UNIT- I

Introduction, How to run R, R Sessions and Functions, Basic Math, Variables, Data Types, Vectors, Conclusion, Advanced Data Structures, Data Frames, Lists, Matrices, Arrays, Classes

Introduction:

R is a programming language and environment commonly used in statistical computing, data analytics and scientific research.

It is one of the most popular languages used by statisticians, data analysts, researchers and marketers to retrieve, clean, analyze, visualize and present data.

Due to its expressive syntax and easy-to-use interface, it has grown in popularity in recent years.

- R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R DevelopmentCore Team.
- The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions.
- R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.
- R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.
- R is free software distributed under a GNU-style copy left, and an official part of the GNU project called **GNU S**

Features of R

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. The following are the important features of R:

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

As a conclusion, R is world’s most widely used statistics programming language. It's the#1 choice of data scientists and supported by a vibrant and talented community of contributors. R is taught in universities and deployed in mission critical business applications.

Things to Know Before Start Learning R

Why use R

- R is an open source programming language and software environment for statistical computing and graphics.
- R is an object oriented programming environment, much more than most other statistical software packages.
- R is a comprehensive statistical platform, offering all manner of data-analytic techniques – any type of data analysis can done in R.
- R has state-of-the-art graphics capabilities- visualize complex data.
- R is a powerful platform for interactive data analysis and exploration.
- Getting data into a usable form from multiple sources .
- R functionality can be integrated into applications written in other languages, including C++, Java, Python , PHP, SAS and SPSS.
- R runs on a wide array of platforms, including Windows, Unix and Mac OS X.
- R is extensible; can be expanded by installing “packages”

Why use R for statistical computing and graphics?

1. **R is open source and free!**
   R is free to download as it is licensed under the terms of GNU General Public license.

   You can look at the source to see what’s happening under the hood. There’s more, most R packages are available under the same license so you can use them, even in commercial applications without having to call your lawyer.

2. **R is popular - and increasing in popularity**
   IEEE publishes a list of the most popular programming languages each year. R was ranked 5th in 2016, up from 6th in 2015. It is a big deal for a domain-specific language like R to be more popular than a general purpose language like C#.

   This not only shows the increasing interest in R as a programming language, but also of the fields like Data Science and Machine Learning where R is commonly used.

3. **R runs on all platforms**
   You can find distributions of R for all popular platforms - Windows, Linux and Mac.

   R code that you write on one platform can easily be ported to another without any issues. Cross-platform interoperability is an important feature to have in today’s computing world - even Microsoft is making its coveted .NET platform available on all platforms after realizing the benefits of technology that runs on all systems.

4. **Learning R will increase your chances of getting a job**
   According to the Data Science Salary Survey conducted by O’Reilly Media in 2014, data scientists are paid a median of $98,000 worldwide.

The figure is higher in the US - around $144,000.

Of course, knowing how to write R programs won't get you a job straight away, a data scientist has to juggle a lot of tools to do their work. Even if you are applying for a software developer position, R programming experience can make you stand out from the crowd.

5. **R is being used by the biggest tech giants**
   Adoption by tech giants is always a sign of a programming language's potential. Today's companies don't make their decisions on a whim. Every major decision has to be backed by concrete analysis of data.
   Companies Using R
   R is the right mix of simplicity and power, and companies all over the world use it to make calculated decisions. Here are a few ways industry stalwarts are using R and contributing to the R ecosystem.

Company	Application/Contribution
Twitter	Monitor user experience
Ford	Analyse social media to support design decisions for their cars
New York Times	Infographics, data journalism
Microsoft	Released Microsoft R Open, an enhanced R distribution and Microsoft R server after acquiring Revolution Analytics in 2015
Human Rights Data Analysis Group	Measure the impact of war
Google	Created the R style guide for the R user community inside Google

While using R, you can rest assured that you are standing on the shoulders of giants.

Is R programming an easy language to learn?

This is a difficult question to answer. Many researchers are learning R as their first language to solve their data analysis needs.

That’s the power of the R programming, it is simple enough to learn as you go. All you need is data and a clear intent to draw a conclusion based on analysis on that data.

In fact, R is built on top of the language *S programming* that was originally intended as a programming language that would help the student learn programming while playing around with data.

However, programmers that come from a Python, PHP or Java background might find R quirky and confusing at first. The syntax that R uses is a bit different from other common programming languages.

While R does have all the capabilities of a programming language, you will not find yourself writing a lot of if conditions or loops while writing code in the R language. There are other programming constructs like vectors, lists, frames, data tables, matrices etc. that allow you to perform transformations on data in bulk.

Applications of R Programming in Real World

1. **Data Science**
   Harvard Business Review named **data scientist the "sexiest job of the 21st century"**. Glassdoor named it the "best job of the year" for 2016. With the advent of IoT devices creating terabytes and terabytes of data that can be used to make better decisions, data science is a field that has no other way to go but up.

   Simply explained, a data scientist is a statistician with an extra asset: computer programming skills. Programming languages like R give a data scientist superpowers that allow them to collect data in realtime, perform statistical and predictive analysis, create visualizations and communicate actionable results to stakeholders.

   Most courses on data science include R in their curriculum because it is the data scientist’s favourite tool.

2. **Statistical computing**
   R is the most popular programming language among statisticians. In fact, it was initially built by statisticians for statisticians. It has a rich package repository with more than 9100 packages with every statistical function you can imagine.

   R’s expressive syntax allows researchers - even those from non computer science backgrounds to quickly import, clean and analyze data from various data sources.
   R also has charting capabilities, which means you can plot your data and create interesting visualizations from any dataset.

3. **Machine Learning**
   R has found a lot of use in predictive analytics and machine learning. It has various package for common ML tasks like linear and non-linear regression, decision trees, linear and non-linear classification and many more.

> Everyone from machine learning enthusiasts to researchers use R to implement machine learning algorithms in fields like finance, genetics research, retail, marketing and health care.

Alternatives to R programming

R is not the only language that you can use for statistical computing and graphics. Some of the popular alternatives of R programming are:

Python - Popular general purpose language

Python is a very powerful high-level, object-oriented programming language with an easy-to-use and simple syntax.

Python is extremely popular among data scientists and researchers. Most of the packages in R have equivalent libraries in Python as well.

While R is the first choice of statisticians and mathematicians, professional programmers prefer implementing new algorithms in a programming language they already know.

The choice between R vs Python also depends on what you are trying to accomplish with your code. If you are trying to analyze a dataset and present the findings in a research paper, then R is probably a better choice. But if you are writing a data analysis program that runs in a distributed system and interacts with lots of other components, it would be preferable to work with Python.

SAS (Statistical Analysis System)

SAS is a powerful software that has been the first choice of private enterprise for their analytics needs for a long time. Its GUI and comprehensive documentation, coupled with reliable technical support make it a very good tool for companies.

While R is the undisputed champion in academics and research, SAS is extremely popular in commercial analytics. But R and Python are gaining momentum in the enterprise space and companies are also trying to move towards open-source technologies. Time will tell if SAS will continue its dominance or R/Python will take over.

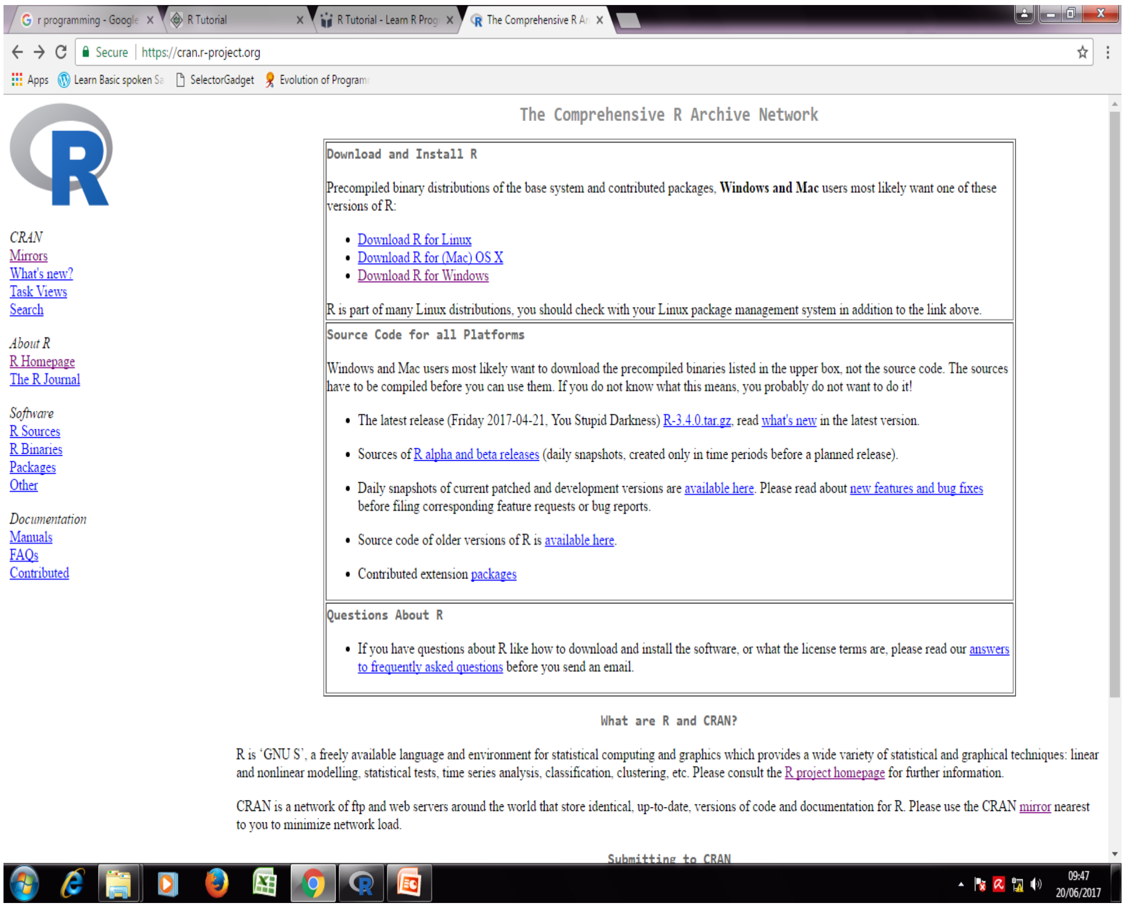
SPSS - Software package for statistical analysis

SPSS is another popular statistical tool. It is used most commonly in the social sciences and is considered the easiest to learn among enterprise statistical tools.

SPSS is loved by non-statisticians because it is similar to excel so those who are already familiar with it will find SPSS very easy to use.

SPSS has the same downside as SAS - it is expensive. SPSS was acquired by IBM in 2009 for a reported $1.2 billion.

Downloading and Installing R

- R is free available from the comprehensive R Archive Network (CRAN) at http://cran.r-project.org
- Precompiled binaries are available for Linux, Mac OS X and windows.
- R latest release R-3.4.0
- Installing R on windows and Mac is just like installing any other program.
- Install R Studio: a free IDE for R at http://www.rstudio.com/
- If we install R and R Studio, then we need to run R Studio only.
- R is case-sensitive.
- R scripts are simply text files with a .R extension.
- 

The Comprehensive R Archive Network

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (Friday 2017-04-21, You Stupid Darkness) R-3.4.0.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

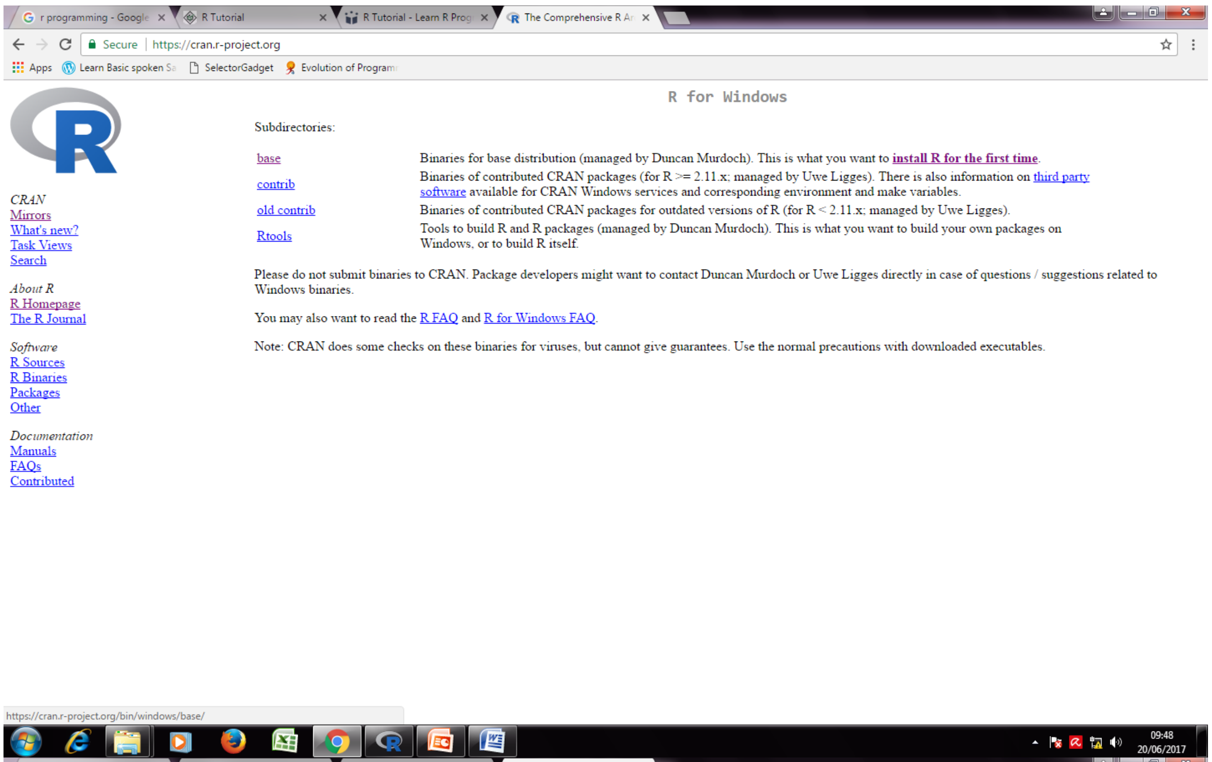
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to minimize network load.

Submitting to CRAN

- 

R for Windows

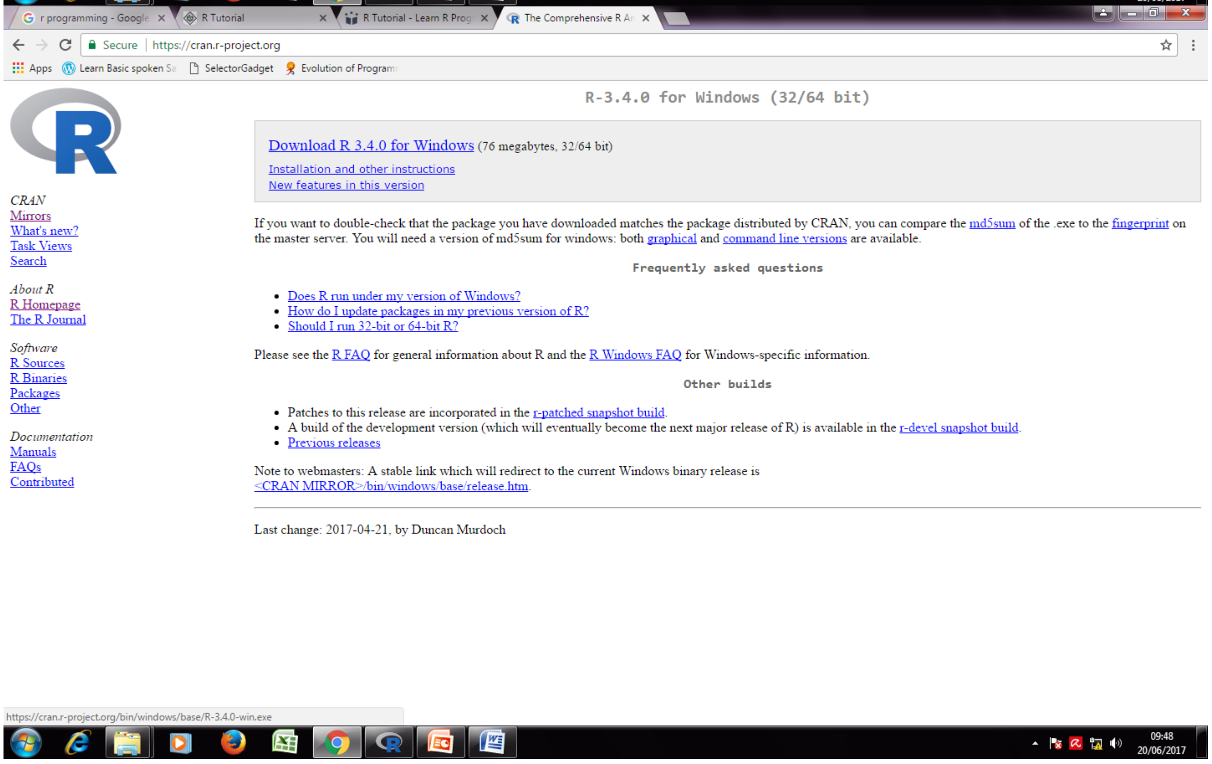
Subdirectories:

base	Binaries for base distribution (managed by Duncan Murdoch). This is what you want to install R for the first time.
contrib	Binaries of contributed CRAN packages (for R >= 2.11.x; managed by Uwe Ligges). There is also information on third party software available for CRAN Windows services and corresponding environment and make variables.
old contrib	Binaries of contributed CRAN packages for outdated versions of R (for R < 2.11.x; managed by Uwe Ligges).
Rtools	Tools to build R and R packages (managed by Duncan Murdoch). This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Duncan Murdoch or Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the R FAQ and R for Windows FAQ.

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.

- 

R-3.4.0 for Windows (32/64 bit)

Download R 3.4.0 for Windows (76 megabytes, 32/64 bit)
Installation and other instructions
New features in this version

If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare the md5sum of the .exe to the fingerprint on the master server. You will need a version of md5sum for windows: both graphical and command line versions are available.

Frequently asked questions

- Does R run under my version of Windows?
- How do I update packages in my previous version of R?
- Should I run 32-bit or 64-bit R?

Please see the R FAQ for general information about R and the R Windows FAQ for Windows-specific information.

Other builds

- Patches to this release are incorporated in the r-patched snapshot build.
- A build of the development version (which will eventually become the next major release of R) is available in the r-devel snapshot build.
- Previous releases

Note to webmasters: A stable link which will redirect to the current Windows binary release is
<CRAN MIRROR>/bin/windows/base/release.htm.

Last change: 2017-04-21, by Duncan Murdoch

Run R Programming on Your Computer

You will find the easiest way to run R programming on your system (Windows) in this section.

Run R Programming in Windows

1. Go to official site of R programming(https://www.r-project.org/)
2. Click on the CRAN link on the left sidebar
3. Select a mirror

4. Click "Download R for Windows"

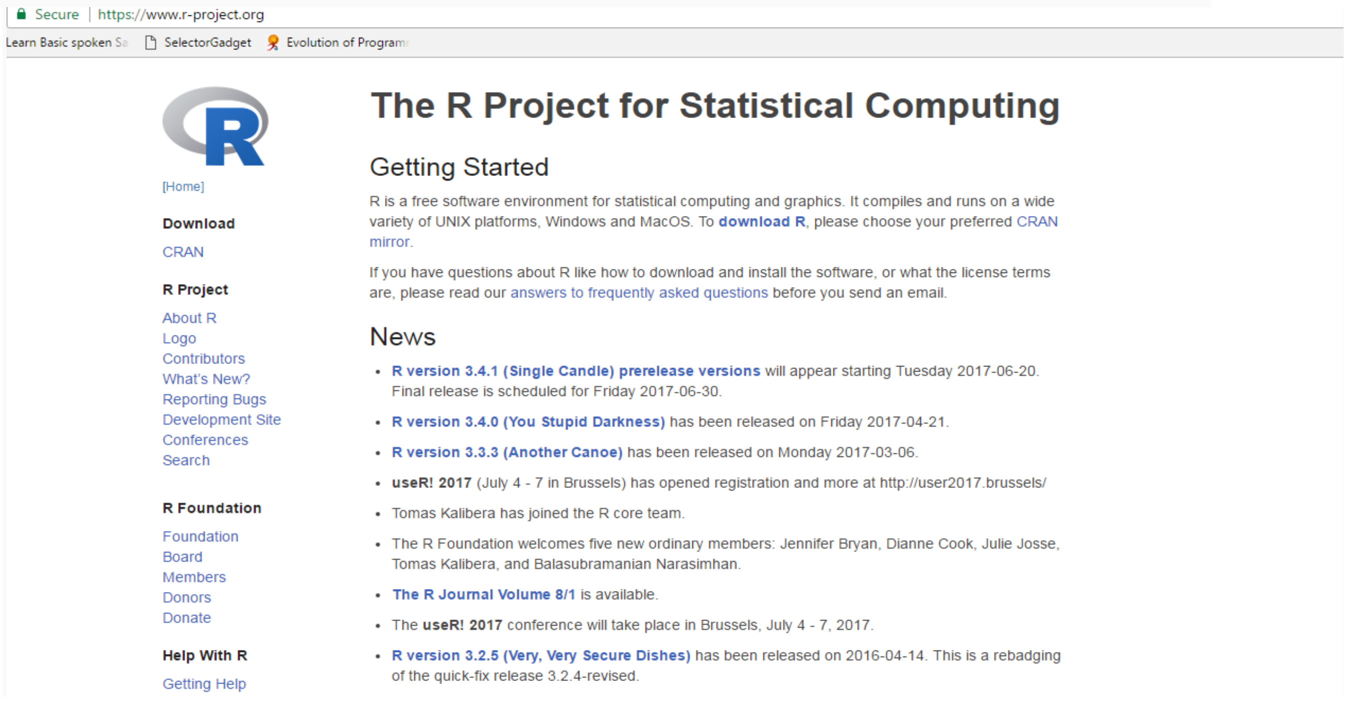
5. Click on the link that downloads the base distribution

6. Run the file and follow the steps in the instructions to install R.

Should I install the 32-bit version or the 64-bit version?

Most people don't need to worry about this. Obviously the 64-bit version of R won't work on a 32-bit machine but both the 32-bit and 64-bit versions of R runs seamlessly on 64-bit Windows.

You might want to consider installing 32-bit version of R if your production environment is 32-bit because some packages might have compatibility issues and might cause the "But it works on my machine" fiasco.



Secure | https://www.r-project.org

Learn Basic spoken Sa SelectorGadget Evolution of Program

[Home]

Download

CRAN

R Project

About R
Logo
Contributors
What's New?
Reporting Bugs
Development Site
Conferences
Search

R Foundation

Foundation
Board
Members
Donors
Donate

Help With R

Getting Help

The R Project for Statistical Computing

Getting Started

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To **download R**, please choose your preferred CRAN mirror.

If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

News

- **R version 3.4.1 (Single Candle) prerelease versions** will appear starting Tuesday 2017-06-20. Final release is scheduled for Friday 2017-06-30.
- **R version 3.4.0 (You Stupid Darkness)** has been released on Friday 2017-04-21.
- **R version 3.3.3 (Another Canoe)** has been released on Monday 2017-03-06.
- **useR! 2017** (July 4 - 7 in Brussels) has opened registration and more at http://user2017.brussels/
- Tomas Kalibera has joined the R core team.
- The R Foundation welcomes five new ordinary members: Jennifer Bryan, Dianne Cook, Julie Josse, Tomas Kalibera, and Balasubramanian Narasimhan.
- **The R Journal Volume 8/1** is available.
- The **useR! 2017** conference will take place in Brussels, July 4 - 7, 2017.
- **R version 3.2.5 (Very, Very Secure Dishes)** has been released on 2016-04-14. This is a rebadging of the quick-fix release 3.2.4-revised.

Getting help in R

To get help on specific topics, we can use the help() function along with the topic we want to search. We can also use the ? operator for this.

```
> help(Syntax)

> ?Syntax
```

We also have the help.search() function to do a search engine type of search. We could use the ?? operator for this.

```
> help.search("histograms")
> ??"histograms"
```

You must be itching to start learning R by now. Our collection of R tutorials will help you learn R. Whether you are a beginner or an expert, each tutorial explains the relevant concepts and syntax with easy-to-understand examples.

R sessions

1. Starting an R session

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. First we will learn the R-session.

To start an R session, type 'R' from the command line in windows or linux OS.

For example, from shell prompt '$' in linux, type

```
$ R
```

This generates the following output before entering the '>>' prompt of R:

```
R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-unknown-linux-gnu (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.

  Natural language support but running in an English locale

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.

[Previously saved workspace restored]

 >
```

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 loose 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 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*.

In Linux, this "working directory" is generally the directory from where R was started through the command 'R'. In windows, it can be either "My Documents" or user's home directory.

When we start R in the same currnt directory next time, the work space 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 = las()) , all objects created so far will be removed. See below:

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

> ls()

[1] "a" "b" "c" "sum"

>


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

>


 > 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 wor 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.

Getting file information from R session

When we are inside R prompt, the operation system commands will not be recognised by R. If we want to list the names of files in the current directory in which R has been started, we should use list.files() commnd to list the files. This lists all the files in the current directory.

In case we need information on a specific file, use file.info("filename") command. This prints all the information about this file on the screen.

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 of double quote"
}
myString <- "Hello, World!"
print ( myString)
```

Though above comments will be executed by R interpreter, they will not interfere with

your actual program. You should put such comments inside, either single or double quote.

R Reserved Words

Reserved words in R programming are a set of words that have special meaning and cannot be used as an identifier (variable name, function name etc.).

Here is a list of reserved words in the R's parser.

Reserved words in R

if	else	repeat	while	function
for	in	next	break	TRUE
FALSE	NULL	Inf	NaN	NA
NA_integer_	NA_real_	NA_complex_	NA_character_	...

This list can be viewed by typing `help(reserved)` or `?reserved` at the R command prompt as follows.

```
> ?reserved
```

Among these words, `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next` and `break` are used for conditions, loops and user defined functions.

They form the basic building blocks of programming in R.

`TRUE` and `FALSE` are the logical constants in R.

`NULL` represents the absence of a value or an undefined value.

`Inf` is for "Infinity", for example when 1 is divided by 0 whereas `NaN` is for "Not a Number", for example when 0 is divided by 0.

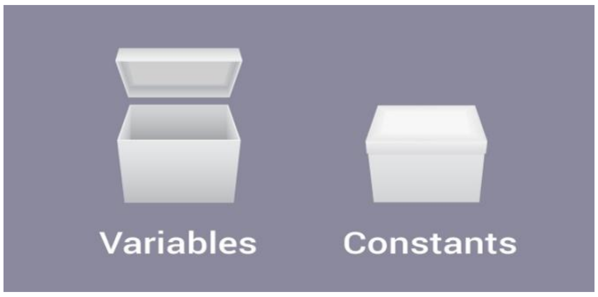
`NA` stands for "Not Available" and is used to represent missing values.

R is a case sensitive language. Which mean that `TRUE` and `True` are not the same.

While the first one is a reserved word denoting a logical constant in R, the latter can be used a variable name.

```
> TRUE <- 1
Error in TRUE <- 1 : invalid (do_set) left-hand side to assignment


> True <- 1


> TRUE
[1] TRUE


> True
[1] 1
```

R Variables and Constants



Variables in R

Variables are used to store data, whose value can be changed according to our need. Unique name given to variable (function and objects as well) is identifier.

Rules for writing Identifiers in R

1. Identifiers can be a combination of letters, digits, period (.) and underscore (_).
2. It must start with a letter or a period. If it starts with a period, it cannot be followed by a digit.
3. Reserved words in R cannot be used as identifiers.

Valid identifiers in R

`total`, `Sum`, `.fine.with.dot`, `this_is_acceptable`, `Number5`

Invalid identifiers in R

`tot@l`, `5um`, `_fine`, `TRUE`, `.0ne`

Best Practices

Earlier versions of R used underscore (_) as an assignment operator. So, the period (.) was used extensively in variable names having multiple words.

Current versions of R support underscore as a valid identifier but it is good practice to use period as word separators.

For example, `a.variable.name` is preferred over `a_variable_name` or alternatively we could use camel case as `aVariableName`

Constants in R

Constants, as the name suggests, are entities whose value cannot be altered. Basic types of constant are numeric constants and character constants.

Numeric Constants

All numbers fall under this category. They can be of type `integer`, `double` or `complex`.

It can be checked with the `typeof()` function.

Numeric constants followed by `L` are regarded as `integer` and those followed by `i` are regarded as `complex`.

```
> typeof(5)
[1] "double"


> typeof(5L)
[1] "integer"


> typeof(5i)
[1] "complex"
```

Numeric constants preceded by `0x` or `0X` are interpreted as hexadecimal numbers.

```
> 0xff
[1] 255

> 0XF + 1
[1] 16
```

Character Constants

Character constants can be represented using either single quotes (') or double quotes (") as delimiters.

```
> 'example'
[1] "example"

> typeof("5")
[1] "character"
```

Built-in Constants

Some of the built-in constants defined in R along with their values is shown below.

```
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P"
"Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"

> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
"q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

> pi
 [1] 3.141593
```

```
> month.name
 [1] "January"   "February"  "March"     "April"     "May"
"June"
 [7] "July"      "August"    "September" "October"   "November"
"December"


> month.abb
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
"Nov" "Dec"
```

But it is not good to rely on these, as they are implemented as variables whose values can be changed.

```
> pi
[1] 3.141593


> pi <- 56
> pi
[1] 56
```

Example: Hello World Program

```
> # We can use the print() function
> print("Hello World!")
[1] "Hello World!"


> # Quotes can be suppressed in the output
> print("Hello World!", quote = FALSE)
[1] Hello World!


> # If there are more than 1 item, we can concatenate using paste()
> print(paste("How","are","you?"))
[1] "How are you?"
```

R - Data Types

Generally, while doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but ***reserved memory locations to store values***. This means that, when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

In contrast to other programming languages like C and java in R, the variables are not declared as some data type. ***The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable***. There are many types of R-objects. The frequently used ones are –

- Vectors
- Lists
- Matrices
- Arrays
- Factors
- Data Frames

The simplest of these objects is the **vector object** and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

Data Type	Example	Verify

Logical	TRUE, FALSE	`v <- TRUE`
		`print(class(v))`
		it produces the following result –
		`[1] "logical"`

Numeric	12.3, 5, 999	`v <- 23.5`
		`print(class(v))`
		it produces the following result –
		`[1] "numeric"`

Integer	2L, 34L, 0L	`v <- 2L`

		`print(class(v))`
		it produces the following result –
		`[1] "integer"`

Complex	3 + 2i	`v <- 2+5i`
		`print(class(v))`
		it produces the following result –
		`[1] "complex"`

Character	'a' , '"good", "TRUE", '23.4'	`v <- "TRUE"`
		`print(class(v))`
		it produces the following result –
		`[1] "character"`

Raw	"Hello" is stored as 48 65 6c 6c 6f	`v <- charToRaw("Hello")`
		`print(class(v))`
		it produces the following result –
		`[1] "raw"`

In R programming, the very basic data types are the R-objects called **vectors** which hold elements of different classes as shown above. Please note in R the number of classes is not confined to only the above six types. For example, we can use many atomic vectors and create an array whose class will become array.

Understanding basic data types in R

- To make the best of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on those.
- **Very Important** to understand because these are the things you will manipulate on a day-to-day basis in R. Most common source of frustration among beginners.
- Everything in `R` is an object.

`R` has 5 basic atomic classes

- logical (e.g., `TRUE`, `FALSE`)
- integer (e.g,, 2L, as.integer(3))
- numeric (real or decimal) (e.g, 2, 2.0, pi)
- complex (e.g, 1 + 0i, 1 + 4i)
- character (e.g, "a", "swc")

```
typeof() # what is it?
class() # what is it? (sorry)
storage.mode() # what is it? (very sorry)
length() # how long is it? What about two dimensional objects?
attributes() # does it have any metadata?
```

R also has many data structures. These include

- vector
- list
- matrix
- data frame
- factors (we will avoid these, but they have their uses)
- tables

Vectors

A vector is the most common and basic data structure in `R` and is pretty much the workhorse of R. Vectors can be of two types:

- atomic vectors
- lists

Atomic Vectors A vector can be a vector of characters, logical, integers or numeric.

Create an empty vector with `vector()`

```
x <- vector()
# with a pre-defined length
x <- vector(length = 10)
# with a length and type
vector("character", length = 10)
vector("numeric", length = 10)
vector("integer", length = 10)
vector("logical", length = 10)
```

The general pattern is `vector(class of object, length)`. You can also create vectors by concatenating them using the `c()` function.

Various examples:

```
x <- c(1, 2, 3)
```

x is a numeric vector. These are the most common kind. They are numeric objects and are treated as double precision real numbers. To explicitly create integers, add a `L` at the end.

```
x1 <- c(1L, 2L, 3L)
```

You can also have logical vectors.

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

(Don't use `T` and `F`!)

Finally you can have character vectors:

```
z <- c("Alec", "Dan", "Rob", "Rich")
```

Examine your vector

```
typeof(z)
length(z)
class(z)
str(z)
```

Question: Do you see property that's common to all these vectors above?

Add elements

```
z <- c(z, "Annette")
z
```

More examples of vectors

```
x <- c(0.5, 0.7)
x <- c(TRUE, FALSE)
x <- c("a", "b", "c", "d", "e")
x <- 9:100
x <- c(i+0i, 2+4i)
```

You can also create vectors as sequence of numbers

```
series <- 1:10
seq(10)
seq(1, 10, by = 0.1)
```

Other objects

`Inf` is infinity. You can have positive or negative infinity.

```
 1/0
# [1] Inf
 1/Inf
# [1] 0
```

`NaN` means Not a number. it's an undefined value.

```
0/0
NaN.
```

Each object has an attribute. Attributes can be part of an object of R. These include

- names
- dimnames
- length
- class

- attributes (contain metadata)

For a vector, `length(vector_name)` is just the total number of elements.

Vectors may only have one type

R will create a resulting vector that is the least common denominator. The coercion will move towards the one that's easiest to coerce to.

Guess what the following do without running them first

```
xx <- c(1.7, "a")
xx <- c(TRUE, 2)
xx <- c("a", TRUE)
```

This is called implicit coercion.

The coersion rule goes `logical` -> `integer` -> `numeric` -> `complex` -> `character`.

You can also coerce vectors explicitly using the `as.<class_name>`. Example

```
as.numeric()
as.character()
```

When you coerce an existing numeric vector with `as.numeric()`, it does nothing.

```
x <- 0:6
as.numeric(x)
as.logical(x)
as.character(x)
as.complex(x)
```

Sometimes coercions, especially nonsensical ones won't work.

```
x <- c("a", "b", "c")
as.numeric(x)
as.logical(x)
# both don't work
```

Sometimes there is implicit conversion

```
1 < "2"
# TRUE
"1" > 2
# FALSE
1 < "a"
# TRUE
```

Matrix

Matrices are a special vector in R. They are not a separate class of object but simply a vector but now with dimensions added on to it. Matrices have rows and columns.

```
m <- matrix(nrow = 2, ncol = 2)
m
dim(m)
same as
```

```
attributes(m)
```

Matrices are constructed columnwise.

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

Other ways to construct a matrix

```
m <- 1:10
dim(m) <- c(2,5)
```

This takes a vector and transform into a matrix with 2 rows and 5 columns.

Another way is to bind columns or rows using `cbind()` and `rbind()`.

```
x <- 1:3
y <- 10:12
cbind(x,y)
# or
rbind(x,y)
```

List

In R lists act as containers. Unlike atomic vectors, its contents are not restricted to a single mode and can encompass any data type. Lists are sometimes called recursive vectors, because a list can contain other lists. This makes them fundamentally different from atomic vectors.

List is a special vector. Each element can be a different class.

Create lists using `list` or coerce other objects using `as.list()`

```
x <- list(1, "a", TRUE, 1+4i)
x <- 1:10
x <- as.list(x)
length(x)
```

What is the class of `x[1]`? how about `x[[1]]`?

```
xlist <- list(a = "Rich FitzJohn", b = 1:10, data = head(iris))
```

what is the length of this object? what about its structure?

List can contain as many lists nested inside.

```
temp <- list(list(list(list())))
temp
is.recursive(temp)
```

Lists are extremely useful inside functions. You can "staple" together lots of different kinds of results into a single object that a function can return.

It doesn't print out like a vector. Prints a new line for each element.

Elements are indexed by double brackets. Single brackets will still return another list.

Factors

Factors are special vectors that represent categorical data. Factors can be ordered or unordered and are important when for modelling functions such as `lm()` and `glm()` and also in plot methods.

Factors can only contain pre-defined values.

Factors are pretty much integers that have labels on them. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings. Some string methods will coerce factors to strings, while others will throw an error.

Sometimes factors can be left unordered. Example: male, female

Other times you might want factors to be ordered (or ranked). Example: low, medium, high.

Underlying it's represented by numbers 1,2,3.

They are better than using simple integer labels because factors are what are called self describing. male and female is more descriptive than 1s and 2s. Helpful when there is no additional metadata.

Which is male? 1 or 2? You wouldn't be able to tell with just integer data. Factors have this information built in.

Factors can be created with `factor()`. Input is a character vector.

```
x <- factor(c("yes", "no", "no", "yes", "yes"))
x
```

`table(x)` will return a frequency table.

`unclass(x)` strips out the class information.

In modelling functions, important to know what baseline levels is. This is the first factor but by default the ordering is determined by alphabetical order of words entered. You can change this by specifying the levels.

```
x <- factor(c("yes", "no", "yes"), levels = c("yes", "no"))
```

Data frame

A data frame is a very important data type in R. It's pretty much the de facto data structure for most tabular data and what we use for statistics.

data frames can have additional attributes such as `rownames()`. This can be useful for annotating data, like subject_id or sample_id. But most of the time they are not used.

e.g. `rownames()` useful for annotating data. subject names. other times they are not useful.

- Data frames Usually created by read.csv and read.table.
- Can convert to `matrix` with `data.matrix()`
- Coercion will force and not always what you expect.
- Can also create with `data.frame()` function.

With and data frame, you can do `nrow(df)` and `ncol(df)` rownames are usually 1..n.

Combining data frames

```
df <- data.frame(id = letters[1:10], x = 1:10, y = rnorm(10))
> df
   id  x          y
1   a  1 -0.3913992
2   b  2 -0.8607609
3   c  3  1.1234612
4   d  4 -0.8283688
5   e  5 -0.8785586
6   f  6  0.2116839
7   g  7 -0.3795995
8   h  8 -0.5992272
9   i  9  0.3203085
10  j 10  0.2901185

cbind(df, data.frame(z = 4))
```

When you combine column wise, only row numbers need to match. If you are adding a vector, it will get repeated.

Useful functions `head()` - see first 5 rows `tail()` - see last 5 rows `dim()` - see dimensions `nrow()` - number of rows `ncol()` - number of columns `str()` - structure of each column `names()` - will list column names for a data.frame (or any object really).

A data frame is a special type of list where every element of a list has same length.

See that it is actually a special list:

```
> is.list(iris)
[1] TRUE
> class(iris)
[1] "data.frame"
 > --
```

Naming objects

Other R objects can also have names not just true for data.frames. Adding names is helpful since it's useful for readable code and self describing objects.

```
x <- 1:3
names(x) <- c("rich", "daniel", "diego")
```

```
x
```

Lists can also have names.

```
x <- as.list(1:10)
names(x) <- letters[seq(x)]
x
```

Finally matrices can have names and these are called `dimnames`

```
m <- matrix(1:4, nrow = 2)
dimnames(m) <- list(c("a", "b"), c("c", "d"))
# first element = rownames
# second element = colnames
```

Missing values

denoted by `NA` and/or `NaN` for undefined mathematical operations.

```
is.na()
is.nan()
```

check for both.

NA values have a class. So you can have both an integer NA and a missing character NA.

NaN is also NA. But not the other way around.

```
x <- c(1,2, NA, 4, 5)
```

`is.na(x)` returns logical. shows third

`is.nan(x)` # none are NaN.

```
x <- c(1,2, NA, NaN, 4, 5)
```

`is.na(x)` shows 2 TRUE. `is.nan(x)` shows 1 TRUE

Missing values are very important in R, but can be very frustrating for new users.

What do these do? What should they do? ~~~ 1 == NA NA == NA ~~~

How can we do that sort of comparison?

Diagnostic functions in R

Super helpful functions

- `str()` Compactly display the internal structure of an R object. Perhaps the most uesful diagnostic function in R.
- `names()` Names of elements within an object
- `class()` Retrieves the internal class of an object

- `mode()` Get or set the type or storage mode of an object.
- `length()` Retrieve or set the dimension of an object.
- `dim()` Retrieve or set the dimension of an object.
- `R --vanilla` - Allows you to start a clean session of R. A great way to test whether your code is reproducible.
- `sessionInfo()` Print version information about R and attached or loaded packages.
- `options()` Allow the user to set and examine a variety of global options which affect the way in which R computes and displays its results.

`str()` is your best friend

`str` is short for structure. You can use it on any object. Try the following:

```
x <- 1:10
class(x)
mode(x)
str(x)
```

`sessionInfo()`: Print version information about R and attached or loaded packages.

R Operators

R has several operators to perform tasks including arithmetic, logical and bitwise operations.

R has many operators to carry out different mathematical and logical operations.

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

Type of operators in R

Arithmetic operators
Relational operators
Logical operators
Assignment operators

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
%%	Modulus (Remainder from 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
```

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
```

R Assignment Operators

These operators are used to assign values to variables.

Assignment Operators in R

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

Example: Hello World Program

```
> # We can use the print() function
> print("Hello World!")
[1] "Hello World!"

> # Quotes can be suppressed in the output
> print("Hello World!", quote = FALSE)
[1] Hello World!

> # If there are more than 1 item, we can concatenate using paste()
> print(paste("How","are","you?"))
[1] "How are you?"
```

R Program to Take Input From User

In this example, you'll learn to take input from a user using readline() function.

When we are working with R in an interactive session, we can use `readline()` function to take input from the user (terminal).

This function will return a single element character vector.

So, if we want numbers, we need to do appropriate conversions.

Example: Take input from user

```
my.name <- readline(prompt="Enter name: ")
my.age <- readline(prompt="Enter age: ")

# convert character into integer
my.age <- as.integer(my.age)

print(paste("Hi,", my.name, "next year you will be", my.age+1,
"years old."))
```

Output

```
Enter name: Mary

Enter age: 17

[1] "Hi, Mary next year you will be 18 years old."
```

Here, we see that with the `prompt` argument we can choose to display an appropriate message for the user.

In the above example, we convert the input age, which is a character vector into integer using the function `as.integer()`.

This is necessary for the purpose of doing further calculations.

Example: Multiplication Table

```
# Program to find the multiplication
# table (from 1 to 10)
# of a number input by the user

# take input from the user
```

```
num = as.integer(readline(prompt = "Enter a number: "))


# use for loop to iterate 10 times
for(i in 1:10) {
    print(paste(num,'x', i, '=', num*i))
}
```

Vector is a basic data structure in R. It contains element of the same type. The data types can be logical, integer, double, character, complex or raw.

A vector's type can be checked with the `typeof()` function.

Another important property of a vector is its length. This is the number of elements in the vector and can be checked with the function `length()`.

How to create a Vector in R programming?

Vectors are generally created using the `c()` function.

Since, a vector must have elements of the same type, this function will try and coerce elements to the same type, if they are different.

Coercion is from lower to higher types from logical to integer to double to character.

```
> x <- c(1, 5, 4, 9, 0)
> typeof(x)
[1] "double"
> length(x)
[1] 5

> x <- c(1, 5.4, TRUE, "hello")
> x
[1] "1"     "5.4"   "TRUE"  "hello"
> typeof(x)
[1] "character"
```

If we want to create a vector of consecutive numbers, the `:` operator is very helpful.

Example 1: Creating a vector using : operator

```
> x <- 1:7; x
[1] 1 2 3 4 5 6 7

> y <- 2:-2; y
[1]  2  1  0 -1 -2
```

More complex sequences can be created using the `seq()` function, like defining number of points in an interval, or the step size.

Example 2: Creating a vector using seq() function

```
> seq(1, 3, by=0.2)          # specify step size

[1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0



> seq(1, 5, length.out=4)    # specify length of the vector

[1] 1.000000 2.333333 3.666667 5.000000

> seq(from = 1, by = 2, length.out = 10)

 [1]  1  3  5  7  9 11 13 15 17 19

> seq(f = 1, b = 5, leng = 10)

 [1]  1  6 11 16 21 26 31 36 41 46

> seq(0, 1, length.out = 11)

 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> seq(1, 9, by = 2)     # matches 'end'

[1] 1 3 5 7 9

> seq(17) # same as 1:17, or even better seq_len(17)

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

```
> seq_len(17)

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17

> seq(1, 6, by = 3)

[1] 1 4

> seq(1.575, 5.125, by = 0.05)

 [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.925 1.975 2.025
2.075 2.125

[13] 2.175 2.225 2.275 2.325 2.375 2.425 2.475 2.525 2.575 2.625
2.675 2.725

[25] 2.775 2.825 2.875 2.925 2.975 3.025 3.075 3.125 3.175 3.225
3.275 3.325

[37] 3.375 3.425 3.475 3.525 3.575 3.625 3.675 3.725 3.775 3.825
3.875 3.925

[49] 3.975 4.025 4.075 4.125 4.175 4.225 4.275 4.325 4.375 4.425
4.475 4.525

[61] 4.575 4.625 4.675 4.725 4.775 4.825 4.875 4.925 4.975 5.025
5.075 5.125
```

How to access Elements of a Vector?

Elements of a vector can be accessed using vector indexing. The vector used for indexing can be logical, integer or character vector.

Using integer vector as index

Vector index in R starts from 1, unlike most programming languages where index start from 0.

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
[1]  0  2  4  6  8 10

> x[3]           # access 3rd element
[1] 4

> x[c(2, 4)]     # access 2nd and 4th element
[1] 2 6

> x[-1]          # access all but 1st element
[1]  2  4  6  8 10

> x[c(2, -4)]    # cannot mix positive and negative integers
Error in x[c(2, -4)] : only 0's may be mixed with negative
subscripts

> x[c(2.4, 3.54)]    # real numbers are truncated to integers
[1] 2 4
```

Using logical vector as index

When we use a logical vector for indexing, the position where the logical vector is `TRUE` is returned.

This useful feature helps us in filtering of vector as shown below.

```
> x[c(TRUE, FALSE, FALSE, TRUE)]
[1] -3  3

> x[x < 0]  # filtering vectors based on conditions
[1] -3 -1

> x[x > 0]
[1] 3
```

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

Using character vector as index

This type of indexing is useful when dealing with named vectors. We can name each elements of a vector.

```
> x <- c("first"=3, "second"=0, "third"=9)
> names(x)
[1] "first"  "second" "third"

> x["second"]
second
    0

> x[c("first", "third")]
first third
   3     9
```

How to modify a vector in R?

We can modify a vector using the assignment operator.

We can use the techniques discussed above to access specific elements and modify them.

If we want to truncate the elements, we can use reassignments.

```
> x
[1] -3 -2 -1  0  1  2

> x[2] <- 0; x        # modify 2nd element
[1] -3  0 -1  0  1  2

> x[x<0] <- 5; x   # modify elements less than 0
[1] 5 0 5 0 1 2
```

```
> x <- x[1:4]; x      # truncate x to first 4 elements
[1] 5 0 5 0
```

How to delete a Vector?

We can delete a vector by simply assigning a `NULL` to it.

```
> x
[1] -3 -2 -1  0  1  2
> x <- NULL


> x
NULL
> x[4]
NULL
```

R Program to Sort a Vector

Sorting of vectors can be done using the `sort()` function.

By default, it sorts in ascending order. To sort in descending order we can pass `decreasing=TRUE`.

Note that sort is not in-place. This means that the original vector is not effected (sorted).

Only a sorted version of it is returned.

Example: Sort a Vector

```
> x
[1] 7 1 8 3 2 6 5 2 2 4


> # sort in ascending order
> sort(x)
```

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


> # sort in descending order
> sort(x, decreasing=TRUE)
[1] 8 7 6 5 4 3 2 2 2 1


> # vector x remains unaffected
> x
[1] 7 1 8 3 2 6 5 2 2 4
```

Sometimes we would want the index of the sorted vector instead of the values. In such case we can use the function `order()`.

```
> order(x)
[1]  2  5  8  9  4 10  7  6  1  3


> order(x, decreasing=TRUE)
[1]  3  1  6  7 10  4  5  8  9  2


> x[order(x)]    # this will also sort x
[1] 1 2 2 2 3 4 5 6 7 8
```

R Program to Find Minimum and Maximum

In this example, to find the minimum and maximum numbers from a list using min() and max() function respectively and to use the range() function.

We can find the minimum and the maximum of a vector using the `min()` or the `max()`function.

A function called `range()` is also available which returns the minimum and maximum in a two element vector.

Example: Find Minimum and Maximum

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

```
> # find the minimum
> min(x)
[1] 2

> # find the maximum
> max(x)
[1] 10

> # find the range
> range(x)
[1]  2 10
```

If we want to find where the minimum or maximum is located, i.e. the index instead of the actual value, then we can use `which.min()` and `which.max()` functions.

Note that these functions will return the index of the first minimum or maximum in case multiple of them exists.

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

> # find index of the minimum
> which.min(x)
[1] 5

> # find index of the minimum
> which.max(x)
[1] 9

> # alternate way to find the minimum
> x[which.min(x)]
[1] 2
```

R Matrix

In this to work with matrix in R. You will learn to create and modify matrix, and access matrix elements.

Matrix is a two dimensional data structure in R programming.

Matrix is similar to vectors but additionally contains the dimension attribute. All attributes of an object can be checked with the `attributes()` function (dimension can be checked directly with the `dim()` function).

We can check if a variable is a matrix or not with the `class()` function.

```
> a
    [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> class(a)
[1] "matrix"

> attributes(a)
$dim
[1] 3 3

> dim(a)
[1] 3 3
```

Let us consider a matrix as above.

How to create a matrix in R programming?

Matrix can be created using the `matrix()` function.

Dimension of the matrix can be defined by passing appropriate value for arguments `nrow`and `ncol`.

Providing value for both dimension is not necessary. If one of the dimension is provided, the other is inferred from length of the data.

```
> matrix(1:9, nrow = 3, ncol = 3)
    [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> # same result is obtained by providing only one dimension
> matrix(1:9, nrow = 3)
    [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

We can see that the matrix is filled column-wise. This can be reversed to row-wise filling by passing `TRUE` to the argument `byrow`.

```
> matrix(1:9, nrow=3, byrow=TRUE)    # fill matrix row-wise
    [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

In all cases, however, a matrix is stored in column-major order internally as we will see in the subsequent sections.

It is possible to name the rows and columns of matrix during creation by passing a 2 element list to the argument `dimnames`.

```
> x <- matrix(1:9, nrow = 3, dimnames = list(c("X","Y","Z"),
c("A","B","C")))
> x
  A B C
X 1 4 7
Y 2 5 8
Z 3 6 9
```

These names can be accessed or changed with two helpful functions `colnames()` and `rownames()`.

```
> colnames(x)
[1] "A" "B" "C"
> rownames(x)
[1] "X" "Y" "Z"

> # It is also possible to change names
> colnames(x) <- c("C1","C2","C3")
> rownames(x) <- c("R1","R2","R3")

> x
  C1 C2 C3
R1  1  4  7
R2  2  5  8
R3  3  6  9
```

Another way of creating a matrix is by using functions `cbind()` and `rbind()` as in column bind and row bind.

```
> cbind(c(1,2,3),c(4,5,6))
    [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> rbind(c(1,2,3),c(4,5,6))
    [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Finally, you can also create a matrix from a vector by setting its dimension using `dim()`.

```
> x <- c(1,2,3,4,5,6)
> x
[1] 1 2 3 4 5 6
> class(x)
[1] "numeric"
```

```
> dim(x) <- c(2,3)
> x
    [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> class(x)
[1] "matrix"
```

How to access Elements of a matrix?

We can access elements of a matrix using the square bracket `[` indexing method. Elements can be accessed as `var[row, column]`. Here `rows` and `columns` are vectors.

Using integer vector as index

We specify the row numbers and column numbers as vectors and use it for indexing.

If any field inside the bracket is left blank, it selects all.

We can use negative integers to specify rows or columns to be excluded.

```
> x
    [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> x[c(1,2),c(2,3)]    # select rows 1 & 2 and columns 2 & 3
    [,1] [,2]
[1,]    4    7
[2,]    5    8

> x[c(3,2),]    # leaving column field blank will select entire
columns
    [,1] [,2] [,3]
[1,]    3    6    9
```

```
[2,]    2    5    8


> x[,]    # leaving row as well as column field blank will select
entire matrix
    [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9


> x[-1,]    # select all rows except first
    [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

One thing to notice here is that, if the matrix returned after indexing is a row matrix or column matrix, the result is given as a vector.

```
> x[1,]
[1] 1 4 7
> class(x[1,])
[1] "integer"
```

This behavior can be avoided by using the argument `drop = FALSE` while indexing.

```
> x[1,,drop=FALSE]  # now the result is a 1X3 matrix rather than a
vector
    [,1] [,2] [,3]
[1,]    1    4    7
> class(x[1,,drop=FALSE])
[1] "matrix"
```

It is possible to index a matrix with a single vector.

While indexing in such a way, it acts like a vector formed by stacking columns of the matrix one after another. The result is returned as a vector.

```
> x
    [,1] [,2] [,3]
[1,]    4    8    3
[2,]    6    0    7
```

```
[3,]    1    2    9

> x[1:4]
[1] 4 6 1 8

> x[c(3,5,7)]
[1] 1 0 3
```

Using logical vector as index

Two logical vectors can be used to index a matrix. In such situation, rows and columns where the value is `TRUE` is returned. These indexing vectors are recycled if necessary and can be mixed with integer vectors.

```
> x
    [,1] [,2] [,3]
[1,]    4    8    3
[2,]    6    0    7
[3,]    1    2    9

> x[c(TRUE,FALSE,TRUE),c(TRUE,TRUE,FALSE)]
    [,1] [,2]
[1,]    4    8
[2,]    1    2

> x[c(TRUE,FALSE),c(2,3)]    # the 2 element logical vector is recycled to 3 element vector
    [,1] [,2]
[1,]    8    3
[2,]    2    9
```

It is also possible to index using a single logical vector where recycling takes place if necessary.

```
> x[c(TRUE, FALSE)]
[1] 4 1 0 3 9
```

In the above example, the matrix `x` is treated as vector formed by stacking columns of the matrix one after another, i.e., `(4,6,1,8,0,2,3,7,9)`.

The indexing logical vector is also recycled and thus alternating elements are selected. This property is utilized for filtering of matrix elements as shown below.

```
> x[x>5]    # select elements greater than 5
[1] 6 8 7 9

> x[x%%2 == 0]    # select even elements
[1] 4 6 8 0 2
```

Using character vector as index

Indexing with character vector is possible for matrix with named row or column. This can be mixed with integer or logical indexing.

```
> x
     A B C
[1,] 4 8 3
[2,] 6 0 7
[3,] 1 2 9

> x[,"A"]
[1] 4 6 1

> x[TRUE,c("A","C")]
     A C
[1,] 4 3
[2,] 6 7
[3,] 1 9

> x[2:3,c("A","C")]
     A C
[1,] 6 7
[2,] 1 9
```

How to modify a matrix in R?

We can combine assignment operator with the above learned methods for accessing elements of a matrix to modify it.

```
> x
    [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> x[2,2] <- 10; x    # modify a single element
    [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2   10    8
[3,]    3    6    9

> x[x<5] <- 0; x    # modify elements less than 5
    [,1] [,2] [,3]
[1,]    0    0    7
[2,]    0   10    8
[3,]    0    6    9
```

A common operation with matrix is to transpose it. This can be done with the function `t()`.

```
> t(x)    # transpose a matrix
    [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0   10    6
[3,]    7    8    9
```

We can add row or column using `rbind()` and `cbind()` function respectively. Similarly, it can be removed through reassignment.

```
> cbind(x, c(1, 2, 3))    # add column
    [,1] [,2] [,3] [,4]
[1,]    0    0    7    1
[2,]    0   10    8    2
[3,]    0    6    9    3
```

```
> rbind(x,c(1,2,3))    # add row
     [,1] [,2] [,3]
[1,]    0    0    7
[2,]    0   10    8
[3,]    0    6    9
[4,]    1    2    3

> x <- x[1:2,]; x    # remove last row
     [,1] [,2] [,3]
[1,]    0    0    7
[2,]    0   10    8
```

Dimension of matrix can be modified as well, using the `dim()` function.

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

> dim(x) <- c(3,2); x    # change to 3X2 matrix
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> dim(x) <- c(1,6); x    # change to 1X6 matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
```

R Lists

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.

We can check if it's a list with `typeof()` function and find its length using `length()`.

Following is an example of a list having three components each of different data type.

```
> x
$a
[1] 2.5

$b
[1] TRUE

$c
[1] 1 2 3

> typeof(x)
[1] "list"

> length(x)
[1] 3
```

How to create a list in R programming?

List can be created using the `list()` function.

```
> x <- list("a" = 2.5, "b" = TRUE, "c" = 1:3)
```

Here, we create a list `x`, of three components with data types `double`, `logical` and `integer`vector respectively.

Its structure can be examined with the `str()` function.

```
> str(x)
List of 3
$ a: num 2.5
$ b: logi TRUE
$ c: int [1:3] 1 2 3
```

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 as follows. In such scenario, numeric indices are used by default.

```
> x <- list(2.5,TRUE,1:3)


> x

[[1]]

[1] 2.5


[[2]]

[1] TRUE


[[3]]

[1] 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. Let us consider a list as follows.

```
> x

$name

[1] "John"


$age

[1] 19


$speaks

[1] "English" "French"


> x[c(1:2)]    # index using integer vector

$name

[1] "John"


$age

[1] 19
```

```
> x[-2]        # using negative integer to exclude second component
$name
[1] "John"


$speaks
[1] "English" "French"


> x[c(T,F,F)]  # index using logical vector
$name
[1] "John"


> x[c("age","speaks")]    # index using character vector
$age
[1] 19


$speaks
[1] "English" "French"
```

Indexing with `[` as shown above will give us sublist not the content inside the component. To retrieve the content, we need to use `[[`.

However, this approach will allow us to access only a single component at a time.

```
> x["age"]
$age
[1] 19


> typeof(x["age"])    # single [ returns a list
[1] "list"


> x[["age"]]    # double [[ returns the content
[1] 19


> typeof(x[["age"]])
[1] "double"
```

An alternative to `[[`, which is used often while accessing content of a list is the `$` operator. They are both the same except that `$` can do partial matching on tags.

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

> x$a                  # partial matching, same as x$ag or x$age
[1] 19

> x[["a"]]             # cannot do partial match with [[
NULL

> # indexing can be done recursively
> x$speaks[1]
[1] "English"

> x[["speaks"]][2]
[1] "French"
```

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.

Notice below that modification causes reordering of components.

```
> x[["name"]] <- "Clair"; x
$age
[1] 19

$speaks
[1] "English" "French"

$name
[1] "Clair"
```

How to add components to a list?

Adding new components is easy. We simply assign values using new tags and it will pop into action.

```
> x[["married"]] <- FALSE
> x
$age
[1] 19

$speaks
[1] "English" "French"

$name
[1] "Clair"

$married
[1] FALSE
```

How to delete components from a list?

We can delete a component by assigning `NULL` to it.

```
> x[["age"]] <- NULL
> str(x)
List of 3
$ speaks : chr [1:2] "English" "French"
$ name   : chr "Clair"
$ married: logi FALSE

> x$married <- NULL
> str(x)
List of 2
$ speaks: chr [1:2] "English" "French"
$ name  : chr "Clair"
```

R 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.

Each component form the column and contents of the component form the rows.

Check if a variable is a data frame using class()

We can check if a variable is a data frame or not using the `class()` function.

```
> x
 SN Age Name
1  1  21 John
2  2  15 Dora

> typeof(x)    # data frame is a special case of  list
[1] "list"

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

In this example, `x` can be considered as a list of 3 components with each component having a two element vector. Some useful functions to know more about a data frame are given below.

Functions of data frame

```
> names(x)
[1] "SN"   "Age"  "Name"

> ncol(x)
[1] 3

> nrow(x)
[1] 2
```

```
> length(x)    # returns length of the list, same as ncol()
[1] 3
```

How to create a Data Frame in R?

We can create a data frame using the `data.frame()` function.

For example, the above shown data frame can be created as follows.

```
> 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
```

Notice above that the third column, `Name` is of type factor, instead of a character vector.

By default, `data.frame()` function converts character vector into factor.

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"
```

Many data input functions of R like, `read.table()`, `read.csv()`, `read.delim()`, `read.fwf()` also read data into a data frame.

How to access Components of a Data Frame?

Components of data frame can be accessed like a list or like a matrix.

Accessing like a list

We can use either `[`, `[[` or `$` operator to access columns of data frame.

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

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

> x[["Name"]]
[1] "John" "Dora"

> x[[3]]
[1] "John" "Dora"
```

Accessing with `[[` or `$` is similar. However, it differs for `[` in that, indexing with `[` will return us a data frame but the other two will reduce it into a vector.

Accessing like a matrix

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

To illustrate this, we use datasets already available in R. Datasets that are available can be listed with the command `library(help = "datasets")`.

We will use the `trees` dataset which contains `Girth`, `Height` and `Volume` for Black Cherry Trees.

A data frame can be examined using functions like `str()` and `head()`.

```
> str(trees)
```

```
'data.frame':   31 obs. of 3 variables:
$ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
$ Height: num  70 65 63 72 81 83 66 75 80 75 ...
$ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...

> head(trees,n=3)
 Girth Height Volume
1   8.3      70   10.3
2   8.6      65   10.3
3   8.8      63   10.2
```

We can see that `trees` is a data frame with 31 rows and 3 columns. We also display the first 3 rows of the data frame.

Now we proceed to access the data frame like a matrix.

```
> trees[2:3,]     # select 2nd and 3rd row
 Girth Height Volume
2   8.6      65   10.3
3   8.8      63   10.2

> trees[trees$Height > 82,]    # selects rows with Height greater
than 82
  Girth Height Volume
6   10.8      83   19.7
17  12.9      85   33.8
18  13.3      86   27.4
31  20.6      87   77.0

> trees[10:12,2]
[1] 75 79 76
```

We can see in the last case that the returned type is a vector since we extracted data from a single column.

This behavior can be avoided by passing the argument `drop=FALSE` as follows.

```
> trees[10:12,2, drop = FALSE]
  Height
10     75
```

```
11      79
12      76
```

How to modify a Data Frame in R?

Data frames can be modified like we modified matrices through reassignment.

```
> x
 SN Age Name
1  1  21 John
2  2  15 Dora

> x[1,"Age"] <- 20; x
 SN Age Name
1  1  20 John
2  2  15 Dora
```

Adding Components

Rows can be added to a data frame using the `rbind()` function.

```
> rbind(x,list(1,16,"Paul"))
 SN Age Name
1  1  20 John
2  2  15 Dora
3  1  16 Paul
```

Similarly, we can add columns using `cbind()`.

```
> cbind(x,State=c("NY","FL"))
 SN Age Name State
1  1  20 John     NY
2  2  15 Dora     FL
```

Since data frames are implemented as list, we can also add new columns through simple list-like assignments.

```
> x
 SN Age Name
1  1   20 John
2  2   15 Dora

> x$State <- c("NY","FL"); x
 SN Age Name State
1  1   20 John     NY
2  2   15 Dora     FL
```

Deleting Component

Data frame columns can be deleted by assigning `NULL` to it.

```
> x$State <- NULL
> x
 SN Age Name
1  1   20 John
2  2   15 Dora
```

Similarly, rows can be deleted through reassignments.

```
> x <- x[-1,]
> x
 SN Age Name
2  2   15 Dora
```

R Factors

Factor is a data structure used for fields that takes only predefined, finite number of values (categorical data).

For example, a data field such as marital status may contain only values from single, married, separated, divorced, or widowed.

In such case, we know the possible values beforehand and these predefined, distinct values are called levels. Following is an example of factor in R.

```
> x
```

```
[1] single  married married single
Levels: married single
```

Here, we can see that factor `x` has four elements and two levels. We can check if a variable is a factor or not using `class()` function.

Similarly, levels of a factor can be checked using the `levels()` function.

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


> levels(x)
[1] "married" "single"
```

How to create a factor in R?

We can create a factor using the function `factor()`. Levels of a factor are inferred from the data if not provided.

```
> x <- factor(c("single", "married", "married", "single"));
> x
[1] single  married married single
Levels: married single


> x <- factor(c("single", "married", "married", "single"), levels =
c("single", "married", "divorced"));
> x
[1] single  married married single
Levels: single married divorced
```

We can see from the above 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
```

We see that levels are stored in a character vector and the individual elements are actually stored as indices.

Factors are also created when we read non-numerical columns into a data frame.

By default, `data.frame()` function converts character vector into factor. To suppress this behavior, we have to pass the argument `stringsAsFactors = FALSE`.

How to access compoments of a factor?

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

```
> x
[1] single  married married single
Levels: married single

> 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
```

How to modify a factor?

Components of a factor can be modified using simple assignments. However, we cannot choose values outside of its predefined levels.

```
> x
[1] single  married married single
Levels: single married divorced

> x[2] <- "divorced"    # modify second element;  x
[1] single   divorced married  single
Levels: single married divorced

> x[3] <- "widowed"    # cannot assign values outside levels
Warning message:
In `[<-.factor`(`*tmp*`, 3, value = "widowed") :
 invalid factor level, NA generated

> x
[1] single   divorced <NA>     single
Levels: single married divorced
```

A workaround to this is to add the value to the level first.

```
> levels(x) <- c(levels(x), "widowed")    # add new level

> x[3] <- "widowed"

> x
[1] single   divorced widowed  single
Levels: single married divorced widowed
```

R- Arrays

Arrays are the R data objects which can store data in more than two dimensions. For example - If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. Arrays can store only data type. An array is created using the **array()** function. It takes vectors as input and uses the values in the **dim** parameter to create an array.

`array(data=NA, dim=length(data), dimnames=NULL)`

`data`: vector to fill the array

`dim`: row and col numbers

:

```
...


> x <- array(1:9)
> x
[1] 1 2 3 4 5 6 7 8 9


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


> x <- 1:64
> dim(x) <- c(2,4,8) #dim() converts the vector
into array
> is.array(x)
[1] TRUE



> x
, , 1

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

, , 2

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

, , 3

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

```
, , 4

     [,1] [,2] [,3] [,4]
[1,]   25   27   29   31
[2,]   26   28   30   32

, , 5

     [,1] [,2] [,3] [,4]
[1,]   33   35   37   39
[2,]   34   36   38   40

, , 6

     [,1] [,2] [,3] [,4]
[1,]   41   43   45   47
[2,]   42   44   46   48

, , 7

     [,1] [,2] [,3] [,4]
[1,]   49   51   53   55
[2,]   50   52   54   56

, , 8

     [,1] [,2] [,3] [,4]
[1,]   57   59   61   63
[2,]   58   60   62   64


> x[1,,]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    9   17   25   33   41   49   57
[2,]    3   11   19   27   35   43   51   59
[3,]    5   13   21   29   37   45   53   61
[4,]    7   15   23   31   39   47   55   63


> x[1,2,]
[1]  3 11 19 27 35 43 51 59
```

```
> x[1,2,1]
[1] 3
```

Example

The following example creates an array of two 3x3 matrices each with 3 rows and 3columns.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim=c(3,3,2))
print(result)
```

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

```
, , 1
[,1] [,2] [,3]
[1,] 5 10 13
[2,] 9 11 14
[3,] 3 12 15
, , 2
[,1] [,2] [,3]
[1,] 5 10 13
[2,] 9 11 14
[3,] 3 12 15
```

Naming Columns and Rows

We can give names to the rows, columns and matrices in the array by using the **dimnames** parameter.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")
# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim=c(3,3,2),dimnames =
list(column.names,row.names,matrix.names))
print(result)
```

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

```
, , Matrix1
ROW1 ROW2 ROW3
COL1 5 10 13
COL2 9 11 14
COL3 3 12 15
, , Matrix2
ROW1 ROW2 ROW3
COL1 5 10 13
COL2 9 11 14
COL3 3 12 15
```

Accessing Array Elements

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")

matrix.names <- c("Matrix1","Matrix2")
# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim=c(3,3,2),dimnames =
list(column.names,row.names,matrix.names))
# Print the third row of the second matrix of the array.
print(result[3,,2])
# Print the element in the 1st row and 3rd column of the
1st matrix.
print(result[1,3,1])
# Print the 2nd Matrix.
print(result[,,2])
```

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

```
ROW1 ROW2 ROW3
3 12 15
[1] 13
ROW1 ROW2 ROW3
COL1 5 10 13
COL2 9 11 14
COL3 3 12 15
```

Manipulating Array Elements

As array is made up matrices in multiple dimensions, the operations on elements of array are carried out by accessing elements of the matrices.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
# Take these vectors as input to the array.
array1 <- array(c(vector1,vector2),dim=c(3,3,2))
# Create two vectors of different lengths.

vector3 <- c(9,1,0)

vector4 <- c(6,0,11,3,14,1,2,6,9)
array2 <- array(c(vector1,vector2),dim=c(3,3,2))
# create matrices from these arrays.
matrix1 <- array1[,,2]
matrix2 <- array2[,,2]
# Add the matrices.
result <- matrix1+matrix2
print(result)
```

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

```
[,1] [,2] [,3]
[1,] 10 20 26
[2,] 18 22 28
[3,] 6 24 30
```

Calculations across Array Elements

We can do calculations across the elements in an array using the **apply()**function.

Syntax

`apply(x, margin, fun)`

Following is the description of the parameters used:

- **x** is an array.
- **margin** is the name of the data set used.
- **fun** is the function to be applied across the elements of the array.

Example

We use the apply() function below to calculate the sum of the elements in the rows of an array across all the matrices.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
# Take these vectors as input to the array.

new.array <- array(c(vector1,vector2),dim=c(3,3,2))
```

```
print(new.array)
# Use apply to calculate the sum of the rows across all
the matrices.
result <- apply(new.array, c(1), sum)
print(result)
```

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

```
, , 1
[,1] [,2] [,3]
[1,] 5 10 13
[2,] 9 11 14
[3,] 3 12 15
, , 2
[,1] [,2] [,3]
[1,] 5 10 13
[2,] 9 11 14
[3,] 3 12 15

[1] 56 68 60
```

You can create an array easily with the **array()** function, where you give the data as the first argument and a vector with the sizes of the dimensions as the second argument. The number of dimension sizes in that argument gives you the number of dimensions. For example, you make an array with four columns, three rows, and two “tables” like this:

```
> my.array <- array(1:24, dim=c(3,4,2))
> my.array
, , 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
```

This array has three dimensions. Notice that, although the rows are given as the first dimension, the tables are filled column-wise. So, for arrays, R fills the columns, then the rows, and then the rest.

CHANGE THE DIMENSIONS OF A VECTOR IN R

Alternatively, you could just add the dimensions using the dim() function. This is a little hack that goes a bit faster than using the array() function; it's especially useful if you have your data already in a vector. (This little trick also works for creating matrices, by the way, because a matrix is nothing more than an array with only two dimensions.)

Say you already have a vector with the numbers 1 through 24, like this:

```
> my.vector <- 1:24
```

You can easily convert that vector to an array exactly like my.array simply by assigning the dimensions, like this:

```
> dim(my.vector) <- c(3,4,2)
```

If you check how my.vector looks like now, you see there is no difference from the array my.array that you created before.

```
> identical(my.array, my.vector)
[1] TRUE
```

The factorial of a number is the product of all the integers from 1 to that number.

For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1.

This example finds the factorial of a number normally. However, you can find it using recursion as well.

Example: Find the factorial of a number

```
# take input from the user
num = as.integer(readline(prompt="Enter a number: "))
factorial = 1
```

```
# check is the number is negative, positive or zero
if(num < 0) {
    print("Sorry, factorial does not exist for negative numbers")
} else if(num == 0) {
    print("The factorial of 0 is 1")
} else {
    for(i in 1:num) {
        factorial = factorial * i
    }
    print(paste("The factorial of", num ,"is",factorial))
}
```

output

```
Enter a number: 8

[1] "The factorial of 8 is 40320"
```

Here, we take input from the user and check if the number is negative, zero or positive using `if...else` statement.

If the number is positive, we use `for` loop to calculate the factorial.

We can also use the built-in function `factorial()` for this.

```
> factorial(8)
[1] 40320
```

R Functions

Functions are used to logically break our code into simpler parts which become easy to maintain and understand.

It's pretty straightforward to create your own function in R programming.

Syntax for Writing Functions in R

```
func_name <- function (argument) {

   statement

}
```

Here, we can see that the reserved word **`function`** is used to declare a function in R.

The statements within the curly braces form the body of the function. These braces are optional if the body contains only a single expression.

Finally, this function object is given a name by assigning it to a variable, `func_name`.

Example of a Function

```
pow <- function(x, y) {
   # function to print x raised to the power y

   result <- x^y
   print(paste(x,"raised to the power", y, "is", result))
}
```

Here, we created a function called `pow()`.

It takes two arguments, finds the first argument raised to the power of second argument and prints the result in appropriate format.

We have used a built-in function `paste()` which is used to concatenate strings.

How to call a function?

We can call the above function as follows.

```
>pow(8, 2)
[1] "8 raised to the power 2 is 64"

> pow(2, 8)
[1] "2 raised to the power 8 is 256"
```

Here, the arguments used in the **function declaration** (`x` and `y`) are called ***formal arguments*** and those used while **calling the function** are called ***actual arguments***.

Named Arguments

In the above function calls, the argument matching of formal argument to the actual arguments takes place in positional order.

This means that, in the call `pow(8,2)`, the formal arguments `x` and `y` are assigned 8 and 2 respectively.

We can also call the function using named arguments.

When calling a function in this way, the order of the actual arguments doesn't matter. For example, all of the function calls given below are equivalent.

```
> pow(8, 2)
[1] "8 raised to the power 2 is 64"

> pow(x = 8, y = 2)
[1] "8 raised to the power 2 is 64"

> pow(y = 2, x = 8)
[1] "8 raised to the power 2 is 64"
```

Furthermore, we can use named and unnamed arguments in a single call.

In such case, all the named arguments are matched first and then the remaining unnamed arguments are matched in a positional order.

```
> pow(x=8, 2)
[1] "8 raised to the power 2 is 64"

> pow(2, x=8)
[1] "8 raised to the power 2 is 64"
```

In all the examples above, `x` gets the value 8 and `y` gets the value 2.

Default Values for Arguments

We can assign default values to arguments in a function in R.

This is done by providing an appropriate value to the formal argument in the function declaration.

Here is the above function with a default value for `y`.

```
pow <- function(x, y = 2) {
   # function to print x raised to the power y
   result <- x^y
   print(paste(x,"raised to the power", y, "is", result))
}
```

The use of default value to an argument makes it optional when calling the function.

```
> pow(3)
[1] "3 raised to the power 2 is 9"

> pow(3,1)
[1] "3 raised to the power 1 is 3"
```

Here, `y` is optional and will take the value 2 when not provided.

R Return Value from Function

In this ,to return a value from a function in R. You'll also learn to use functions without the return function.

Many a times, we will require our functions to do some processing and return back the result. This is accomplished with the `return()` function in R.

Syntax of return()

```
return(expression)
```

The value returned from a function can be any valid object.

Example: return()

Let us look at an example which will return whether a given number is positive, negative or zero.

```
check <- function(x) {
   if (x > 0) {
       result <- "Positive"
   }
   else if (x < 0) {
       result <- "Negative"
   }
   else {
       result <- "Zero"
   }
   return(result)
}
```

Here, are some sample runs.

```
> check(1)

[1] "Positive"



> check(-10)

[1] "Negative"



> check(0)

[1] "Zero"
```

Functions without return()

If there are no explicit returns from a function, the value of the last evaluated expression is returned automatically in R.

For example, the following is equivalent to the above function.

```
check <- function(x) {
   if (x > 0) {
       result <- "Positive"
   }
   else if (x < 0) {
       result <- "Negative"
   }
   else {
       result <- "Zero"
   }
   result
}
```

We generally use explicit `return()` functions to return a value immediately from a function.

If it is not the last statement of the function, it will prematurely end the function bringing the control to the place from which it was called.

```
check <- function(x) {
   if (x>0) {
       return("Positive")
   }
   else if (x<0) {
       return("Negative")
   }
   else {
       return("Zero")
   }
}
```

In the above example, if `x > 0`, the function immediately returns `"Positive"` without evaluating rest of the body.

Multiple Returns

The `return()` function can return ***only a single object***. If we want to return multiple values in R, we can use a list (or other objects) and return it.

Following is an example.

```
multi_return <- function() {
   my_list <- list("color" = "red", "size" = 20, "shape" = "round")
   return(my_list)
}
```

Here, we create a list `my_list` with multiple elements and return this single list.

```
> a <- multi_return()

> a$color

[1] "red"



> a$size

[1] 20



> a$shape

[1] "round"
```

Lazy Evaluation of Function

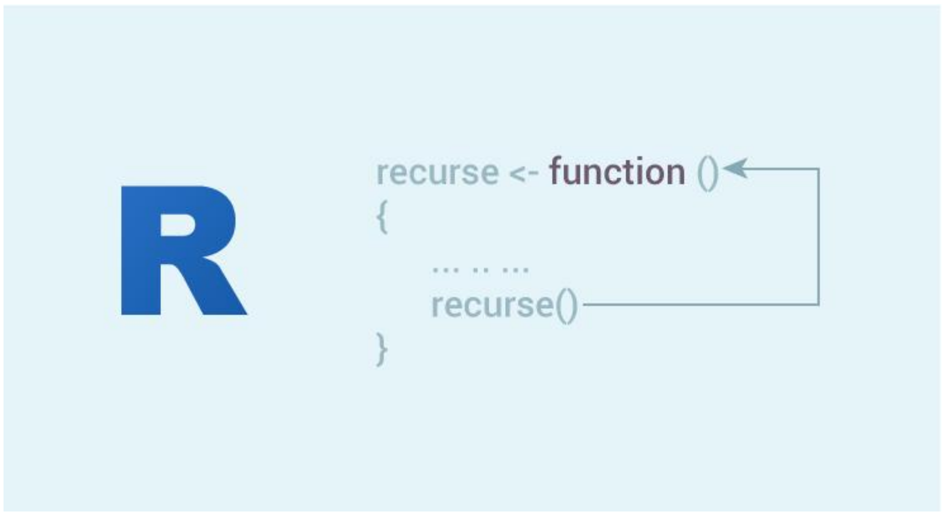
Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# 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
```

R Recursive Function



A function that calls itself is called a recursive function and this technique is known as recursion.

This special programming technique can be used to solve problems by breaking them into smaller and simpler sub-problems.

An example can help clarify this concept:

Let us take the example of finding the factorial of a number. Factorial of a positive integer number is defined as the product of all the integers from 1 to that number. For example, the factorial of 5 (denoted as `5!`) will be

```
5! = 1*2*3*4*5 = 120
```

This problem of finding factorial of 5 can be broken down into a sub-problem of multiplying the factorial of 4 with 5.

```
5! = 5*4!
```

Or more generally,

```
n! = n*(n-1)!
```

Now we can continue this until we reach `0!` which is `1`.

The implementation of this is provided below.

Example: Recursive Function in R

```r
# Recursive function to find factorial

recursive.factorial <- function(x) {
   if (x == 0)     return (1)
   else            return (x * recursive.factorial(x-1))
}
```

Here, we have a function which will call itself. Something like `recursive.factorial(x)` will turn into `x * recursive.factorial(x)` until `x` becomes equal to `0`.

When `x` becomes `0`, we return `1` since the factorial of `0` is `1`. This is the terminating condition and is very important.

Without this the recursion will not end and continue indefinitely (in theory). Here are some sample function calls to our function.

```
> recursive.factorial(0)
[1] 1

> recursive.factorial(5)
[1] 120

> recursive.factorial(7)
[1] 5040
```

The use of recursion, often, makes code shorter and looks clean.

However, it is sometimes hard to follow through the code logic. It might be hard to think of a problem in a recursive way.

Recursive functions are also memory intensive, since it can result into a lot of nested function calls. This must be kept in mind when using it for solving big problems.

R Objects and Classes: Introduction and Types

R has 3 classes. In this you'll be introduced to all three classes (S3, S4 and reference class) in R programming.

We can do object oriented programming in R. In fact, everything in R is an object.

An object is a data structure having some attributes and methods which act on its attributes.

Class is a blueprint for the object. We can think of class like a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house.

House is the object. As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called instantiation.

While most programming languages have a single class system, R has three class systems. Namely, S3, S4 and more recently Reference class systems.

They have their own features and peculiarities and choosing one over the other is a matter of preference. Below, we give a brief introduction to them.

S3 Class

S3 class is somewhat primitive in nature. It lacks a formal definition and object of this class can be created simply by adding a class attribute to it.

This simplicity accounts for the fact that it is widely used in R programming language. In fact most of the R built-in classes are of this type.

Example 1: S3 class

```
> # create a list with required components
> s <- list(name = "John", age = 21, GPA = 3.5)


> # name the class appropriately
> class(s) <- "student"
```

Above example creates a S3 class with the given list.

S4 Class

S4 class are an improvement over the S3 class. They have a formally defined structure which helps in making object of the same class look more or less similar.

Class components are properly defined using the `setClass()` function and objects are created using the `new()` function.

Example 2: S4 class

```
< setClass("student", slots=list(name="character", age="numeric",
GPA="numeric"))
```

Reference Class

Reference class were introduced later, compared to the other two. It is more similar to the object oriented programming we are used to seeing in other major programming languages.

Reference classes are basically S4 classed with an environment added to it.

Example 3: Reference class

```
< setRefClass("student")
```

Comparision between S3 vs S4 vs Reference Class

S3 Class	S4 Class	Referene Class
Lacks formal definition	Class defined using `setClass()`	Class defined using `setRefClass()`
Objects are created by setting the class attribute	Objects are created using `new()`	Objects are created using generator functions
Attributes are accessed using `$`	Attributes are accessed using `@`	Attributes are accessed using `$`
Methods belong to generic function	Methods belong to generic function	Methods belong to the class

Follows copy-on-modify semantics	Follows copy-on-modify semantics	Does not follow copy-on-modify semantics

R S3 Class

In this article, you will learn to work with S3 classes (one of the three class systems in R).

S3 class is the most popular and prevalent class in R programming language.

Most of the classes that come predefined in R are of this type. The fact that it is simple and easy to implement is the reason behind this.

How to define S3 class and create S3 objects?

S3 class has no formal, predefined definition.

Basically, a list with its class attribute set to some class name, is an S3 object. The components of the list become the member variables of the object.

Following is a simple example of how an S3 object of class student can be created.

```
> # create a list with required components
> s <- list(name = "John", age = 21, GPA = 3.5)

> # name the class appropriately
> class(s) <- "student"

> # That's it! we now have an object of class "student"
> s
$name
[1] "John"
```

```
$age
[1] 21


$GPA
[1] 3.5


attr(,"class")
[1] "student"
```

This might look awkward for programmers coming from C++, Python etc. where there are formal class definitions and objects have properly defined attributes and methods.

In R S3 system, it's pretty ad hoc. You can convert an object's class according to your will with objects of the same class looking completely different. It's all up to you.

How to use constructors to create objects?

It is a good practice to use a function with the same name as class (not a necessity) to create objects.

This will bring some uniformity in the creation of objects and make them look similar.

We can also add some integrity check on the member attributes. Here is an example. Note that in this example we use the `attr()` function to set the class attribute of the object.

```
# a constructor function for the "student" class


student <- function(n,a,g) {
 # we can add our own integrity checks
 if(g>4 || g<0)  stop("GPA must be between 0 and 4")
 value <- list(name = n, age = a, GPA = g)


 # class can be set using class() or attr() function
 attr(value, "class") <- "student"
 value
}
```

Here is a sample run where we create objects using this constructor.

```
> s <- student("Paul", 26, 3.7)

> s
$name
[1] "Paul"

$age
[1] 26

$GPA
[1] 3.7

attr(,"class")
[1] "student"

> class(s)
[1] "student"

> s <- student("Paul", 26, 5)
Error in student("Paul", 26, 5) : GPA must be between 0 and 4

> # these integrity check only work while creating the object using
constructor
> s <- student("Paul", 26, 2.5)

> # it's up to us to maintain it or not
> s$GPA <- 5
```

Methods and Generic Functions

In the above example, when we simply write the name of the object, its internals get printed.

In interactive mode, writing the name alone will print it using the `print()` function.

```
> s
$name
[1] "Paul"


$age
[1] 26


$GPA
[1] 3.7


attr(,"class")
[1] "student"
```

Furthermore, we can use `print()` with vectors, matrix, data frames, factors etc. and they get printed differently according to the class they belong to.

How does `print()` know how to print these variety of dissimilar looking object?

The answer is, `print()` is a generic function. Actually, it has a collection of a number of methods. You can check all these methods with `methods(print)`.

```
> methods(print)
  [1] print.acf*
  [2] print.anova*
...
[181] print.xngettext*
[182] print.xtabs*


   Non-visible functions are asterisked
```

We can see methods like `print.data.frame` and `print.factor` in the above list.

When we call `print()` on a data frame, it is dispatched to `print.data.frame()`.

If we had done the same with a factor, the call would dispatch to `print.factor()`. Here, we can observe that the method names are in the form `generic_name.class_name()`. This is how R is able to figure out which method to call depending on the class.

Printing our object of class "`student`" looks for a method of the form `print.student()`, but there is no method of this form.

So, which method did our object of class "`student`" call?

It called `print.default()`. This is the fallback method which is called if no other match is found. Generic functions have a default method.

There are plenty of generic functions like `print()`. You can list them all with `methods(class="default")`.

```
> methods(class="default")
 [1] add1.default*            aggregate.default*
 [3] AIC.default*             all.equal.default
...
```

How to write your own method?

Now let us implement a method `print.student()` ourself.

```
print.student <- function(obj) {
 cat(obj$name, "\n")
 cat(obj$age, "years old\n")
 cat("GPA:", obj$GPA, "\n")
}
```

Now this method will be called whenever we `print()` an object of class "`student`".

In S3 system, methods do not belong to object or class, they belong to generic functions. This will work as long as the class of the object is set.

```
> # our above implemented method is called
> s
Paul
26 years old
GPA: 3.7


> # removing the class attribute will restore as previous
> unclass(s)
$name
[1] "Paul"
```

```
$age
[1] 26


$GPA
[1] 3.7
```

Writing Your Own Generic Function

It is possible to make our own generic function like `print()` or `plot()`. Let us first look at how these functions are implemented.

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x0674e230>
<environment: namespace:base>


> plot
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x04fe6574>
<environment: namespace:graphics>
```

We can see that they have a single call to `UseMethod()` with the name of the generic function passed to it. This is the dispatcher function which will handle all the background details. It is this simple to implement a generic function.

For the sake of example, we make a new generic function called `grade`.

```
grade <- function(obj) {
 UseMethod("grade")
}
```

A generic function is useless without any method. Let us implement the default method.

```
grade.default <- function(obj) {
 cat("This is a generic function\n")
}
```

Now let us make method for our class "`student`".

```
grade.student <- function(obj) {
 cat("Your grade is", obj$GPA, "\n")
}
```

A sample run.

```
> grade(s)
Your grade is 3.7
```

In this way, we implemented a generic function called `grade` and later a method for our class.

R S4 Class

In this ,you'll learn everything about S4 classes in R; how to define them, create them, access their slots, and use them efficiently in your program.

Unlike S3 classes and objects which lacks formal definition, we look at S4 class which is stricter in the sense that it has a formal definition and a uniform way to create objects.

This adds safety to our code and prevents us from accidentally making naive mistakes.

How to define S4 Class?

S4 class is defined using the `setClass()` function.

In R terminology, member variables are called slots. While defining a class, we need to set the name and the slots (along with class of the slot) it is going to have.

Example 1: Definition of S4 class

```
setClass("student", slots=list(name="character", age="numeric",
GPA="numeric"))
```

In the above example, we defined a new class called `student` along with three slots it's going to have `name`, `age` and `GPA`.

There are other optional arguments of `setClass()` which you can explore in the help section with `?setClass`.

How to create S4 objects?

S4 objects are created using the `new()` function.

Example 2: Creation of S4 object

```
> # create an object using new()
> # provide the class name and value for slots
> s <- new("student",name="John", age=21, GPA=3.5)

> s
An object of class "student"
Slot "name":
[1] "John"

Slot "age":
[1] 21

Slot "GPA":
[1] 3.5
```

We can check if an object is an S4 object through the function `isS4()`.

```
> isS4(s)
[1] TRUE
```

The function `setClass()` returns a generator function.

This generator function (usually having same name as the class) can be used to create new objects. It acts as a constructor.

```
> student <- setClass("student", slots=list(name="character",
age="numeric", GPA="numeric"))


> student

class generator function for class “student” from package
‘.GlobalEnv’

function (...)

new("student", ...)
```

Now we can use this constructor function to create new objects.

Note above that our constructor in turn uses the `new()` function to create objects. It is just a wrap around.

Example 3: Creation of S4 objects using generator function

```
> student(name="John", age=21, GPA=3.5)

An object of class "student"

Slot "name":

[1] "John"


Slot "age":

[1] 21


Slot "GPA":

[1] 3.5
```

How to access and modify slot?

Just as components of a list are accessed using `$`, slot of an object are accessed using `@`.

Accessing slot

```
> s@name

[1] "John"


> s@GPA

[1] 3.5
```

```
> s@age
[1] 21
```

Modifying slot directly

A slot can be modified through reassignment.

```
> # modify GPA
> s@GPA <- 3.7

> s
An object of class "student"
Slot "name":
[1] "John"

Slot "age":
[1] 21

Slot "GPA":
[1] 3.7
```

Modifying slots using slot() function

Similarly, slots can be access or modified using the `slot()` function.

```
> slot(s,"name")
[1] "John"

> slot(s,"name") <- "Paul"

> s
An object of class "student"
Slot "name":
[1] "Paul"

Slot "age":
[1] 21
```

```
Slot "GPA":
[1] 3.7
```

Methods and Generic Functions

As in the case of S3 class, methods for S4 class also belong to generic functions rather than the class itself. Working with S4 generics is pretty much similar to S3 generics.

You can list all the S4 generic functions and methods available, using the function `showMethods()`.

Example 4: List all generic functions

```
> showMethods()
Function: - (package base)


Function: != (package base)
...
Function: trigamma (package base)


Function: trunc (package base)
```

Writing the name of the object in interactive mode prints it. This is done using the S4 generic function `show()`.

You can see this function in the above list. This function is the S4 analogy of the S3 `print()`function.

Example 5: Check if a function is a generic function

```
> isS4(print)
[1] FALSE


> isS4(show)
[1] TRUE
```

We can list all the methods of show generic function using `showMethods(show)`.

Example 6: List all methods of a generic function

```
> showMethods(show)
Function: show (package methods)
object="ANY"
```

```
object="classGeneratorFunction"
...
object="standardGeneric"
(inherited from: object="genericFunction")
object="traceable"
```

How to write your own method?

We can write our own method using `setMethod()` helper function.

For example, we can implement our class method for the `show()` generic as follows.

```
setMethod("show",
         "student",
         function(object) {
           cat(object@name, "\n")
           cat(object@age, "years old\n")
           cat("GPA:", object@GPA, "\n")
         }
)
```

Now, if we write out the name of the object in interactive mode as before, the above code is executed.

```
> s <- new("student",name="John", age=21, GPA=3.5)

> s    # this is same as show(s)
John
21 years old
GPA: 3.5
```

In this way we can write our own S4 class methods for generic functions.

R Reference Class

In this, you will learn to work with reference class which is one of the three class systems (other two are S3 and S4).

Reference class in R programming is similar to the object oriented programming we are used to seeing in common languages like C++, Java, Python etc.

Unlike S3 and S4 classes, methods belong to class rather than generic functions. Reference class are internally implemented as S4 classes with an environment added to it.

How to defined a reference class?

Defining reference class is similar to defining a S4 class. Instead of `setClass()` we use the `setRefClass()` function.

```
> setRefClass("student")
```

Member variables of a class, if defined, need to be included in the class definition. Member variables of reference class are called fields (analogous to slots in S4 classes).

Following is an example to define a class called `student` with 3 fields, `name`, `age` and `GPA`.

```
> setRefClass("student", fields = list(name = "character", age =
"numeric", GPA = "numeric"))
```

How to create a reference objects?

The function `setRefClass()` returns a generator function which is used to create objects of that class.

```
> student <- setRefClass("student",

    fields = list(name = "character", age = "numeric", GPA =
"numeric"))
```

```
> # now student() is our generator function which can be used to
create new objects


> s <- student(name = "John", age = 21, GPA = 3.5)


> s
Reference class object of class "student"
Field "name":
[1] "John"
Field "age":
[1] 21
Field "GPA":
[1] 3.5
```

How to access and modify fields?

Fields of the object can be accessed using the `$` operator.

```
> s$name
[1] "John"


> s$age
[1] 21


> s$GPA
[1] 3.5
```

Similarly, it is modified by reassignment.

```
> s$name <- "Paul"


> s
Reference class object of class "student"
Field "name":
[1] "Paul"
Field "age":
```

```
[1] 21
Field "GPA":
[1] 3.5
```

Warning Note:

In R programming, objects are copied when assigned to new variable or passed to a function (pass by value). For example.

```
> # create list a and assign to b
> a <- list("x" = 1, "y" = 2)
> b <- a

> # modify b
> b$y = 3

> # a remains unaffected
> a
$x
[1] 1

$y
[1] 2

> # only b is modified
> b
$x
[1] 1

$y
[1] 3
```

But this is not the case with reference objects. Only a single copy exist and all variables reference to the same copy. Hence the name, reference.

```
> # create reference object a and assign to b
> a <- student(name = "John", age = 21, GPA = 3.5)
> b <- a
> # modify b
> b$name <- "Paul"

> # a and b both are modified
> a
Reference class object of class "student"
Field "name":
[1] "Paul"
Field "age":
[1] 21
Field "GPA":
[1] 3.5

> b
Reference class object of class "student"
Field "name":
[1] "Paul"
Field "age":
[1] 21
Field "GPA":
[1] 3.5
```

This can cause some unwanted change in values and be the source of strange bugs. We need to keep this in mind while working with reference objects. To make a copy, we can use the `copy()` method made availabe to us.

```
> # create reference object a and assign a’s copy to b
> a <- student(name = "John", age = 21, GPA = 3.5)
> b <- a$copy()

> # modify b
> b$name <- "Paul"

> # a remains unaffected
> a
Reference class object of class "student"
Field "name":
```

```
[1] "John"
Field "age":
[1] 21
Field "GPA":
[1] 3.5

> # only b is modified
> b
Reference class object of class "student"
Field "name":
[1] "Paul"
Field "age":
[1] 21
Field "GPA":
[1] 3.5
```

Reference Methods

Methods are defined for a reference class and do not belong to generic functions as in S3 and S4 classes.

All reference class have some methods predefined because they all are inherited from the superclass `envRefClass`.

```
> student
Generator for class "student":

Class fields:

Name:       name       age       GPA
Class: character   numeric   numeric

 Class Methods:
    "callSuper", "copy", "export", "field", "getClass",
"getRefClass",
"import", "initFields", "show", "trace", "untrace", "usingMethods"
```

```
Reference Superclasses:
   "envRefClass"
```

We can see class methods like `copy()`, `field()` and `show()` in the above list. We can create our own methods for the class.

This can be done during the class definition by passing a list of function definitions to `methods` argument of `setRefClass()`.

```
student <- setRefClass("student",
  fields = list(name = "character", age = "numeric", GPA = 
"numeric"),
  methods = list(
    inc_age = function(x) {
      age <<- age + x
    },
    dec_age = function(x) {
      age <<- age - x
    }
  )
)
```

In the above section of our code, we defined two methods called `inc_age()` and `dec_age()`. These two method modify the field `age`.

Note that we have to use the non-local assignment operator `<<-` since age isn't in the method's local environment. This is important.

Using the simple assignment operator `<-` would have created a local variable called `age`, which is not what we want. R will issue a warning in such case.

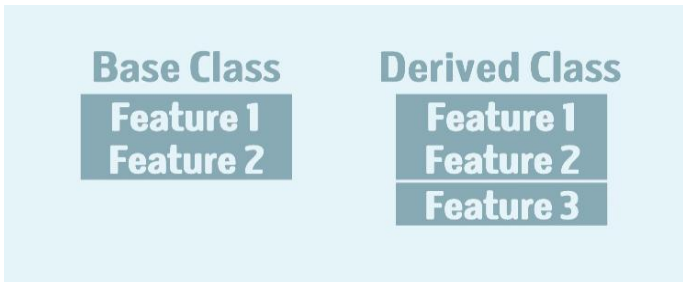
Here is a sample run where we use the above defined methods.

```
> s <- student(name = "John", age = 21, GPA = 3.5)


> s$inc_age(5)
> s$age
[1] 26


> s$dec_age(10)
> s$age
```

```
[1] 16
```

R Inheritance

In this, you'll learn everything about inheritance in R. More specifically, how to create inheritance in S3, S4 and Reference classes, and use them efficiently in your program.

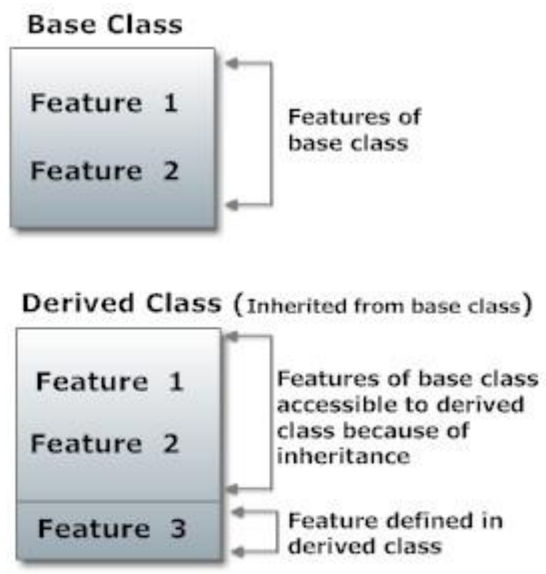


One the most useful feature of an object oriented programming language is inheritance.

Defining new classes out of existing ones. This is to say, we can derive new classes from existing base classes and adding new features. We don't have to write from scratch. Hence, inheritance provides reusability of code.

Inheritance forms a hierarchy of class just like a family tree. Important thing to note is that the attributes define for a base class will automatically be present in the derived class.

Moreover, the methods for the base class will work for the derived.


Base Class
Feature 1
Feature 2
Features of base class
Derived Class (Inherited from base class)
Feature 1
Feature 2
Feature 3
Features of base class accessible to derived class because of inheritance
Feature defined in derived class

Below, we discuss how inheritance is carried out for the three different class systems in R programming language.

Inheritance in S3 Class

S3 classes do not have any fixed definition. Hence attributes of S3 objects can be arbitrary.

Derived class, however, inherit the methods defined for base class. Let us suppose we have a function that creates new objects of class `student` as follows.

```
student <- function(n,a,g) {
  value <- list(name=n, age=a, GPA=g)
  attr(value, "class") <- "student"
  value
}
```

Furthermore, we have a method defined for generic function `print()` as follows.

```
print.student <- function(obj) {
  cat(obj$name, "\n")
  cat(obj$age, "years old\n")
  cat("GPA:", obj$GPA, "\n")
}
```

Now we want to create an object of class `InternationalStudent` which inherits from `student`.

This is be done by assigning a character vector of class names like `class(obj) <- c(child, parent)`.

```
> # create a list
> s <- list(name="John", age=21, GPA=3.5, country="France")

> # make it of the class InternationalStudent which is derived from the class student
> class(s) <- c("InternationalStudent","student")

> # print it out
> s
John
21 years old
GPA: 3.5
```

We can see above that, since we haven't defined any method of the form `print.InternationalStudent()`, the method `print.student()` got called. This method of class `student` was inherited.

Now let us define `print.InternationalStudent()`.

```
print.InternationalStudent <- function(obj) {
  cat(obj$name, "is from", obj$country, "\n")
}
```

This will overwrite the method defined for class `student` as shown below.

```
> s
John is from France
```

We can check for inheritance with functions like `inherits()` or `is()`.

```
> inherits(s,"student")
[1] TRUE

> is(s,"student")
[1] TRUE
```

Inheritance in S4 Class

Since S4 classes have proper definition, derived classes will inherit both attributes and methods of the parent class.

Let us define a class `student` with a method for the generic function `show()`.

```
# define a class called student
setClass("student",
  slots=list(name="character", age="numeric", GPA="numeric")
)

# define class method for the show() generic function
setMethod("show",
  "student",
  function(object) {
    cat(object@name, "\n")
    cat(object@age, "years old\n")
    cat("GPA:", object@GPA, "\n")
  }
)
```

Inheritance is done during the derived class definition with the argument `contains` as shown below.

```
# inherit from student
setClass("InternationalStudent",
  slots=list(country="character"),
  contains="student"
)
```

Here we have added an attribute `country`, rest will be inherited from the parent.

```
> s <- new("InternationalStudent",name="John", age=21, GPA=3.5, country="France")


> show(s)
John
21 years old
GPA: 3.5
```

We see that method define for class `student` got called when we did `show(s)`.

We can define methods for the derived class which will overwrite methods of the base class, like in the case of S3 systems.

Inheritance in Reference Class

Inheritance in reference class is very much similar to that of the S4 class. We define in the `contains` argument, from which base class to derive from.

Here is an example of `student` reference class with two methods `inc_age()` and `dec_age()`.

```
student <- setRefClass("student",
  fields=list(name="character", age="numeric", GPA="numeric"),
  methods=list(
    inc_age = function(x) {
      age <<- age + x
    },
    dec_age = function(x) {
      age <<- age - x
    }
  )
)
```

Now we will inherit from this class. We also overwrite `dec_age()` method to add an integrity check to make sure `age` is never negative.

```
InternationalStudent <- setRefClass("InternationalStudent",
  fields=list(country="character"),
  contains="student",
  methods=list(
    dec_age = function(x) {
      if((age - x)<0)  stop("Age cannot be negative")
      age <<- age - x
    }
  )
)
```

Let us put it to test.

```
> s <- InternationalStudent(name="John", age=21, GPA=3.5,
country="France")


> s$dec_age(5)
> s$age
[1] 16


> s$dec_age(20)
Error in s$dec_age(20) : Age cannot be negative
> s$age
[1] 16
```

In this way, we are able to inherit from the parent class.