

STATISTICS WITH R PROGRAMMING

UNIT-1

Introduction

How to run R

R Sessions and Functions

Basic Math

Variables

Data Types

Vectors

Data Frames

Lists

Matrices

Arrays

Classes

INTRODUCTION

A. What Is R?

- R is a scripting language for statistical data manipulation and analysis.
- It was inspired by, and is mostly compatible with, the statistical language S developed by AT&T.
- The name S, obviously standing for statistics, was an allusion to another programming language developed at AT&T with a one-letter name, C.
- S later was sold to a small firm, which added a GUI interface and named the result S-Plus.
- R has become more popular than S/S-Plus, both because it's free and because more people are contributing to it.
- R is sometimes called "GNU S."

B. Why Use R for Your Statistical Work?

R is a scripting language is inexpensive and beautiful.

R - a scripting language which is

- a public-domain implementation of the widely-regarded S statistical language.
- comparable, and often superior, in power to commercial products in most senses
- available for Windows, Macs, Linux
- in addition to enabling statistical operations, it's a general programming language, so that you can automate your analyses and create new functions
- object-oriented and functional programming structure your data sets are saved between sessions, so you don't have to reload each time
- open-software nature means it's easy to get help from the user community, and lots of new functions get contributed by users, many of which are prominent statisticians

C. The advantages of R Language are:

- Clearer, more compact code.
- Potentially much faster execution speed.
- Less debugging (since you write less code).
- Easier transition to parallel programming.

How to Run R

R has two modes, interactive and batch. The former is the typical one used.

A. Running R in Interactive Mode

- You start R by typing "R" on the command line in Linux or on a Mac, or in a Windows Run window. You'll get a greeting, and then the R prompt, >.
- We can then execute R commands and execute own R code like R files with .r extension.
- The command `> source("z.r")` which would execute the contents of that file.

B. Running R in Batch Mode

It is possible to invoke R from the Windows command-line by entering the full path name of the executable, such as

```
C:\Program Files\R\R-3.3.0\bin\R --vanilla
```

The option `--vanilla` is an alias for several options, which in short summary tell R to not load any files at startup and to not ask the user whether to save the workspace image upon exit. If you were to invoke R from within the `bin/` directory, you could enter the much simpler command

R `--vanilla`

And in the spirit of keeping things simple, if you place the `bin/` directory in your `PATH` variable, then no matter the location of your current directory, you can always use the simpler command to invoke R. Here's how to set that `bin/` directory in your `PATH` variable:

1. Press the Windows Key.
2. Type `systempropertiesadvanced` – all one word – and press enter — OR — type `sysdm.cpl`, press enter, and click the “Advanced” tab.
3. Click “Environment Variables”.
4. Select “Path” and click “Edit”.
5. Place the cursor at the very end of the “Variable value” field.
6. Type the appropriate path name to the `bin/` directory with a preceding `;` (path names are `;` delimited); here's an example of what I typed:

```
;C:\Program Files\R\R-3.3.0\bin\
```

7. Click all the “OK” buttons until you have exited.
8. The BATCH tool
9. The above invocation of R will launch R in the command-line window – just as though you were using the command-line in RStudio or R GUI. However, from within the command line there are several CMD “tools” which are available to the user which are not meant to be called directly (from a GUI).
10. One such tool, BATCH, allows the user to run R files at the command-line (similar to using `source()` in the interactive GUI). The command

```
R --vanilla CMD BATCH file.R file.Rout
```

will execute `file.R` and save the output to `file.Rout` — assuming you are within (your working directory is) `file.R`'s directory.

R Sessions and Functions

1.3.A. Sessions

The R programming can be done in two ways. We can either type the command lines on the screen inside an “R-session”, or we can save the commands as a “script” file and execute the whole file inside R.

```
[Previously saved workspace restored]
```

```
>
```

Session in R is a workspace between start and end point in R console.

Session starts when R console is initiated. It end when we quit from the R console.

Working with R session

Once we are inside the R session, we can directly execute R language commands by typing them line by line. Pressing the enter key terminates typing of command and brings the `>` prompt again. In the example session below, we declare 2 variables 'a' and 'b' to have values 5 and 6 respectively, and assign their sum to another variable called 'c':

```
> a = 5
```

```
> b = 6
> c = a + b
> c
```

The value of the variable 'c' is printed as,

```
[1] 11
```

In R session, typing a variable name prints its value on the screen.

Get help inside R session

To get help on any function of R, type **help(function-name)** in R prompt. For example, if we need help on "if" logic, type,

```
> help("if")
```

then, help lines for the "if" statement are printed.

Exit the R session

To exit the R session, type **quit()** in the R prompt, and say 'n' (no) for saving the workspace image. This means, we do not want to save the memory of all the commands we typed in the current session:

```
> quit()
```

Save workspace image? [y/n/c]: n

Saving the R session

Note that by not saving the current session, we lose all the memory of current session commands and the variables and objects created when we exit R prompt.

When we work in R, the R objects we created and loaded are stored in a memory portion called workspace. When we say 'no' to saving the workspace, we lose all these objects are wiped out from the workspace memory. If we say 'yes', they are saved into a file called ".RData" is written to the present working directory.

When we start R in the same current directory next time, the workspace and all the created objects are restored automatically from this ".RData" directory.

Listing the objects in the current R session

We can list the names of the objects in the current R session by **ls()** command. For example, start R session fresh and proceed as follows:

```
> a = 5
> b = 6
> c = 8
> sum = a+b+c
> sum
[1] 19
> ls()
[1] "a" "b" "c" "sum"
```

Here, the objects we created have been listed.

Removing objects from the current R session

Specific objects created in the current session can be removed using **rm()** command. If we specify the name of an object, it will be removed. If we just say **rm(list = ls())**, all objects created so far will be removed.

```
> a = 5
> b = 6
> c = 8
> sum = a+b+c
> sum
[1] 19
> ls()
```

```
[1] "a" "b" "c" "sum"
> rm(list = ls())
> ls()
character(0)
```

Getting and setting the current working directories

From R prompt, we can get information about the current working directory using **getwd()** command:

```
> getwd()
[1] "/home/user"
```

Similarly, we can set the current working directory by calling **setwd()** function:

```
> setwd("/home/user/prog")
```

After this, "/home/user/prog" will be the working directory.

In Windows version of R, the working directory can be set from menu in R window.

Comments

Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program. Single comment is written using # in the beginning of the statement as follows:

```
# My first program in R Programming
```

R does not support multi-line comments but you can perform a trick which is something as follows:

```
if(FALSE)
{
  "This is a demo for multi-line comments and it should be put inside either a single or double quote"
}
```

B. 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.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows –

```
function_name<- function(arg_1, arg_2, ...) {
  Function body
}
```

Function Components

The different parts of a function are –

- **Function Name** – This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** – An argument is a placeholder. 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.
- **Function Body** – The function body contains a collection of statements that defines what the function does.
- **Return Value** – The return value of a function is the last expression in the function body to be evaluated.

Programming Example

In the following example, define a function `oddcount()` while in R's interactive mode. Then call the function on a couple of test cases. The function is supposed to count the number of odd numbers in its argument vector.

```
# comment: counts the number of odd integers in x
>oddcount<- function(x) {
+ k <- 0
+ for (n in x) {
+ if (n %% 2 == 1) k <- k+1
+ }
+ return(k)
+ }
```

```
>oddcount(c(1,3,5))
[1] 3
```

```
>oddcount(c(1,2,3,7,9))
[1] 4
```

Here is what happened: We first told R that we would define a function `oddcount()` of one argument `x`. The left brace demarcates the start of the body of the function. We wrote one R statement per line. Since we were still in the body of the function, R reminded us of that by using `+` as its prompt1 instead of the usual

`>`. After we finally entered a right braces to end the function body, R resumed the `>` prompt.

Note that arguments in R functions are read-only, in that a copy of the argument is made to a local variable, and changes to the latter don't affect the original variable. Thus changes to the original variable are typically made by reassigning the return value of the function.

Types of Functions

Functions are two type, they are

1. Built-in function
2. User defined functions

1. Built-in Function

Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc. They are directly called by user written programs. You can refer [most widely used R functions](#).

```
# Create a sequence of numbers from 32 to 44.
print(seq(32,44))

# Find mean of numbers from 25 to 82.
print(mean(25:82))

# Find sum of numbers from 41 to 68.
print(sum(41:68))
```

When we execute the above code, it produces the following result -

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

2. User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
new.function<- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}
```

Calling a Function

```
# Create a function to print squares of numbers in sequence.
new.function<- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}

# Call the function new.function supplying 6 as an argument.
new.function(6)
```

When we execute the above code, it produces the following result -

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

Calling a Function without an Argument

```
# Create a function without an argument.
new.function<- function() {
  for(i in 1:5) {
    print(i^2)
  }
}

# Call the function without supplying an argument.
new.function()
```

When we execute the above code, it produces the following result –

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with arguments.
new.function<- function(a,b,c) {
  result<- a * b + c
  print(result)
}

# Call the function by position of arguments.
new.function(5,3,11)

# Call the function by names of the arguments.
new.function(a = 11, b = 5, c = 3)
```

When we execute the above code, it produces the following result –

```
[1] 26
[1] 58
```

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments.
new.function<- function(a = 3, b = 6) {
  result<- a * b
  print(result)
}

# Call the function without giving any argument.
new.function()
```



```
# Call the function with giving new values of the argument.
new.function(9,5)
```

When we execute the above code, it produces the following result –

```
[1] 18
[1] 45
```

Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# Create a function with arguments.
new.function<- function(a, b) {
  print(a^2)
  print(a)
  print(b)
}
# Evaluate the function without supplying one of the arguments.
new.function(6)
```

When we execute the above code, it produces the following result –

```
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
```

Basic Math in R

1.4.1. R Arithmetic Operators

These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

Arithmetic Operators in R

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulo Division
%/%	Integer Division

An example run

```
> x <-5
> y <-16
```

```
>x+y
[1]21
```

```
>x-y
[1]-11
```

```
>x*y
[1]80
```

```
>y/x
[1]3.2
```

```
>y/%x
[1]3
```

```
>y%%x
[1]1
```

```
>y^x
[1]1048576
```

Examples in Arithmetic operators

```
# 1 plus 1
1+1
[1] 2
# 4 minus 3
4-3
[1] 1
# 14 divided by 10
14/10
[1] 1.4
# 10 multiplied by 5
10*5
[1] 50
# 3 squared
3^2
[1] 9
# 5 mod 2
5%%2
[1] 1
# 4 divided by 2 (integer division)
4%/%2
[1] 2
# log to the base e of 2
log(2)
[1] 0.6931472
# antilog of 2
exp(2)
[1] 7.389056
# log to base 2 of 3
```

```
log(3,2)
[1] 1.584963
# log to base 10 of 2
log10(2)
[1] 0.30103
# square root of 2
sqrt(2)
[1] 1.414214
# !5
factorial(4)
[1] 24
# largest interger smaller than 2
floor(2)
[1] 2
# smallest integer greater than 6
ceiling(6)
[1] 6
# round 3.14159 to three digits
round(3.14159, digits=3)
[1] 3.142
# create 10 random digits between zero and 1 (from a uniform distribution)
runif(10)
[1] 0.07613962 0.66543266 0.48379168 0.40593920 0.67715428 0.49170373
[7] 0.62351598 0.19275859 0.48018351 0.34890640
# cosine of 3
cos(3)
[1] -0.9899925
# sine of 3
sin(3)
[1] 0.14112
# tangent of 3
tan(3)
[1] -0.1425465
# absolute value of -3
abs(-3)
[1] 3
```

R Relational Operators

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

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

An example run

```
> x <-5
> y <-16

> x<y
[1] TRUE

> x>y
[1] FALSE

> x<=5
[1] TRUE

> y>=20
[1] FALSE

>y==16
[1] TRUE

>x !=5
[1] FALSE
```

Operation on Vectors

The above mentioned operators work on vectors. The variables used above were in fact single element vectors.

We can use the function `c()` (as in concatenate) to make vectors in R.

All operations are carried out in element-wise fashion. Here is an example.

```
> x <-c(2,8,3)
> y <-c(6,4,1)
```

```
>x+y
[1] 8 12 4

> x>y
[1] FALSE TRUE TRUE
```

When there is a mismatch in length (number of elements) of operand vectors, the elements in shorter one is recycled in a cyclic manner to match the length of the longer one.

R will issue a warning if the length of the longer vector is not an integral multiple of the shorter vector.

```
> x <-c(2,1,8,3)
> y <-c(9,4)

>x+y# Element of y is recycled to 9,4,9,4
[1] 11 5 17 7

>x-1# Scalar 1 is recycled to 1,1,1,1
[1] 1 0 7 2

>x+c(1,2,3)
[1] 3 3 11 4
Warning message:
In x +c(1,2,3):
longer object length is not a multiple of shorter object length
```

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

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

```

Variables

- A variable provides us with named storage that our programs can manipulate.
- A variable in R can store an atomic vector, group of atomic vectors or a combination of many Objects.
- A valid variable name consists of alphabets, numbers, the dot(.) or underline characters.
- The variable name starts with a letter or the dot
- The variable name should not followed by a number.

Examples

Variable Name	Validity	Reason
var_name2.	Valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	Invalid	Starts with a number
.var_name , var.name	Valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	Invalid	The starting dot is followed by a number making it invalid.
_var_name	Invalid	Starts with _ which is not valid

Variable Assignment

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using **print()** or **cat()**function. The **cat()** function combines multiple items into a continuous print output.

```

# Assignment using equal operator.
var.1=c(0,1,2,3)

```

```
# Assignment using leftward operator.
var.2<-c("learn","R")

# Assignment using rightward operator.
c(TRUE,1)->var.3

print(var.1)
cat("var.1 is ",var.1,"\n")
cat("var.2 is ",var.2,"\n")
cat("var.3 is ",var.3,"\n")
```

When we execute the above code, it produces the following result –

```
[1] 0 1 2 3
var.1 is 0 1 2 3
var.2 is learn R
var.3 is 1 1
```

Note – The vector `c(TRUE,1)` has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

Data Type of a Variable

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
> x<-"Hello"
>class(x)
[1] "character"

> y<-34.5
>class(y)
[1] "numeric"
```

Finding Variables – ls() function

To know all the variables currently available in the workspace we use the **ls()** function. Also the `ls()` function can use patterns to match the variable names.

```
print(ls())
```

When we execute the above code, it produces the following result –

```
[1] "x"  "y"
```

Deleting Variables – rm() function

Variables can be deleted by using the **rm()** function. Below we delete the variable “x”. On printing the value of the variable error is thrown.

```
>
> rm(x)
> x
Error: object 'x' not found
```

All the variables can be deleted by using the `rm()` and `ls()` function together.

```
rm(list = ls())
print(ls())
```

R Assignment Operators on Variables

These operators are used to assign values to variables.

Assignment Operators in R

Operator	Description
<code><-</code> , <code><<-</code> , <code>=</code>	Leftwards assignment
<code>-></code> , <code>->></code>	Rightwards assignment

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

The `<<-` operator is used for assigning to variables in the parent environments (more like global assignments). The rightward assignments, although available are rarely used.

```
> x <-5
>x
[1] 5
```

```
> x =9
>x
[1] 9
```

```
>10-> x
>x
[1] 10
```

Data Types

There are numerous data types in R that store various kinds of data. The four main types of data most likely to be used are `numeric`, `character` (string), `Date/POSIXct` (time-based) and `logical` (`TRUE/FALSE`).

The type of data contained in a variable is checked with the `class` function.

```
>class(x)
[1] "numeric"
```

Numeric Data

The most commonly used numeric data is `numeric`. This is similar to a `float` or `double` in other languages. It handles integers and decimals, both positive and negative, and, of course,

zero. A numeric value stored in a variable is automatically assumed to be `numeric`. Testing whether a variable is `numeric` is done with the function `is.numeric`.

```
>is.numeric(x)

[1] TRUE
```

Another important, if less frequently used, type is `integer`. As the name implies this is for whole numbers only, no decimals. To set an integer to a variable it is necessary to append the value with an `L`. As with checking for a `numeric`, the `is.integer` function is used.

```
>i<- 5L
>i

[1] 5

>is.integer(i)

[1] TRUE
```

Do note that, even though `i` is an `integer`, it will also pass a `numeric` check.

```
>is.numeric(i)

[1] TRUE
```

R nicely promotes integers to `numeric` when needed. This is obvious when multiplying an `integer` by a `numeric`, but importantly it works when dividing an `integer` by another `integer`, resulting in a decimal number.

```
>class(4L)

[1] "integer"

>class(2.8)

[1] "numeric"
```

Character Data

The character data type is a string type data which is very common in statistical analysis and must be handled with care. R has two primary ways of handling character data: `character` and `factor`.

Example:

```
> x <- "data"
>x
[1] "data"

> y <- factor("data")
> y
[1] data
Levels: data
```

Notice that `x` contains the word “data” encapsulated in quotes, while `y` has the word “data” without quotes and a second line of information about the `levels` of `y`.

Characters are case sensitive, so “Data” is different from “data” or “DATA.”

To find the length of a character (or numeric) use the `nchar` function.

```

>nchar(x)
[1] 4

>nchar("hello")
[1] 5

>nchar(3)
[1] 1

>nchar(452)
[1] 3

```

This will not work for factor data.

Example:

```

>nchar(y)
Error: 'nchar()' requires a character vector

```

Logical

logical are a way of representing data that can be either TRUE or FALSE. Numerically, TRUE is the same as 1 and FALSE is the same as 0.

So TRUE * 5 equals 5
while FALSE * 5 equals 0.

```

> TRUE * 5
[1] 5

> FALSE * 5
[1] 0

```

Similar to other types, logicals have their own test, using the `is.logical` function.

```

> k <- TRUE
>class(k)
[1] "logical"

>is.logical(k)
[1] TRUE

```

R provides T and F as shortcuts for TRUE and FALSE, respectively, but it is best practice not to use them, as they are simply variables storing the values TRUE and FALSE and can be overwritten, which can cause a great deal of frustration as seen in the following example.

```

>TRUE
[1] TRUE

> T
[1] TRUE

>class(T)
[1] "logical"

> T <- 7
> T
[1] 7

>class(T)
[1] "numeric"

```

Logical can result from comparing two numbers, or characters.

Example:

```

> # does 2 equal 3?
> 2 == 3

```

```
[1] FALSE

> # does 2 not equal three?
> 2 != 3

[1] TRUE

> # is two less than three?
> 2 < 3

[1] TRUE

> # is two less than or equal to three?
> 2 <= 3

[1] TRUE
```

Vectors

- A `vector` is a collection of elements, all of the same type. For instance, `c(1, 3, 2, 1, 5)` is a `vector` consisting of the numbers 1, 3, 2, 1, 5, in that order.
- Similarly, `c("R", "Excel", "SAS", "Excel")` is a `vector` of the character elements “R,” “Excel,” “SAS” and “Excel.”
- A `vector` cannot be of mixed type.
- `Vectors` play a crucial, and helpful, role in `R`.
- `Vectors` do not have a dimension, meaning there is no such thing as a column `vector` or row `vector`.
- These `vectors` are not like the mathematical `vector` where there is a difference between row and column orientation.
- **The most common way to create a `vector` is with `c`. The “c” stands for combine because multiple elements are being combined into a `vector`.**

Example:

```
> x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> x
[1] 1 2 3 4 5 6 7 8 9 10
```

Vector Operations

Now that we have a `vector` of the first ten numbers, we might want to multiply each element by 3. In `R` this is a simple operation using just the multiplication operator (`.`).

Example:

```
> x * 3
[1] 3 6 9 12 15 18 21 24 27 30
```

No loops are necessary. Addition, subtraction and division are just as easy. This also works for any number of operations.

Example:

```
> x + 2
[1] 3 4 5 6 7 8 9 10 11 12

> x - 3
[1] -2 -1 0 1 2 3 4 5 6 7

> x/4
[1] 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50
```

```
> x^2
[1] 1 4 9 16 25 36 49 64 81 100

> sqrt(x)
[1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000 3.162
```

Earlier we created a `vector` of the first ten numbers using the `c` function, which creates a `vector`. A shortcut is the `:` operator, which generates a sequence of consecutive numbers, in either direction.

Example:

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

> 10:1
[1] 10 9 8 7 6 5 4 3 2 1

> -2:3
[1] -2 -1 0 1 2 3

> 5:-7
[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7
```

Vector operations can be extended even further. Let's say we have two vectors of equal length. Each of the corresponding elements can be operated on together.

Example:

```
> # create two vectors of equal length
> x <- 1:10
> y <- -5:4
> # add them
> x + y

[1] -4 -2 0 2 4 6 8 10 12 14

> # subtract them
> x - y
[1] 6 6 6 6 6 6 6 6 6 6

> # multiply them
> x * y
[1] -5 -8 -9 -8 -5 0 7 16 27 40

> # check the length of each
> length(x)
[1] 10

> length(y)
[1] 10

> # the length of them added together should be the same
> length(x + y)
[1] 10
```

- Things get a little more complicated when operating on two `vectors` of unequal length. The shorter `vector` gets recycled, that is, its elements are repeated, in order, until they

have been matched up with every element of the longer `vector`.

- If the longer one is not a multiple of the shorter one, a warning is given.

Example:

```
> x + c(1, 2)
[1] 2 4 4 6 6 8 8 10 10 12

> x + c(1, 2, 3)
Warning: longer object length is not a multiple of shorter object
length
[1] 2 4 6 5 7 9 8 10 12 11
```

Comparisons also work on `vectors`. Here the result is a vector of the same length containing `TRUE` or `FALSE` for each element.

Example:

```
> x <= 5
[1] TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE

> x > y
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

> x < y
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

To test whether all the resulting elements are `TRUE`, use the `all` function. Similarly, the `any` function checks whether any element is `TRUE`.

```
> x <- 10:1
> y <- -4:5
> any(x < y)
[1] TRUE

> all(x < y)
[1] FALSE
```

The `nchar` function also acts on each element of a `vector`.

Example:

```
> q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
+       "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
> nchar(q)
[1] 6 8 8 7 5 8 10 6 7 6

> nchar(y)
[1] 2 2 2 2 1 1 1 1 1 1
```

Accessing individual elements of a `vector` is done using square brackets (`[]`). The first element of `x` is retrieved by typing `x[1]`, the first two elements by `x[1:2]` and nonconsecutive elements by `x[c(1, 4)]`.

```
> x[1]
[1] 10

> x[1:2]
[1] 10 9

> x[c(1, 4)]
[1] 10 7
```

Factor Vectors

Factors are used to create a vector to the integer data type.

Let's create a simple vector of text data that has a few repeats. We will start with the vector `q` we created earlier and add some elements to it.

Example:

```
> x <- c("Hockey", "Cricket", "Volleyball", "Tennis")
> x
[1] "Hockey"      "Cricket"      "Volleyball"   "Tennis"
```

Converting this to a factor is easy with `as.factor`.

```
> y<-as.factor(x)
> y
[1] Hockey      Cricket      Volleyball   Tennis
Levels: Cricket Hockey Tennis Volleyball
```

Missing Data

Missing data plays a critical role in both statistics and computing, and R has two types of missing data, `NA` and `NULL`. While they are similar, they behave differently and that difference needs attention.

NA

Often we will have data that has missing values for any number of reasons. Statistical programs use varying techniques to represent missing data such as a dash, a period or even the number 99. R uses `NA`. `NA` will often be seen as just another element of a vector. `is.na` tests each element of a vector for missing.

Example:

```
> z <- c(1, 2, NA, 8, 3, NA, 3)
> z
[1] 1 2 NA 8 3 NA 3
>is.na(z)
[1] FALSE FALSE TRUE FALSE FALSE TRUE FALSE
```

`NA` is entered simply by typing the letters “N” and “A” as if they were normal text. This works for any kind of vector.

NULL

`NULL` is the absence of anything. It is not exactly missingness, it is nothingness. Functions can sometimes return `NULL` and their arguments can be `NULL`. An important difference between `NA` and `NULL` is that `NULL` is atomical and cannot exist within a vector. If used inside a vector it simply disappears.

```
> z <- c(1, NULL, 3)
> z
[1] 1 3
```

Even though it was entered into the vector `z`, it did not get stored in `z`. In fact, `z` is only two elements long.

The test for a `NULL` value is `is.null`.

```
> d <- NULL
> is.null(d)
[1] TRUE

> is.null(7)
[1] FALSE
```

Since `NULL` cannot be a part of a vector, `is.null` is appropriately not vectorized.

Advanced Data Structures

data.frames

The `data.frame` is just like an Excel spreadsheet in that it has columns and rows. In statistical terms, each column is a variable and each row is an observation.

In terms of how R organizes `data.frames`, each column is actually a vector, each of which has the same length. That is very important because it lets each column hold a different type of data. This also implies that within a column each element must be of the same type, just like with vectors.

There are numerous ways to construct a `data.frame`, the simplest being to use the `data.frame` function. Let's create a basic `data.frame` using some of the vectors we have already introduced, namely `x`, `y` and `q`.

Example:

```
> x <- 10:1
> y <- -4:5
> q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
+       "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
> df <- data.frame(x, y, q)
> df
```

	x	y	q
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

Assigned names during the creation process

Example:

```
>df<- data.frame(First = x, Second = y, Sport = q)
>df
```

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

Functions on Data Frames

```
>nrow(df)
[1] 10
```

```
>ncol(df)
[1] 3
```

```
>dim(df)
[1] 10 3
```

```
>names(df)
[1] "First" "Second" "Sport"
```

```
>names(df)[3]
```

```
[1] "Sport"
```

Row names of a data.frame.

Example:

```
>rownames(df)
```

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

```
>rownames(df) <- c("One", "Two", "Three", "Four", "Five", "Six",
+                  "Seven", "Eight", "Nine", "Ten")
```

```
>rownames(df)
```

```
[1] "One" "Two" "Three" "Four" "Five" "Six" "Seven" "Eight"
[9] "Nine" "Ten"
```

```
># set them back to the generic index
```

```
>rownames(df) <- NULL
```

```
>rownames(df)
```

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

Usually a `data.frame` has far too many rows to print them all to the screen, so thankfully the `head` function prints out only the first few rows.

Example:

```
>head(df)
```

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse

```
>head(df, n = 7)
```

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball

```
>tail(df)
```

	First	Second	Sport
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

As we can with other variables, we can check the `class` of a `data.frame` using the `class` function.

```
>class(df)
[1] "data.frame"
```

Since each column of the `data.frame` is an individual vector, it can be accessed individually and each has its own `class`. Like many other aspects of R, there are multiple ways to access an individual column. There is the `$` operator and also the square brackets. Running `theDF$Sport` will give the third column in `theDF`. That allows us to specify one particular column by name.

Example:

```
>df$Sport
```

```
[1] Hockey      Football    Baseball    Curling     Rugby       Lacrosse
[7] Basketball  Tennis      Cricket     Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis
```

Similar to vectors, `data.frames` allow us to access individual elements by their position using square brackets, but instead of having one position two are specified. The first is the row number and the second is the column number. So to get the third row from the second column we use `theDF[3, 2]`.

```
>theDF[3, 2]
```

```
[1] -2
```

To specify more than one row or column use a vector of indices.

Example:

```
># row 3, columns 2 through 3
>df[3, 2:3]

      Second      Sport
3         -2 Baseball

>
># rows 3 and 5, column 2
>df[c(3, 5), 2]

[1] -2 0

>
># rows 3 and 5, columns 2 through 3
>df[c(3, 5), 2:3]

      Second      Sport
3         -2 Baseball
5          0    Rugby
```

To access an entire row, specify that row while not specifying any column. Likewise, **to access an entire column**, specify that column while not specifying any row.

Example:

```
># all of column 3
># since it is only one column a vector is returned
>df[, 3]

[1] Hockey      Football    Baseball    Curling     Rugby       Lacrosse
[7] Basketball Tennis      Cricket     Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis
>
># all of columns 2 through 3
>df[, 2:3]

      Second      Sport
1         -4    Hockey
2         -3  Football
3         -2  Baseball
4         -1    Curling
5          0    Rugby
6          1  Lacrosse
7          2 Basketball
8          3    Tennis
9          4    Cricket
10         5    Soccer

>
># all of row 2
>df[2, ]

      First Second      Sport
2          9         -3 Football

>
># all of rows 2 through 4
>df[2:4, ]

      First Second      Sport
2          9         -3 Football
3          8         -2 Baseball
4          7         -1  Curling
```

Lists

`list` in R stores any number of items of any type. A `list` can contain all numerics or characters or a mix of the two or `data.frames` or, recursively, other `lists`.

`Lists` are created with the `list` function where each argument to the function becomes an element of the `list`.

Example:

```
># creates a three element list
>list(1, 2, 3)

[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

>
># creates a single element list where the only element is a vector
># that has three elements
>list(c(1, 2, 3))

[[1]]
[1] 1 2 3

>
># creates a two element list
># the first element is a three element vector
># the second element is a five element vector
>list3 <- list(c(1, 2, 3), 3:7)

[[1]]
[1] 1 2 3

[[2]]
[1] 3 4 5 6 7

>
># two element list
># first element is a data.frame
># second element is a 10 element vector
>list4 <- list(df, 1:10)

>list4
[[1]]
  First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0     Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer

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

Like `data.frames`, `lists` can have names. Each element has a unique name that can be either viewed or assigned using `names`.

Example:

```
> names(list4)
NULL

> names(list4) <- c("data.frame", "vector")
> names(list4)
[1] "data.frame" "vector"

>names(list4)
NULL

>names(list4) <- c("data.frame", "vector",)
>names(list4)
[1] "data.frame" "vector"      "list"

> list4
$`data.frame`
      x y      q
1  10 -4    Hockey
2   9 -3   Football
3   8 -2   Baseball
4   7 -1    Curling
5   6  0     Rugby
6   5  1   Lacrosse
7   4  2 Basketball
8   3  3     Tennis
9   2  4    Cricket
10  1  5     Soccer

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

Creating an empty list of a certain size is, perhaps confusingly, done with `vector`.

Example:

```
> (emptyList<- vector(mode = "list", length = 4))
[[1]]
NULL
[[2]]
NULL
[[3]]
NULL
[[4]]
NULL
```

To access an individual element of a `list`,

use double square brackets, specifying either the element number or name. Note that this allows access to only one element at a time.

Example:

```
> list4[1]
$`data.frame`
      x  y      q
1  10 -4    Hockey
2   9 -3   Football
3   8 -2   Baseball
4   7 -1    Curling
5   6  0     Rugby
6   5  1   Lacrosse
7   4  2 Basketball
8   3  3     Tennis
9   2  4    Cricket
10  1  5     Soccer

> list4[2]

$`vector`

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

Once an element is accessed it can be treated as if that actual element is being used, allowing nested indexing of elements.

Example:

```
> list4[[1]]$q
[1] Hockey   Football  Baseball  Curling   Rugby    Lacrosse
Basketball Tennis    Cricket   Soccer

Levels: Baseball Basketball Cricket Curling Football Hockey Lacrosse Rugby
Soccer Tennis

> list4[[1]][,3,drop=FALSE]
      q
1    Hockey
2   Football
3   Baseball
4    Curling
5     Rugby
6   Lacrosse
7 Basketball
8     Tennis
9    Cricket
10    Soccer
>
```

It is possible to append elements to a `list` simply by using an index (either numeric or named) that does not exist.

Example:

```

># see how long it currently is
> length(list4)

[1] 2

>
># add a third element, unnamed
> list4[[3]] <- 2
> list4
$`data.frame`
  x y      q
1 10 -4   Hockey
2  9 -3  Football
3  8 -2  Baseball
4  7 -1   Curling
5  6  0    Rugby
6  5  1  Lacrosse
7  4  2 Basketball
8  3  3    Tennis
9  2  4   Cricket
10 1  5    Soccer

$vector
[1] 1 2 3 4 5 6 7 8 9 10

[[3]]
[1] 2

```

Here number 2 is add to the list as a third element

Matrices

`matrix` is similar to a `data.frame` in that it is rectangular with rows and columns with numerical values and with same type. They also act similarly to `vectors` with element-by-element addition, multiplication, subtraction, division and equality. The `nrow`, `ncol` and `dim` functions work just like they do for `data.frames`.

Example:

```

># create a 5x2 matrix
> A <- matrix(1:10, nrow = 5)

># create another 5x2 matrix
> B <- matrix(21:30, nrow = 5)

># create another 5x2 matrix
> C <- matrix(21:40, nrow = 2)
> A

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

```

```
> B
```

```
      [,1] [,2]
[1,]    21    26
[2,]    22    27
[3,]    23    28
[4,]    24    29
[5,]    25    30
```

```
> C
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    21    23    25    27    29    31    33    35    37    39
[2,]    22    24    26    28    30    32    34    36    38    40
```

```
>nrow(A)
```

```
[1] 5
```

```
>ncol(A)
```

```
[1] 2
```

```
>dim(A)
```

```
[1] 5 2
```

```
># addition
```

```
> A + B
```

```
      [,1] [,2]
[1,]    22    32
[2,]    24    34
[3,]    26    36
[4,]    28    38
[5,]    30    40
```

```
># multiplication
```

```
> A * B
```

```
      [,1] [,2]
[1,]    21   156
[2,]    44   189
[3,]    69   224
[4,]    96   261
[5,]   125   300
```

```
># see if the elements are equal
> A == B
```

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

Matrix multiplication is a commonly used operation in mathematics, requiring the number of columns of the left-hand `matrix` to be the same as the number of rows of the right-hand `matrix`. Both `A` and `B` are 5×2 so we will transpose `B` so it can be used on the right-hand side.

Example:

```
> A %*% t(B)
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,] 177 184 191 198 205
[2,] 224 233 242 251 260
[3,] 271 282 293 304 315
[4,] 318 331 344 357 370
[5,] 365 380 395 410 425
```

Another similarity with `data.frames` is that `matrices` can also have row and column names.

Example:

```
> colnames(A)
NULL
```

```
> rownames(A)
NULL
```

```
> colnames(A) <- c("Left", "Right")
> rownames(A) <- c("1st", "2nd", "3rd", "4th", "5th")
>
```

```
> colnames(B)
NULL
```

```
> rownames(B)
NULL
```

```
> colnames(B) <- c("First", "Second")
> rownames(B) <- c("One", "Two", "Three", "Four", "Five")
>
```

```
> colnames(C)
NULL
```

```
> rownames(C)
NULL
```



```
>colnames(C) <- LETTERS[1:10]
>rownames(C) <- c("Top", "Bottom")
```

There are two special `vectors`, `letters` and `LETTERS`, that contain the lower-case and upper-case letters, respectively.

Arrays

An `array` is a multidimensional `vector`. It must all be of the same type and individual elements are accessed in a similar fashion using square brackets. The first element is the row index, the second is the column index and the remaining elements are for outer dimensions.

Example:

```
>theArray<- array(1:12, dim = c(2, 3, 2))
```

```
>theArray
```

```
., 1
```

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

```
., 2
```

```
      [,1] [,2] [,3]
[1,]   7   9  11
[2,]   8  10  12
```

```
>theArray[1, , ]
```

```
      [,1] [,2]
[1,]   1   7
[2,]   3   9
[3,]   5  11
```

```
>theArray[1, , 1]
```

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

```
>theArray[, , 1]
```

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

The main difference between an `array` and a `matrix` is that `matrices` are restricted to two dimensions while `arrays` can have an arbitrary number.