developed programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.

R is an interpreted computer language.

- Most user-visible functions are written in R itself, calling upon a smaller set of internal primitives.
- It is possible to interface procedures written in C, C+, or FORTRAN languages for efficiency, and to write additional primitives.
- System commands can be called from within R.

Why R?

- R is a programming and statistical language.
- R is used for data analysis and visualization.
  - The R distribution contains functionality for large number of statistical procedures.
  - R also has a large set of functions which provide a flexible graphical environment for creating various kinds of data presentations.
- R is simple and easy to learn, read and write.
- Free and open-source software (FOSS): Anyone is freely licensed to use, copy, study, and change the software in any way, and the source code is openly shared so that people are encouraged to voluntarily improve the design of the software.
- R has a very active and helpful online community - normally a quick search is all it takes to find that somebody has already solved the problem you're having.

Installation of R

- Go to the link- `https://cran.r-project.org/`
  Download and install R on your system.
- Download and install Rstudio on your system:
  `https://rstudio.com/products/rstudio/download/`
- RStudio is a tool that can help you do your work better and faster. In technical terms, RStudio is an integrated development environment (IDE) for R. It includes a console, syntax-highlighting editor that supports direct code execution, and a variety of robust tools for plotting, viewing history, debugging and managing your workspace.
- For many R developers this represents their preferred way of working with R. Working in the source editor makes it much easier to reproduce sequences of commands and to package commands for reuse as a function.

Resources

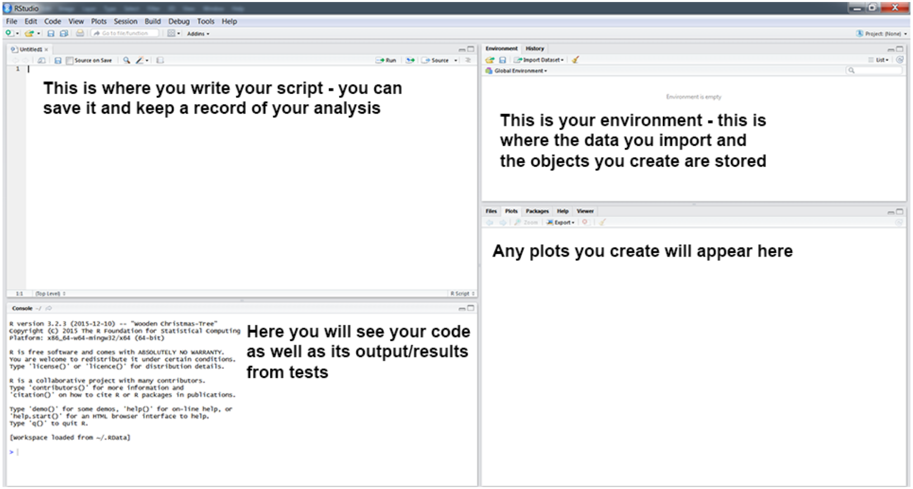
- R functions and datasets can be organized into a package. Package is essentially a library of functions for specific domain and helps to accomplish a specific for this domain set of tasks.
- CRAN and R homepage:
    - `http://www.r-project.org/` It is R's central homepage, giving information on the R project and everything related to it.
    - `http://cran.r-project.org/` It acts as the download area,carrying the software itself, extension packages, PDF manuals.
    - An Introduction to R:`https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf`
- R Site Search: `http://finzi.psych.upenn.edu/search.html` This search will allow you to search the contents of the R functions, package vignettes, and task views.
- R bloggers: Here you will find daily news and tutorials about R, contributed by hundreds of bloggers. `https://www.r-bloggers.com/`
- A hand-picked list of resources that we used when learning to code `https://ourcodingclub.github.io/links/`

Learning R

- Books:
    - **The Introduction to R** is highly recommended as a basic source of information on R. `https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf`
    - **The Art of R Programming: A Tour of Statistical Software Design**, by Norman Matloff, 2011. No Starch Press, 400 p.
    - **The R Book**, by Michael J. Crawley, 2012. Wiley, 1076 p.
- **Cookbook for R** (`http://www.cookbook-r.com`) has recipes for analyzing your data. The goal of the cookbook is to provide solutions to common tasks and problems in analyzing data.
- **Stack Overflow** (`http://stackoverflow.com/questions/tagged/r`) is a question and answer site for programmers. Users post questions, other users post answers, and these get voted up or down, so you can see what the community regards as the right answer. Stack Overflow is great for many languages, and the R community that uses it is growing.

Rstudio

Open RStudio. Click on File/New File/R script.



This is where you write your script - you can save it and keep a record of your analysis

This is your environment - this is where the data you import and the objects you create are stored

Any plots you create will appear here

Here you will see your code as well as its output/results from tests

```
R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

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

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

[workspace loaded from ~/.RData]

> 
```

Begin to write in your script

To install a package, type **install.packages(”package-name”)**. You only need to install packages once.
Once installed, you just need to load the packages using **library(package-name)**.

```
install.packages(”dplyr”)
library(dplyr)
# Note that there are quotation marks when installing a package, but
# not when loading it and remember that hashtags let you add useful
notes to your code!
```

Getting help

Details about a specific command whose name you know (input arguments, options, algorithm, results):

?t.test

or

help(t.test)

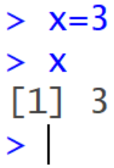
Objects

R works with objects, and there are many types. Objects store data, and they have commands that can operate on them, which depend the type and structure of data that is stored. A single number or string of text is the simplest object and is known as a scalar. [Note that a scalar in R is simply a vector of length one.]

- To give a value to an object use one of the two assignment operators: $"<-"$(arrow),$"="$.

```
> x = 3
> x <- 3
```

- To display the value of any object, type its name at the prompt.



```
> x=3
> x
[1] 3
> |
```

Functions

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

- An argument is a placeholder.Arguments:When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
  For example, arguments of **log** funciton:
  x : a numeric or complex vector.
  base : a positive or complex number: the base with respect to which logarithms are computed. Defaults to e.
  $log(x)$ computes logarithms, by default natural logarithms The general form $log(x, base)$ computes logarithms with *base* base.
- Functions usually return a value. You can assign this value to an object.

The maximum of two values is calculated with the max() function:

```
> max(9, 5)
Output: 9
>M< −max(9,5) #the returned valued 9 is stored in the object M.
```

Data types

Everything in R is an object.
There are 5 basic data types.

- character: "a", "swc"
- numeric:12.5,3.18. The numeric data type is for numeric values. It is the default data type for numbers in R.
- integer: 2L (the L tells R to store this as an integer)
- logical: TRUE, FALSE. The logical data type stores logical or boolean values.
- complex: 1+4i (complex numbers with real and imaginary parts)

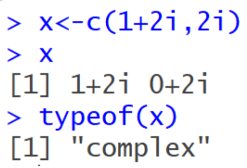
Elements of these data types may be combined to form data structures, such as atomic vectors. When we call a vector atomic, we mean that the vector only holds data of a single data type.

Vectors

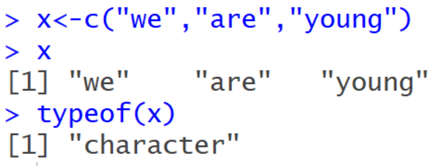
- Vectors are common data structures, and you will use them frequently.
- A vector is a series of values, which may be numeric or text, where all values have the same type
  - logical: a logical value.
  - integer: an integer (positive or negative). Many R programmers do not use this mode since every integer value can be represented as a double.
  - double: a real number stored in "double-precision floating point format." For example, 3.14.
  - complex: a complex number
  - character: a sequence of characters
- Vectors are created most easily with the c() function .

  $> x < -c(10.4, 5.6, 3.1, 6.4, 21.7)$
- A vector's type can be checked with the typeof() function.

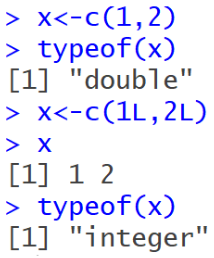
By default, when you create a numeric vector using the c() function it will produce a vector of double precision numeric values. To create a vector of integers using c() you must specify explicity by placing an L directly after each number.



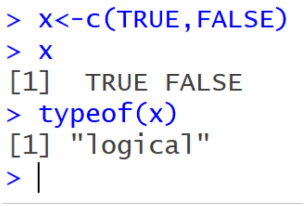
```
> x<-c(1+2i,2i)
> x
[1] 1+2i 0+2i
> typeof(x)
[1] "complex"
```



```
> x<-c("we","are","young")
> x
[1] "we"     "are"    "young"
> typeof(x)
[1] "character"
```



```
> x<-c(1,2)
> typeof(x)
[1] "double"
> x<-c(1L,2L)
> x
[1] 1 2
> typeof(x)
[1] "integer"
```



```
> x<-c(TRUE,FALSE)
> x
[1]  TRUE FALSE
> typeof(x)
[1] "logical"
> 
```

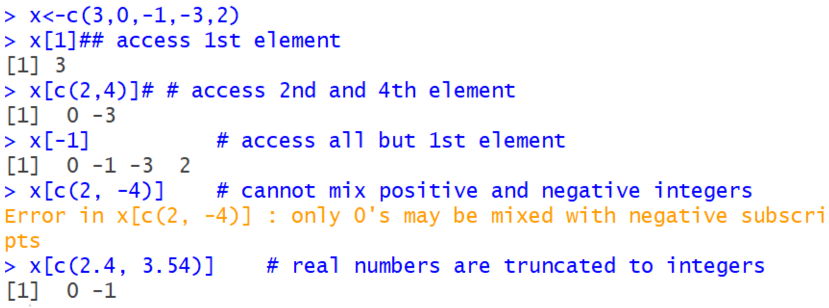
Retrieving

An element of a vector can be retrieved by using an index.

- Using integer vector as index.
  We can use a vector of integers as index to access specific elements.

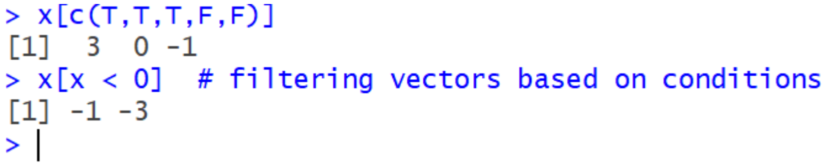
  We can also use negative integers to return all elements except that those specified.

  But we cannot mix positive and negative integers while indexing and real numbers, if used, are truncated to integers.



```
> x<-c(3,0,-1,-3,2)
> x[1]## access 1st element
[1] 3
> x[c(2,4)]# # access 2nd and 4th element
[1]  0 -3
> x[-1]          # access all but 1st element
[1]  0 -1 -3  2
> x[c(2, -4)]    # cannot mix positive and negative integers
Error in x[c(2, -4)] : only 0's may be mixed with negative subscri
pts
> x[c(2.4, 3.54)]    # real numbers are truncated to integers
[1]  0 -1
```

- Using logical vector as index
  When we use a logical vector for indexing, the position where the logical vector is TRUE is returned.

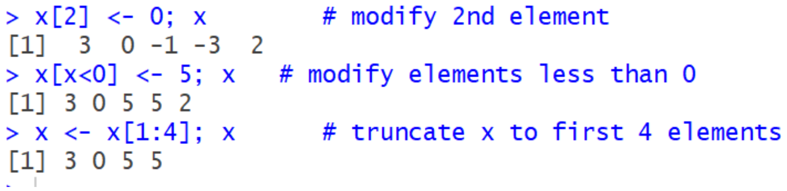


```
> x[c(T,T,T,F,F)]
[1]  3  0 -1
> x[x < 0]  # filtering vectors based on conditions
[1] -1 -3
> |
```

In the above example, the expression $x < 0$ will yield a logical vector (FALSE, FALSE,TRUE,TRUE FALSE) which is then used for indexing.

How to modify a vector in R?

We can modify a vector using the assignment operator.



```
> x[2] <- 0; x        # modify 2nd element
[1]  3  0 -1 -3  2
> x[x<0] <- 5; x   # modify elements less than 0
[1] 3 0 5 5 2
> x <- x[1:4]; x      # truncate x to first 4 elements
[1] 3 0 5 5
```

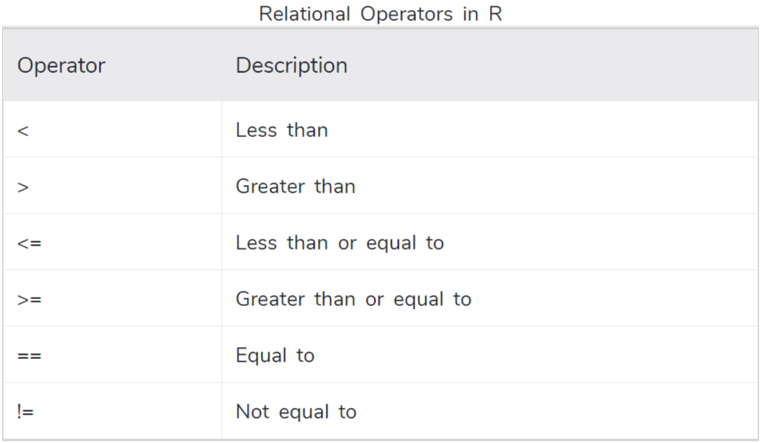
R Operators

Operators in R can mainly be classified into the following categories:

- Assignment operators
- Arithmetic operators
- Relational operators
- Logical operators

Relational operator

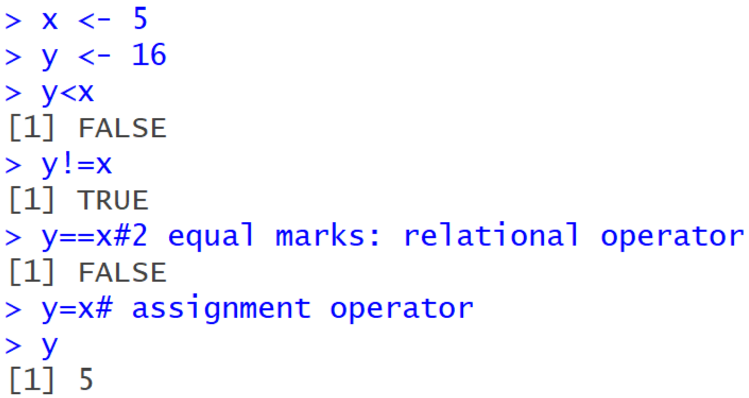
Relational Operator: It defines a relation between two entities. For example: $<, >, <=, ==$etc.


Relational Operators in R

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Relational operator

Relational Operator:

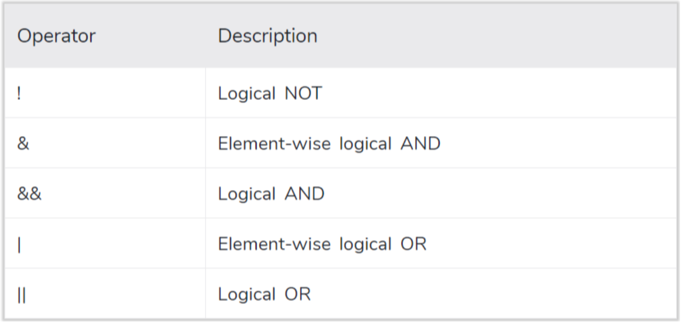


```
> x <- 5
> y <- 16
> y<x
[1] FALSE
> y!=x
[1] TRUE
> y==x#2 equal marks: relational operator
[1] FALSE
> y=x# assignment operator
> y
[1] 5
```

R Logical Operators

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

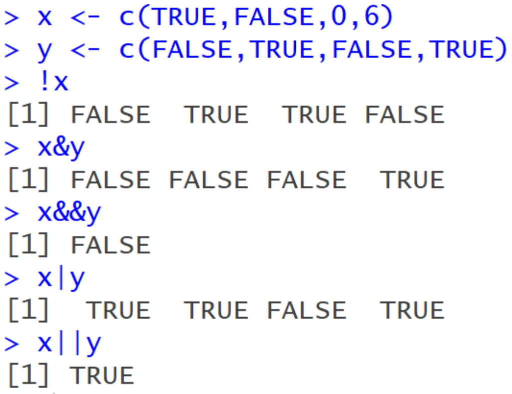
Logical Operators in R



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

R Logical Operators

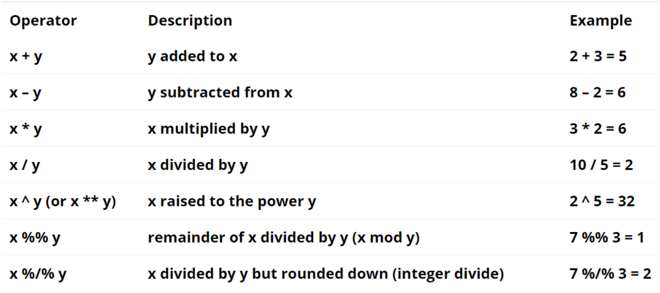
Operators & and | perform element-wise operation producing result having length of the longer operand.
But && and || examines only the first element of the operands resulting into a single length logical vector.
Zero is considered FALSE and non-zero numbers are taken as TRUE. An example run.



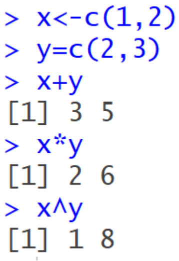
```
> x <- c(TRUE,FALSE,0,6)
> y <- c(FALSE,TRUE,FALSE,TRUE)
> !x
[1] FALSE  TRUE  TRUE FALSE
> x&y
[1] FALSE FALSE FALSE  TRUE
> x&&y
[1] FALSE
> x|y
[1]  TRUE  TRUE FALSE  TRUE
> x||y
[1] TRUE
```

Arithmetic Operators

The elementary arithmetic operators



Operator	Description	Example
x + y	y added to x	2 + 3 = 5
x – y	y subtracted from x	8 – 2 = 6
x * y	x multiplied by y	3 * 2 = 6
x / y	x divided by y	10 / 5 = 2
x ^ y (or x ** y)	x raised to the power y	2 ^ 5 = 32
x %% y	remainder of x divided by y (x mod y)	7 %% 3 = 1
x %/% y	x divided by y but rounded down (integer divide)	7 %/% 3 = 2



```
> x<-c(1,2)
> y=c(2,3)
> x+y
[1] 3 5
> x*y
[1] 2 6
> x^y
[1] 1 8
```

Common mathematical functions

Common mathematical functions:*log*, *exp*, *sin*, *cos*, *tan*, *sqrt*, and so on, all have their usual meaning.

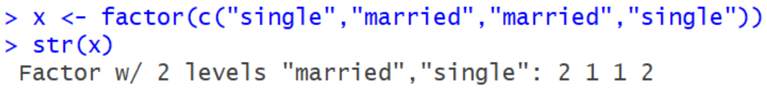
- **max** and **min** select the largest and smallest elements of a vector respectively.
- **length**(x) is the number of elements in*x*
- **sum**(x)gives the total of the elements in *x*
- **prod**(x) their product.
- Two statistical functions are **mean**(x) and **var**(x), which calculates the sample mean and sample variance.
- **sort**(x) returns a vector of the same size as x with the elements arranged in increasing order(see *order*()or *sort*.*list*())

Data structure

- Vector
- Factor
- Array and Matrix
- List
- Dataframe

Factor

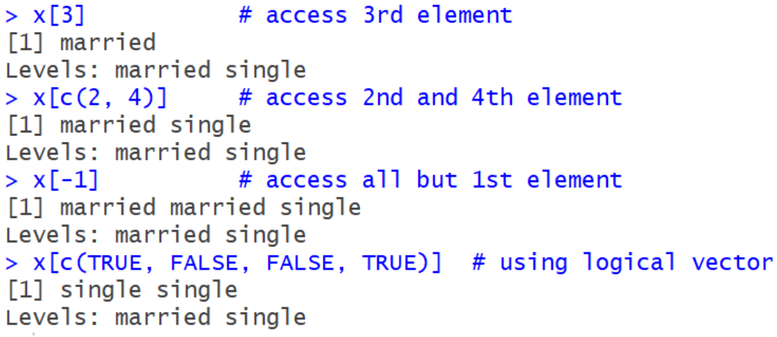
- Factors are similar to vectors, but they have a fixed number of levels and are ideal for expressing categorical variables.
- Levels of a factor can be checked using the levels() function.
- We can see from the following example that levels may be predefined even if not used.
- Factors are closely related with vectors. In fact, factors are stored as integer vectors. This is clearly seen from its structure.



```
> x <- factor(c("single","married","married","single"))
> str(x)
 Factor w/ 2 levels "married","single": 2 1 1 2
```

How to access compoments of a factor?

Accessing components of a factor is very much similar to that of vectors.



```
> x[3]           # access 3rd element
[1] married
Levels: married single
> x[c(2, 4)]     # access 2nd and 4th element
[1] married single
Levels: married single
> x[-1]          # access all but 1st element
[1] married married single
Levels: married single
> x[c(TRUE, FALSE, FALSE, TRUE)]  # using logical vector
[1] single single
Levels: married single
```

Arrays and Matrices

Matrix is a two dimensional data structure in R programming. Matrix is similar to vector.

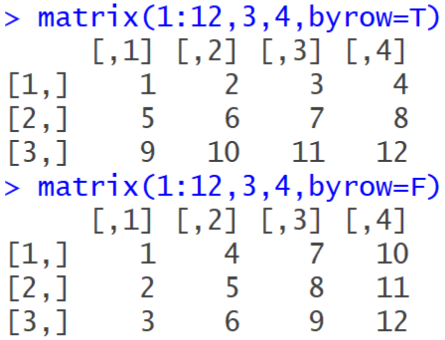
matrix(data, nrow, ncol, byrow, dimnames)

nrow is the number of rows to be created.

ncol is the number of columns to be created.

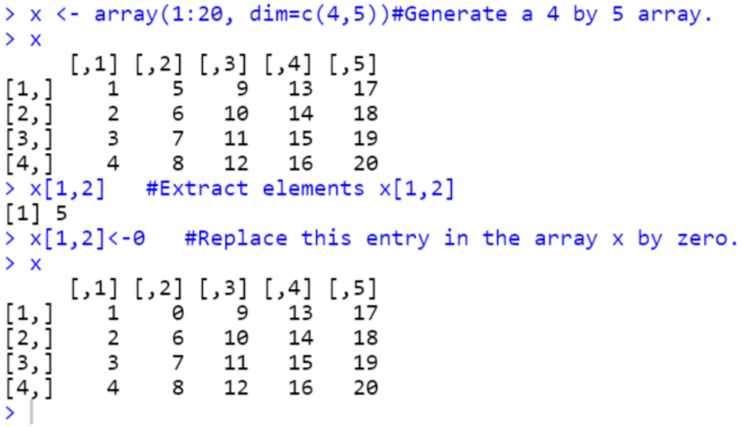
byrow is a logical clue. If TRUE then the input vector elements are arranged by row.

dimnames is the names assigned to the rows and columns.



```
> matrix(1:12,3,4,byrow=T)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> matrix(1:12,3,4,byrow=F)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Arrays in R are data objects which can be used to store data in more than two dimensions.
array(data = *data*, dim = *the dim attribute for the array to be created*)



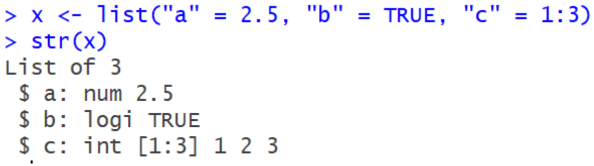
```
> x <- array(1:20, dim=c(4,5))#Generate a 4 by 5 array.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> x[1,2]   #Extract elements x[1,2]
[1] 5
> x[1,2]<-0   #Replace this entry in the array x by zero.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> 
```

How to access Elements of a matrix or array?
Using integer or logical vector as index.

List

List is a data structure having components of mixed data types.
A vector having all elements of the same type is called atomic vector but a vector having elements of different type is called list.

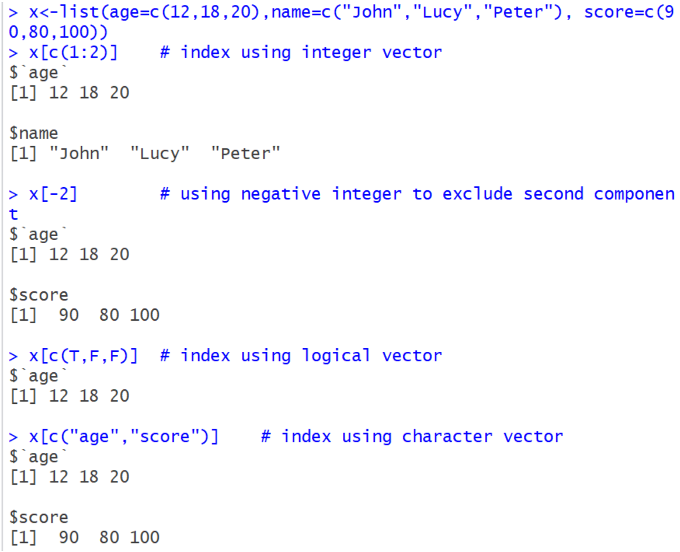
- List can be created using the list() function.
- Its structure can be examined with the str() function.
- In this example, a, b and c are called tags which makes it easier to reference the components of the list. However, tags are optional. We can create the same list without the tags.



```
> x <- list("a" = 2.5, "b" = TRUE, "c" = 1:3)
> str(x)
List of 3
 $ a: num 2.5
 $ b: logi TRUE
 $ c: int [1:3] 1 2 3
```

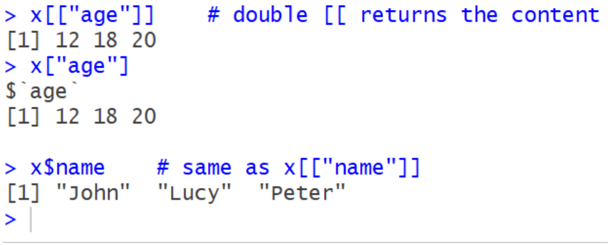
How to access components of a list?

Lists can be accessed in similar fashion to vectors. Integer, logical or character vectors can be used for indexing.



```
> x<-list(age=c(12,18,20),name=c("John","Lucy","Peter"), score=c(9
0,80,100))
> x[c(1:2)]    # index using integer vector
$`age`
[1] 12 18 20

$name
[1] "John"  "Lucy"  "Peter"

> x[-2]        # using negative integer to exclude second componen
t
$`age`
[1] 12 18 20

$score
[1]  90  80 100

> x[c(T,F,F)]  # index using logical vector
$`age`
[1] 12 18 20

> x[c("age","score")]    # index using character vector
$`age`
[1] 12 18 20

$score
[1]  90  80 100
```

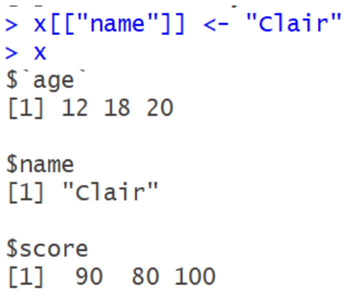
Indexing with the square bracket [as shown above will give us sublist not the content inside the component. To retrieve the content, we need to use [[. An alternative to [[, which is used often while accessing content of a list is the $ operator.



```
> x[["age"]]    # double [[ returns the content
[1] 12 18 20
> x["age"]
$`age`
[1] 12 18 20

> x$name    # same as x[["name"]]
[1] "John"  "Lucy"  "Peter"
> 
```

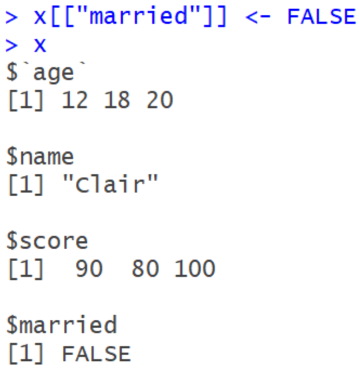
Change, add, delete components of lists

- How to modify a list in R?
  We can change components of a list through reassignment. We can choose any of the component accessing techniques discussed above to modify it.



```
> x[["name"]] <- "Clair"
> x
$`age`
[1] 12 18 20

$name
[1] "Clair"

$score
[1]  90  80 100
```

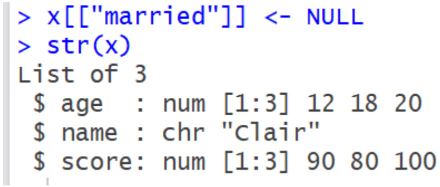
- How to add components to a list?



```
> x[["married"]] <- FALSE
> x
$`age`
[1] 12 18 20

$name
[1] "Clair"

$score
[1]  90  80 100

$married
[1] FALSE
```

- How to delete components to a list?

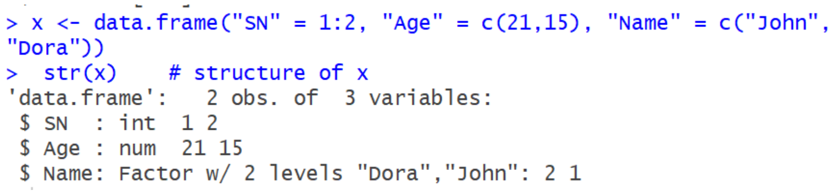
We can delete a component by assigning NULL to it.



```
> x[["married"]] <- NULL
> str(x)
List of 3
 $ age  : num [1:3] 12 18 20
 $ name : chr "Clair"
 $ score: num [1:3] 90 80 100
```

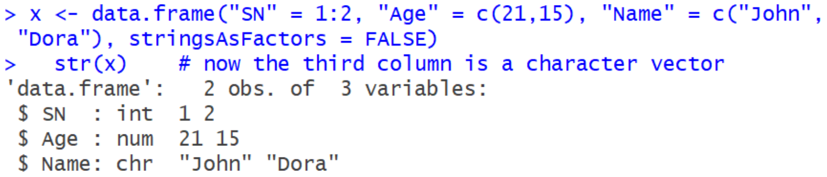
Data frame

- Data frame is a two dimensional data structure in R. It is a special case of a list which has each component of equal length.
- Notice that the third column, Name is of type factor, instead of a character vector.
  By default, data.frame() function converts character vector into factor.



```
> x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" = c("John",
"Dora"))
>  str(x)    # structure of x
'data.frame':   2 obs. of  3 variables:
 $ SN  : int  1 2
 $ Age : num  21 15
 $ Name: Factor w/ 2 levels "Dora","John": 2 1
```

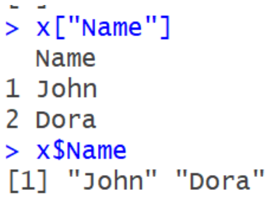
To suppress this behavior, we can pass the argument stringsAsFactors=FALSE.



```
> x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" = c("John",
 "Dora"), stringsAsFactors = FALSE)
>   str(x)    # now the third column is a character vector
'data.frame':   2 obs. of  3 variables:
 $ SN  : int  1 2
 $ Age : num  21 15
 $ Name: chr  "John" "Dora"
```

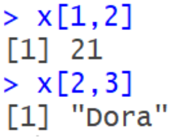
How to access Components of a Data Frame?

Components of data frame can be accessed like a list or like a matrix.
1.We can use either [, [[or $ operator to access columns of data frame.



```
> x["Name"]
  Name
1 John
2 Dora
> x$Name
[1] "John" "Dora"
```

2.Data frames can be accessed like a matrix by providing index for row and column.



```
> x[1,2]
[1] 21
> x[2,3]
[1] "Dora"
```

Programming: Flow Control

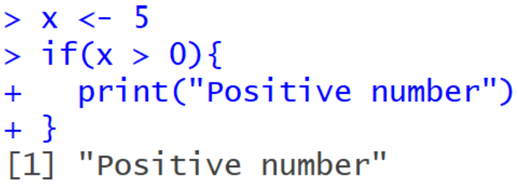
In computer programming, control flow or flow of control is the order function calls, instructions, and statements are executed or evaluated when a program is running.

- Conditional statement:
    - if statement
    - if-else statement.
- Loops: Loops are used in programming to repeat a specific block of code.
    - for loop
    - repeat loop
    - while loop
- *break* and *next* statement

R if statement

```
if (test expression) {
statement
}
```

If the **test expression** is TRUE, the **statement** gets executed. But if it's FALSE, nothing happens.
Here, **test expression** can be a logical or numeric value.
In the case of numeric value, zero is taken as FALSE, rest as TRUE.



```
> x <- 5
> if(x > 0){
+   print("Positive number")
+ }
[1] "Positive number"
```

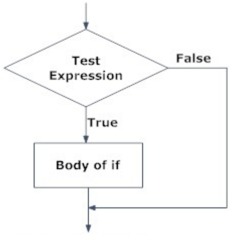
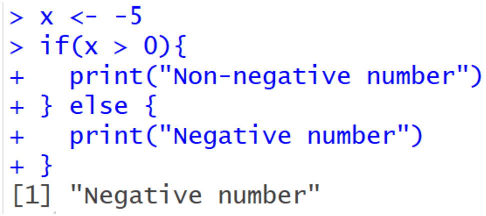


Fig: Operation of if statement

if else statement

```
if (test expression) {
statement1
} else{
statement2
}
```

It is important to note that else must be in the same line as the closing braces of the if statement.



```
> x <- -5
> if(x > 0){
+   print("Non-negative number")
+ } else {
+   print("Negative number")
+ }
[1] "Negative number"
```

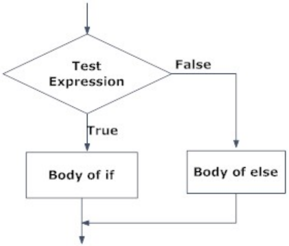
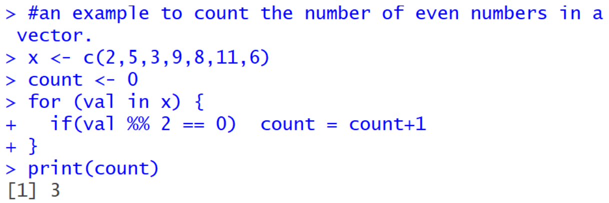


Fig: Operation of if...else statement

For loop

```
for (val in sequence)
{
statement
}
```

Here, **sequence** is a vector and **val** takes on each of its value during the loop. In each iteration, statement is evaluated.

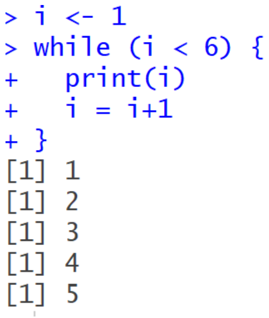


```
> #an example to count the number of even numbers in a
 vector.
> x <- c(2,5,3,9,8,11,6)
> count <- 0
> for (val in x) {
+   if(val %% 2 == 0)  count = count+1
+ }
> print(count)
[1] 3
```

while loop

while (test-expression)
{ statement
}

Here, test-expression is evaluated and the body of the loop is entered if the result is TRUE.
The statements inside the loop are executed and the flow returns to evaluate the test-expression again.
This is repeated each time until test- expression evaluates to FALSE, in which case, the loop exits.



```
> i <- 1
> while (i < 6) {
+   print(i)
+   i = i+1
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

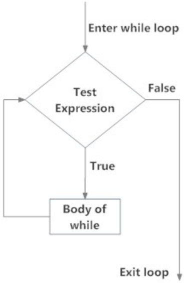
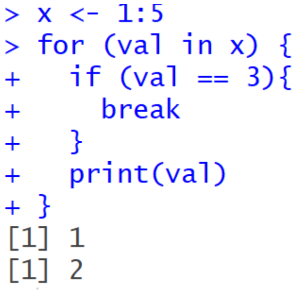


Fig: operation of while loop

break and next Statement

A break statement is used inside a loop (repeat, for, while) to stop the iterations and flow the control outside of the loop.

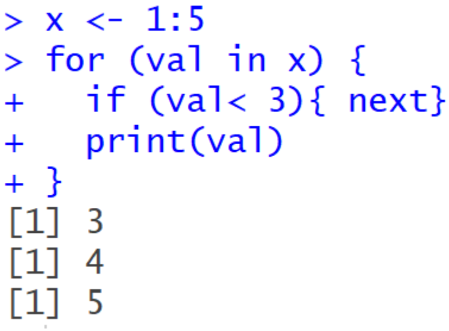
```
if (test-expression) { break }
```



```
> x <- 1:5
> for (val in x) {
+   if (val == 3){
+     break
+   }
+   print(val)
+ }
[1] 1
[1] 2
```

A **next** statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

```
if (test-condition) { next }
```

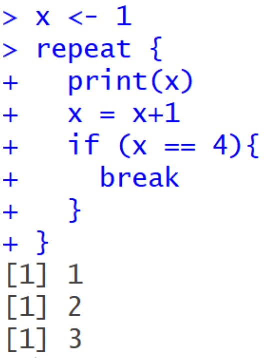


```
> x <- 1:5
> for (val in x) {
+   if (val< 3){ next}
+   print(val)
+ }
[1] 3
[1] 4
[1] 5
```

repeat loop

A **repeat loop** is used to iterate over a block of code multiple number of times. There is **no condition check** in repeat loop to exit the loop.We must ourselves put a condition explicitly inside the body of the loop and use the **break** statement to exit the loop.

repeat{ statement }



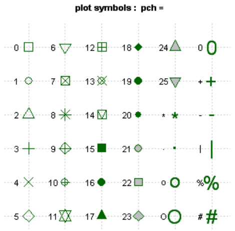
```
> x <- 1
> repeat {
+   print(x)
+   x = x+1
+   if (x == 4){
+     break
+   }
+ }
[1] 1
[1] 2
[1] 3
```

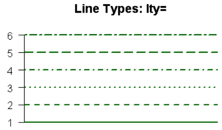
Graph

- Scatterplots:plot()
- Line Charts:plot(),lines()
- Bar Chart:barplot()
- Histograms: hist()
- Pie Charts:pie()

Graphical Parameters

You can customize many features of your graphs (fonts, colors, axes, titles) through graphic options.
Use the pch= option to specify symbols to use when plotting points.





You can change line types using lty= option

Axes

Many high level plotting functions (plot, hist, boxplot, etc.) allow you to include axis options

```
# Specify axis options within plot()
plot(x, y, main="title",sub="subtitle"
xlab="X-axis label", ylab="y-axis label",
xlim=c(xmin, xmax), ylim=c(ymin, ymax))
# Use the title( ) function to add labels to a plot.
title(main="main title", sub="sub-title",
xlab="x-axis label", ylab="y-axis label")
```