DATA ANALYTICS USING R **II B. TECH- II SEMESTER :**

Course Code	Category	Hours / Week			Credits	Maximum Marks		
A6DS02	PCC	L	T	P	C	CIE	SEE	Total
		2	-	-	2	40	60	100

Contact Classes-53 **Tutorial Classes-Nil** **Practical Classes-Nil** **Total Classes-53**

COURSE OBJECTIVES:

1. How to use R for analytical programming
2. Understanding different techniques of data analysis in R
3. Be exposed to different concepts of data stream in R
4. How to perform Regression and clustering
5. Understanding different visualization techniques in R programming

COURSE OUTCOMES:

At the end of the course, student will be able to

1. Explain critical R programming concepts
2. Demonstrate how to install and configure RStudio
3. Explain the use of data structure and loop functions
4. Apply different techniques to generate reports based on the data
5. Develop R applications using visualization for different real world problems

UNIT-I Introduction to R Classes: 12

Introduction to R-Features of R, Installation of R, Getting Started, Variables in R, Mathematical Operators and Vectors, Assigning Variables, Special Numbers, classes, different types of numbers, changing classes, examining variables, the workplace, library of package, getting to know a package. Input of Data, Output in R, In-Built Functions in R, Packages in R

UNIT-II Data Types & Functions Classes: 11

Data Types of R-Vectors, Matrices, Arrays, Lists, Factors, Data Frame-Basic Expressions Vectors – sequences, lengths, names, indexing vectors, vector recycling and repetition, Matrices and Arrays – creating arrays and matrices, rows, columns, dimensions, indexing arrays, Arithmetic. Decision-Making Structures – Loops, User- Defined Functions, User-Defined Package, Reports using Rmarkdown, Conditional – if and else, vectorized if,

multiple selection, Loops – repeat loops, while loops, for loops, Advanced looping – replication, looping over lists, looping over arrays, Multiple – input apply, instant vectorization.

UNIT-III Lists, Strings and Data Frames Classes: 10

Lists, Functions, Strings and Factors Lists – creating lists, automatic and recursive variables, list dimensions and arithmetic, indexing lists, conversion between vectors and lists, Combining lists, NULL, Pairlists, Data Frames – Creating Data Frames, Indexing Data Frames, Basic Data Frame Manipulation, Functions – Creating and Calling Functions, Passing functions, variable scope, Strings – Constructing and printing strings, Formatting numbers, Special characters, Changing case, Extracting Substrings.

UNIT -I

Introduction to R

Features of R

□R allows branching and looping as well as modular programming using functions.

□R allows integration with different programming languages like C, C++, .Net, Python etc.

□R has an extensive community of contributors.

□R has an effective data handling and storage facility for numeric and textual data.

□R provides a collection of operators for calculations on arrays, lists, factor, vectors, data frame and matrices.

□R provides large and integrated collection of tools for data analysis and statistical functions.

□R provides graphical facilities for data analysis and can show result both in soft and hard copies.

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

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"

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.

Installation of R

□R can be installed from R-3.2.2 for Windows (32/64bit) and save it in a local directory. In windows, installer (.exe) with a name "R-version-win.exe" will be downloaded. Double click and run the installer accepting the default settings. R is available for both the versions of windows (32-bit/ 64-bit.)

Windows and OS X users can download R from CRAN, the Comprehensive R Archive Network. Linux and Unix users can install R packages using their package management tool:

Windows

1. Open *http://www.r-project.org/* in your browser.
2. Click on “CRAN”. You’ll see a list of mirror sites, organized by country.
3. Select a site near you.
4. Click on “Windows” under “Download and Install R”.
5. Click on “base”.
6. Click on the link for downloading the latest version of R (an *.exe* file).
7. When the download completes, double-click on the *.exe* file and answer the usual questions.

OS X

1. Open *http://www.r-project.org/* in your browser.
2. Click on “CRAN”. You’ll see a list of mirror sites, organized by country.
3. Select a site near you.
4. Click on “MacOS X”.
5. Click on the *.pkg* file for the latest version of R, under “Files:”, to download it.
6. When the download completes, double-click on the *.pkg* file and answer the usual questions.

The major Linux distributions have packages for installing R. Here are some examples:

Distribution	Package name
Ubuntu or Debian	r-base
Red Hat or Fedora	R.i386
Suse	R-base

Use the system’s package manager to download and install the package. Normally, you will need the root password or sudo privileges; otherwise, ask a system administrator to perform the installation.

Getting Started

Windows
Click on Start → All Programs → R; or double-click on the R icon on your desktop (assuming the installer created an icon for you).
OS X
Either click on the icon in the *Applications* directory or put the R icon on the dock and click on the icon there. Alternatively, you can just type R on a Unix command line in a shell.
Linux or Unix
Start the R program from the shell prompt using the R command (uppercase R).

Starting on Windows
When you start R, it opens a new window. The window includes a text pane, called the R Console, where you enter R expressions

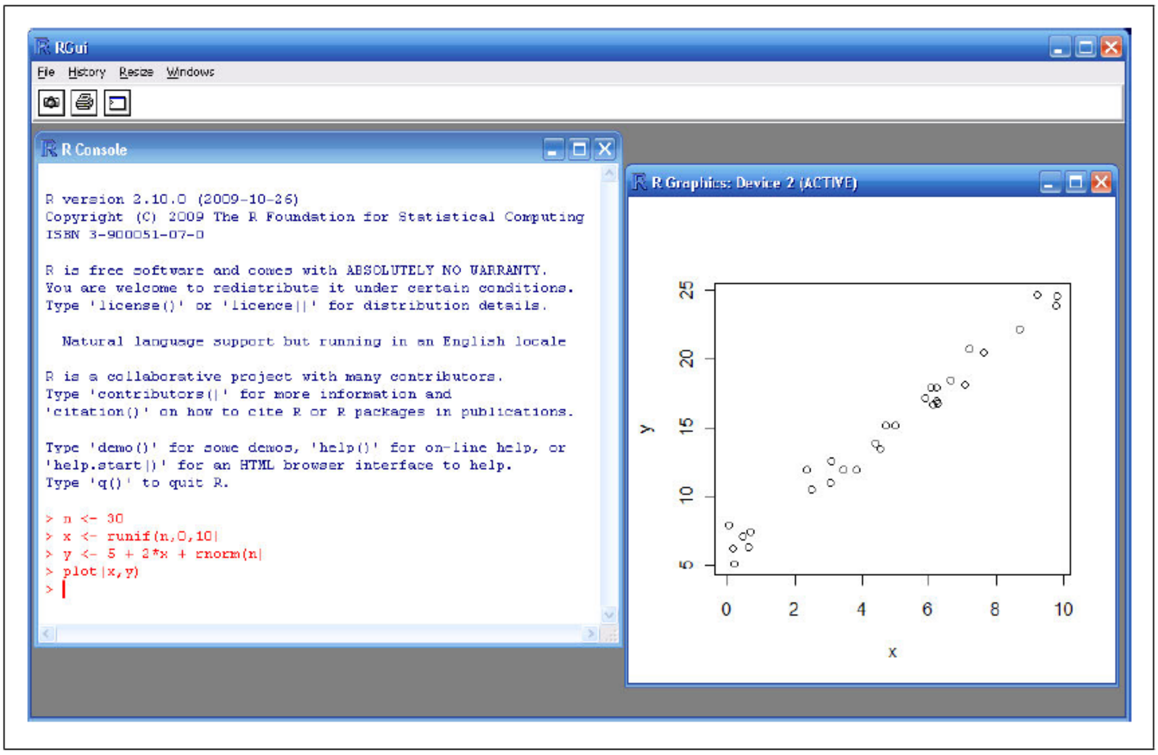
R Studio has four main window sections:

□Top-Left Section: To write and save R code (Script section)

□Bottom-Left Section: To execute R code and doing calculations. The nature and values of all variables and objects appear here (Console section)

□Top-Right Section: To manage datasets and variables (Data section)

□Bottom-Right Section: To display plots ,installed packages and seek help on R functions (Plot, Packages and Help Section)



```
R version 2.10.0 (2009-10-26)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

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.

> n <- 30
> x <- runif(n,0,10)
> y <- 5 + 2*x + rnorm(n)
> plot(x,y)
> 
```

There is an odd thing about the Windows Start menu for R. Every time you upgrade to a new version of R, the Start menu expands to contain the new version while keeping all the previously installed versions. So if you've upgraded, you may face several choices such as "R 2.8.1", "R 2.9.1", "R 2.10.1", and so forth. Pick the newest one. (You might also consider uninstalling the older versions to reduce the clutter.)

Another way to start R is by double-clicking on a *.RData* file in your working directory. This is the file that R creates to save your workspace. The first time you create a directory, start R and change to that directory. Save your workspace there, either by exiting or using the save.image function. That will create the *.RData* file. Thereafter, you can simply open the directory in Windows Explorer and then double-click on the *.RData* file to start R.

- If you start R from the Start menu, the working directory is normally either *C:\Documents and Settings\<username>\My Documents* (Windows XP) or *C:\Users \<username>\Documents* (Windows Vista, Windows 7). You can override this default by setting the R_USER environment variable to an alternative directory path.
- If you start R from a desktop shortcut, you can specify an alternative startup directory that becomes the working directory when R is started. To specify the alternative directory, right-click on the shortcut, select Properties, enter the directory path in the box labeled "Start in", and click OK.
- Starting R by double-clicking on your *.RData* file is the most straightforward solution to this little problem. R will automatically change its working directory to be the file's directory, which is usually what you want.

In any event, you can always use the getwd function to discover your current working directory (Recipe 3.1).
Just for the record, Windows also has a console version of R called *Rterm.exe*. You'll find it in the *bin* subdirectory of your R installation. It is much less convenient than the graphic user interface (GUI) version, and I never use it. I recommend it only for batch (noninteractive) usage such as running jobs from the Windows scheduler. In this book, I assume you are running the GUI version of R, not the console version.

R prompts you with ">". To get started, just treat R like a big calculator: enter an expression, and R will evaluate the expression and print the result:

```
> 1+1
[1] 2
```

The computer adds one and one, giving two, and displays the result.
The [1] before the 2 might be confusing. To R, the result is a vector, even though it has only one element. R labels the value with [1] to signify that this is the first element of the vector...which is not surprising, since it's the *only* element of the vector.
R will prompt you for input until you type a complete expression. The expression max(1,3,5) is a complete expression, so R stops reading input and evaluates what it's got:

```
> max(1,3,5)
[1] 5
```

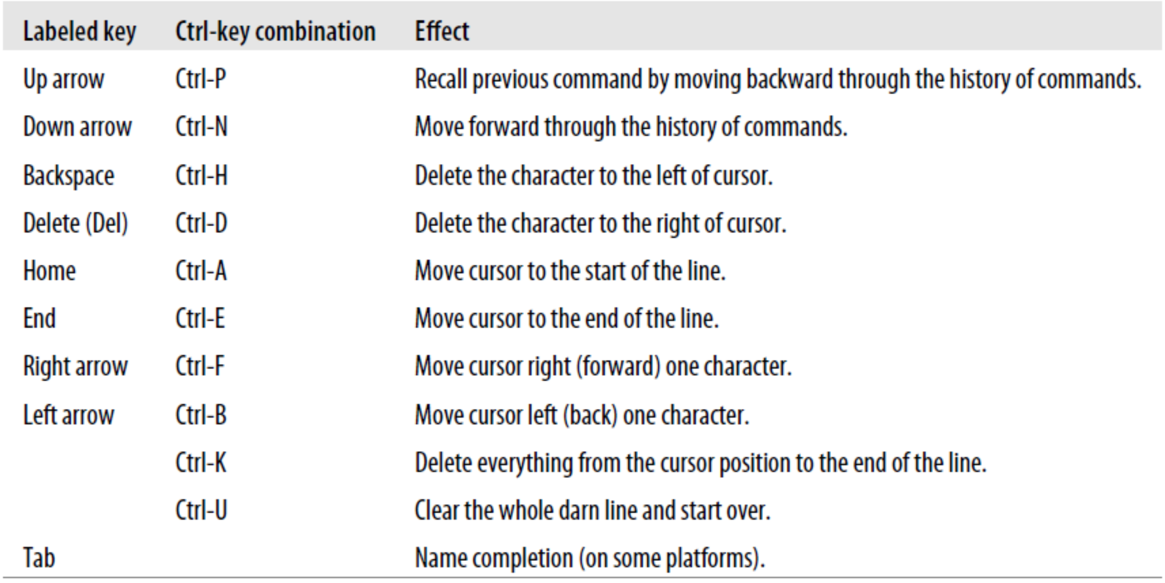
In contrast, "max(1,3," is an incomplete expression, so R prompts you for more input. The prompt changes from greater-than (>) to plus (+), letting you know that R expects more:

```
> max(1,3,
+ 5)
[1] 5
```

It’s easy to mistype commands, and retyping them is tedious and frustrating. So R includes command-line editing to make life easier. It defines single keystrokes that let you easily recall, correct, and reexecute your commands. My own typical commandline interaction goes like this:

1. I enter an R expression with a typo.
2. R complains about my mistake.
3. I press the up-arrow key to recall my mistaken line.
4. I use the left and right arrow keys to move the cursor back to the error.
5. I use the Delete key to delete the offending characters.
6. I type the corrected characters, which inserts them into the command line.
7. I press Enter to reexecute the corrected command.

That’s just the basics. R supports the usual keystrokes for recalling and editing command lines, as listed in



Labeled key	Ctrl-key combination	Effect
Up arrow	Ctrl-P	Recall previous command by moving backward through the history of commands.
Down arrow	Ctrl-N	Move forward through the history of commands.
Backspace	Ctrl-H	Delete the character to the left of cursor.
Delete (Del)	Ctrl-D	Delete the character to the right of cursor.
Home	Ctrl-A	Move cursor to the start of the line.
End	Ctrl-E	Move cursor to the end of the line.
Right arrow	Ctrl-F	Move cursor right (forward) one character.
Left arrow	Ctrl-B	Move cursor left (back) one character.
	Ctrl-K	Delete everything from the cursor position to the end of the line.
	Ctrl-U	Clear the whole darn line and start over.
Tab		Name completion (on some platforms).

Exiting from R

Windows
Select File → Exit from the main menu; or click on the red X in the upper-right corner of the window frame.

Whenever you exit, R asks if you want to save your workspace. You have three choices:

- Save your workspace and exit.
- Don’t save your workspace, but exit anyway.
- Cancel, returning to the command prompt rather than exiting.

If you save your workspace, then R writes it to a file called .RData in the current working directory. This will overwrite the previously saved workspace, if any, so don’t save if you don’t like the changes to your workspace (e.g., if you have accidentally erased critical data).

Interrupting R

You want to interrupt a long-running computation and return to the command prompt without exiting R.

Windows or OS X

Either press the Esc key or click on the Stop-sign icon.

Viewing the Supplied Documentation

Use the help.start function to see the documentation's table of contents:

```
> help.start()
```

From there, links are available to all the installed documentation.

The base distribution of R includes a wealth of documentation—literally thousands of pages. When you install additional packages, those packages contain documentation that is also installed on your machine.

It is easy to browse this documentation via the help.start function, which opens a window on the top-level table of contents; see Figure 1-2.

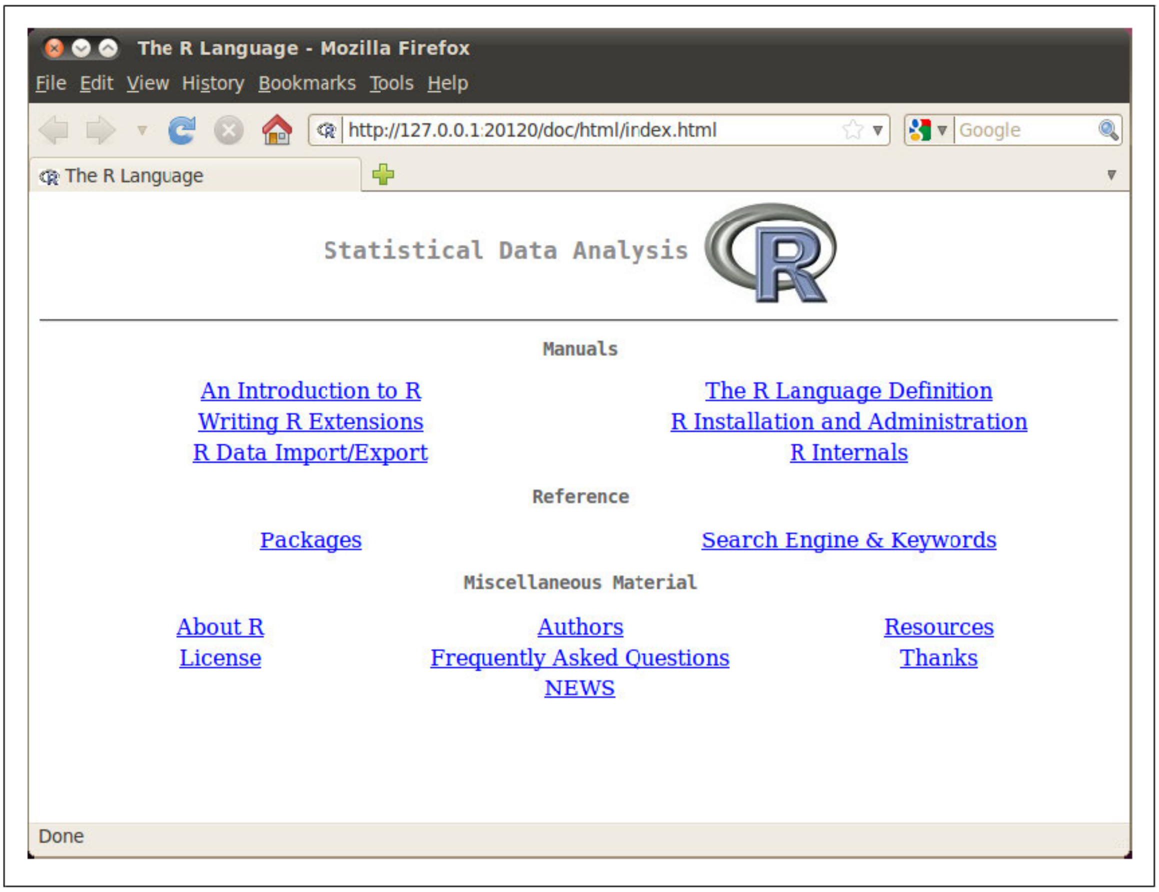
The two links in the Reference section are especially useful:

Packages

Click here to see a list of all the installed packages, both in the base packages and the additional, installed packages. Click on a package name to see a list of its functions and datasets.

Search Engine & Keywords

Click here to access a simple search engine, which allows you to search the documentation by keyword or phrase. There is also a list of common keywords, organized by topic; click one to see the associated pages.


The R Language - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:20120/doc/html/index.html
Google
The R Language
Done

Statistical Data Analysis

Manuals

An Introduction to R
Writing R Extensions
R Data Import/Export
The R Language Definition
R Installation and Administration
R Internals

Reference

Packages
Search Engine & Keywords

Miscellaneous Material

About R
License
Authors
Frequently Asked Questions
NEWS
Resources
Thanks

Fig : Documentation table of content

ing the help page for that function. One of its bells or whistles might be very useful to you.

Suppose you want to know more about the mean function. Use the help function like this:

```
> help(mean)
```

This will either open a window with function documentation or display the documentation on your console, depending upon your platform. A shortcut for the help command is to simply type ? followed by the function name:

```
> ?mean
```

Sometimes you just want a quick reminder of the arguments to a function: What are they, and in what order do they occur? Use the args function:

```
> args(mean)
function (x, ...)
NULL
> args(sd)
function (x, na.rm = FALSE)
NULL
```

The first line of output from args is a synopsis of the function call. For mean, the synopsis shows one argument, x, which is a vector of numbers. For sd, the synopsis shows the same vector, x, and an optional argument called na.rm. (You can ignore the second line of output, which is often just NULL.)

Most documentation for functions includes examples near the end. A cool feature of R is that you can request that it execute the examples, giving you a little demonstration of the function's capabilities. The documentation for the mean function, for instance, contains examples, but you don't need to type them yourself. Just use the example function to watch them run:

```
> example(mean)
mean> x <- c(0:10, 50)
mean> xm <- mean(x)
mean> c(xm, mean(x, trim = 0.1))
[1] 8.75 5.50
mean> mean(USArrests, trim = 0.2)
Murder Assault UrbanPop Rape
7.42 167.60 66.20 20.16
```

The user typed example(mean). Everything else was produced by R, which executed the examples from the help page and displayed the results.

Getting Help on a Package

You want to learn more about a package installed on your computer.

Use the help function and specify a package name (without a function name):

```
> help(package="packagename")
```

Discussion

Sometimes you want to know the contents of a package (the functions and datasets). This is especially true after you download and install a new package, for example. The help function can provide the contents plus other information once you specify the package name.

This call to help will display the information for the tseries package, a standard package in the base distribution:

```
> help(package="tseries")
```

The information begins with a description and continues with an index of functions and datasets. On my machine, the first few lines look like this:

```
Information on package 'tseries'
Description:
Package: tseries
Version: 0.10-22
Date: 2009-11-22
Title: Time series analysis and computational finance
Author: Compiled by Adrian Trapletti
<a.trapletti@swissonline.ch>
Maintainer: Kurt Hornik <Kurt.Hornik@R-project.org>
Description: Package for time series analysis and computational
finance
Depends: R (>= 2.4.0), quadprog, stats, zoo
Suggests: its
Imports: graphics, stats, utils
License: GPL-2
Packaged: 2009-11-22 19:03:45 UTC; hornik
Repository: CRAN
Date/Publication: 2009-11-22 19:06:50
Built: R 2.10.0; i386-pc-mingw32; 2009-12-01 19:32:47 UTC;
windows
Index:
NelPlo Nelson-Plosser Macroeconomic Time Series
USeconomic U.S. Economic Variables
adf.test Augmented Dickey-Fuller Test
arma Fit ARMA Models to Time Series
.
. (etc.)
.
```

Some packages also include vignettes, which are additional documents such as introductions, tutorials, or reference cards. They are installed on your computer as part of the package documentation when you install the package. The help page for a package includes a list of its vignettes near the bottom.

You can see a list of all vignettes on your computer by using the vignette function:

```
> vignette()
```

You can see the vignettes for a particular package by including its name:

```
> vignette(package="packagename")
```

Each vignette has a name, which you use to view the vignette:

```
> vignette("vignettename")
```

Printing Something

You want to display the value of a variable or expression

Solution

If you simply enter the variable name or expression at the command prompt, R will print its value. Use the print function for generic printing of any object. Use the cat function for producing custom formatted output.

It’s very easy to ask R to print something: just enter it at the command prompt:

```
> pi
[1] 3.141593
> sqrt(2)
[1] 1.414214
```

When you enter expressions like that, R evaluates the expression and then implicitly calls the print function. So the previous example is identical to this:

```
> print(pi)
[1] 3.141593
> print(sqrt(2))
[1] 1.414214
```

The beauty of print is that it knows how to format any R value for printing, including structured values such as matrices and lists:

```
> print(matrix(c(1,2,3,4), 2, 2))
[,1] [,2]
[1,] 1 3
[2,] 2 4
> print(list("a","b","c"))
[[1]]
[1] "a"
[[2]]
[1] "b"
[[3]]
[1] "c"
```

This is useful because you can always view your data: just print it. You needn’t write special printing logic, even for complicated data structures.

The print function has a significant limitation, however: it prints only one object at a time. Trying to print multiple items gives this mind-numbing error message:

```
> print("The zero occurs at", 2*pi, "radians.")
Error in print.default("The zero occurs at", 2 * pi, "radians.") :
unimplemented type 'character' in 'asLogical'
```

The only way to print multiple items is to print them one at a time, which probably isn’t what you want:

```
> print("The zero occurs at"); print(2*pi); print("radians")
[1] "The zero occurs at"
[1] 6.283185
[1] "radians"
```

The cat function is an alternative to print that lets you combine multiple items into a continuous output:

```
> cat("The zero occurs at", 2*pi, "radians.", "\n")
The zero occurs at 6.283185 radians.
```

Notice that cat puts a space between each item by default. You must provide a newline character (\n) to terminate the line.

The cat function can print simple vectors, too:

```
> fib <- c(0,1,1,2,3,5,8,13,21,34)
> cat("The first few Fibonacci numbers are:", fib, "...\n")
The first few Fibonacci numbers are: 0 1 1 2 3 5 8 13 21 34 ...
```

Using cat gives you more control over your output, which makes it especially useful in R scripts. A serious limitation, however, is that it cannot print compound data structures such as matrices and lists. Trying to cat them only produces another mindnumbing message:

```
> cat(list("a","b","c"))
Error in cat(list(...), file, sep, fill, labels, append) :
argument 1 (type 'list') cannot be handled by 'cat'
```

Variables in R

Naming Variables: A variable in R can store any object in R including atomic vector, list, matrix, array, factor and data frame. A valid variable name consists of letters, numbers and the dot or underline characters.

Assigning Values to Variables: In R, an assignment to a variable can be done in three ways = , <- and -> sign.

Finding Variables: To know all the variables currently available in the workspace we use the ls() function.

Removing Variables: Variable can be deleted by using the rm() function along with variable name.

Use the assignment operator (<-). There is no need to declare your variable first:

```
> x <- 3
```

Discussion

Using R in “calculator mode” gets old pretty fast. Soon you will want to define variables and save values in them. This reduces typing, saves time, and clarifies your work.

There is no need to declare or explicitly create variables in R. Just assign a value to the name and R will create the variable:

```
> x <- 3
> y <- 4
> z <- sqrt(x^2 + y^2)
> print(z)
[1] 5
```

Notice that the assignment operator is formed from a less-than character (<) and a hyphen (-) with no space between them.

When you define a variable at the command prompt like this, the variable is held in your *workspace*. The workspace is held in the computer’s main memory but can be saved to disk when you exit from R. The variable definition remains in the workspace until you remove it.

R is a *dynamically typed language*, which means that we can change a variable’s data type at will. We could set x to be numeric, as just shown, and then turn around and immediately overwrite that with (say) a vector of character strings. R will not complain:

```
> x <- 3
> print(x)
[1] 3
> x <- c("fee", "fie", "foe", "fum")
> print(x)
```

```
[1] "fee" "fie" "foe" "fum"
```

In some R functions you will see assignment statements that use the strange-looking assignment operator <<-:

```
x <<- 3
```

That forces the assignment to a global variable rather than a local variable.

In the spirit of full disclosure, I will reveal that R also supports two other forms of assignment statements. A single equal sign (=) can be used as an assignment operator at the command prompt. A rightward assignment operator (->) can be used anywhere the leftward assignment operator (<-) can be used:

```
> foo = 3
> print(foo)
[1] 3
> 5 -> fum
> print(fum)
[1] 5
```

These forms are never used in this book, and I recommend that you avoid them. The equals-sign assignment is easily confused with the test for equality. The rightward assignment is just too unconventional and, worse, becomes difficult to read when the expression is long.

Listing Variables

Problem

You want to know what variables and functions are defined in your workspace.

Solution

Use the ls function. Use ls.str for more details about each variable.

Discussion

The ls function displays the names of objects in your workspace:

```
> x <- 10
> y <- 50
> z <- c("three", "blind", "mice")

> f <- function(n,p) sqrt(p*(1-p)/n)
> ls()
[1] "f" "x" "y" "z"
```

Notice that ls returns a vector of character strings in which each string is the name of one variable or function. When your workspace is empty, ls returns an empty vector, which produces this puzzling output:

```
> ls()
character(0)
```

That is R’s quaint way of saying that ls returned a zero-length vector of strings; that is, it returned an empty vector because nothing is defined in your workspace.

If you want more than just a list of names, try ls.str; this will also tell you something about each variable:

```
> ls.str()
f : function (n, p)
x : num 10
y : num 50
z : chr [1:3] "three" "blind" "mice"
```

The function is called ls.str because it is both listing your variables and applying the

str function to them, showing their structure (Recipe 12.15).
Ordinarily, ls does not return any name that begins with a dot (.). Such names are considered hidden and are not normally of interest to users. (This mirrors the Unix convention of not listing files whose names begin with dot.) You can force ls to list everything by setting the all.names argument to TRUE:

```
> .hidvar <- 10
> ls()
[1] "f" "x" "y" "z"
> ls(all.names=TRUE)
[1] ".hidvar" "f" "x" "y" "z"
```

Deleting Variables
Problem
You want to remove unneeded variables or functions from your workspace or to erase its contents completely.
Solution
Use the rm function

Your workspace can get cluttered quickly. The rm function removes, permanently, one or more objects from the workspace:

```
> x <- 2*pi
> x
[1] 6.283185
> rm(x)
> x
Error: object "x" not found
```

There is no “undo”; once the variable is gone, it’s gone.
You can remove several variables at once:

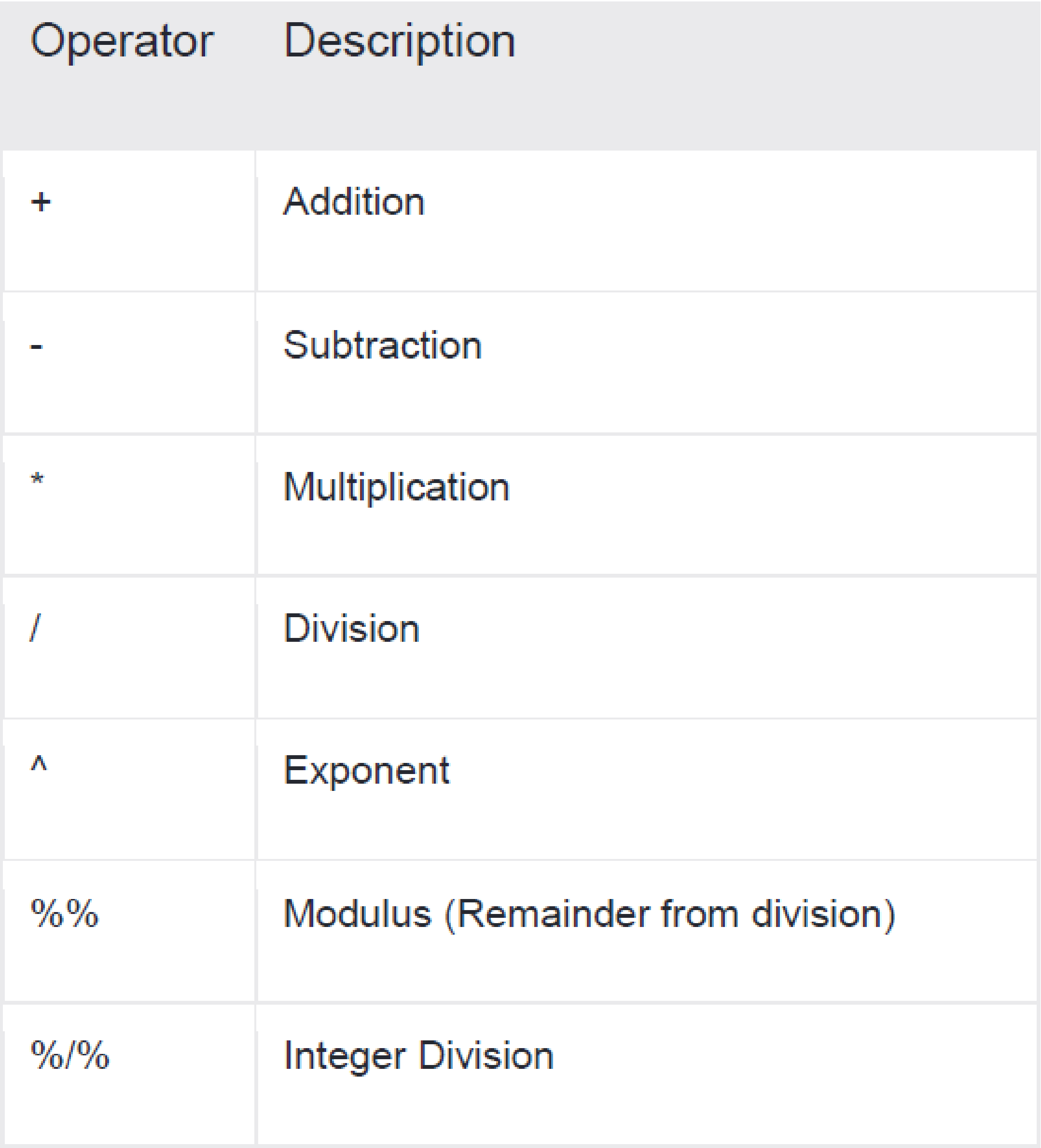
```
> rm(x,y,z)
```

You can even erase your entire workspace at once. The rm function has a list argument consisting of a vector of names of variables to remove. Recall that the ls function returns a vector of variables names; hence you can combine rm and ls to erase everything:

```
> ls()
[1] "f" "x" "y" "z"
> rm(list=ls())
> ls()
character(0)
```

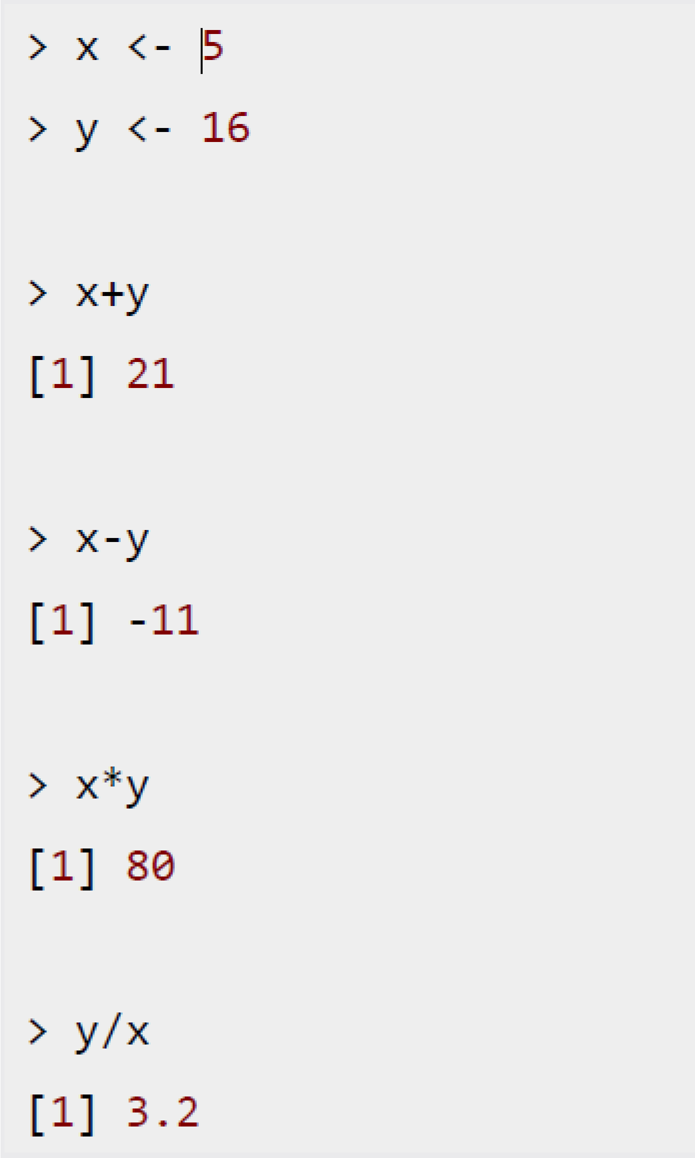
Arithmetic Operations in R

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

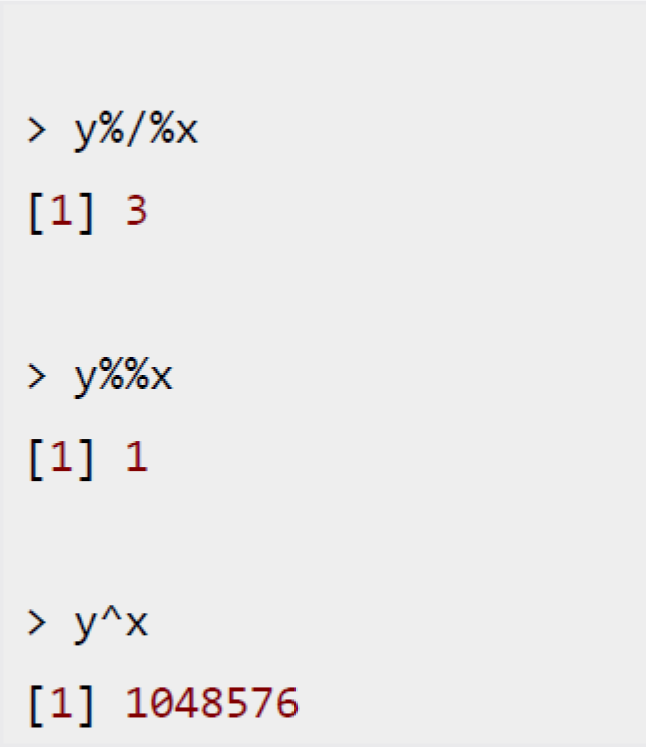


Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)
%/%	Integer Division

An example run



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

> x+y
[1] 21

> x-y
[1] -11

> x*y
[1] 80

> y/x
[1] 3.2
```

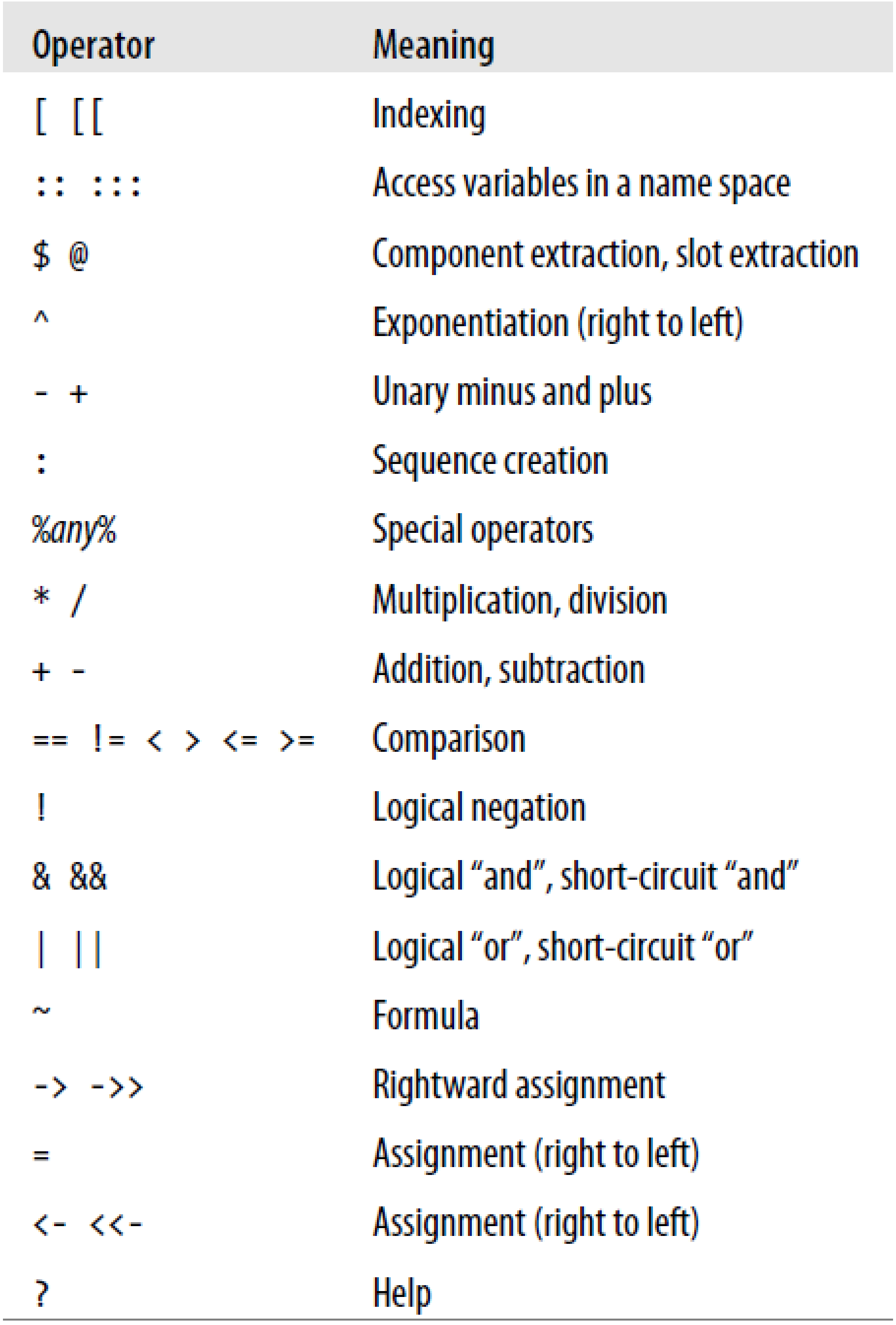


```
> y%/%x

[1] 3


> y%%x

[1] 1


> y^x

[1] 1048576
```

Getting Operator Precedence Right

Your R expression is producing a curious result, and you wonder if operator precedence is causing problems.

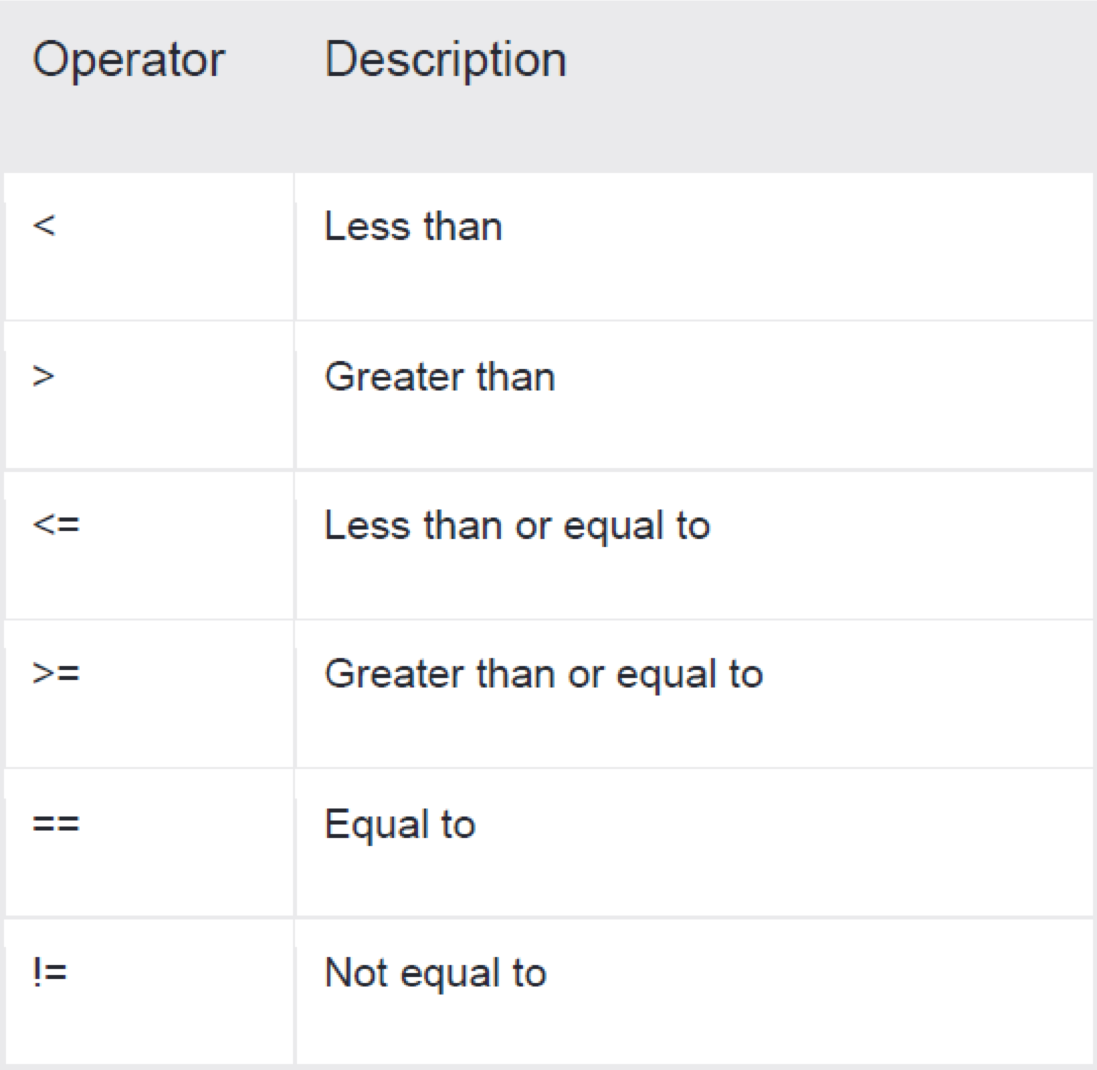
The full list of operators is shown in Table 2-1, listed in order of precedence from highest to lowest. Operators of equal precedence are evaluated from left to right except where indicated.



Operator	Meaning
[[[	Indexing
:: :::	Access variables in a name space
$ @	Component extraction, slot extraction
^	Exponentiation (right to left)
- +	Unary minus and plus
:	Sequence creation
%any%	Special operators
* /	Multiplication, division
+ -	Addition, subtraction
== != < > <= >=	Comparison
!	Logical negation
& &&	Logical "and", short-circuit "and"
| ||	Logical "or", short-circuit "or"
~	Formula
-> ->>	Rightward assignment
=	Assignment (right to left)
<- <<-	Assignment (right to left)
?	Help

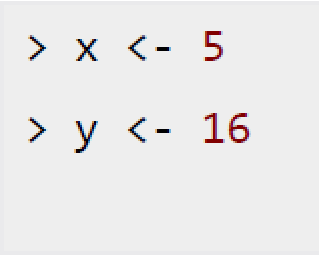
Relational Operators in R

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

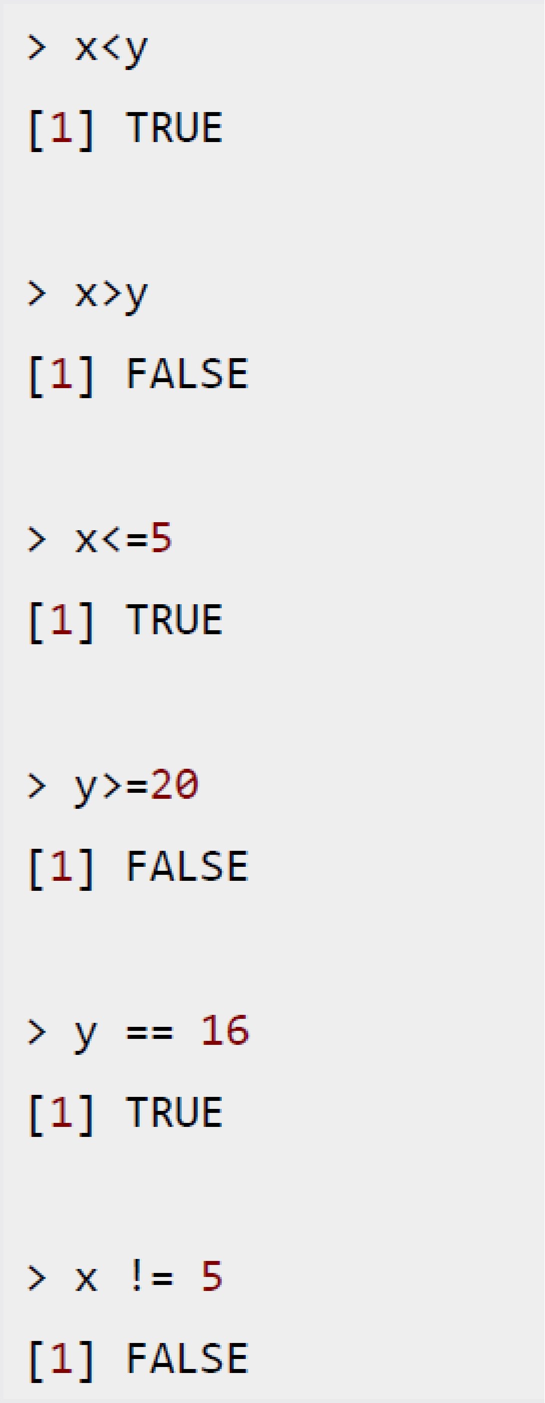


Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

An example run



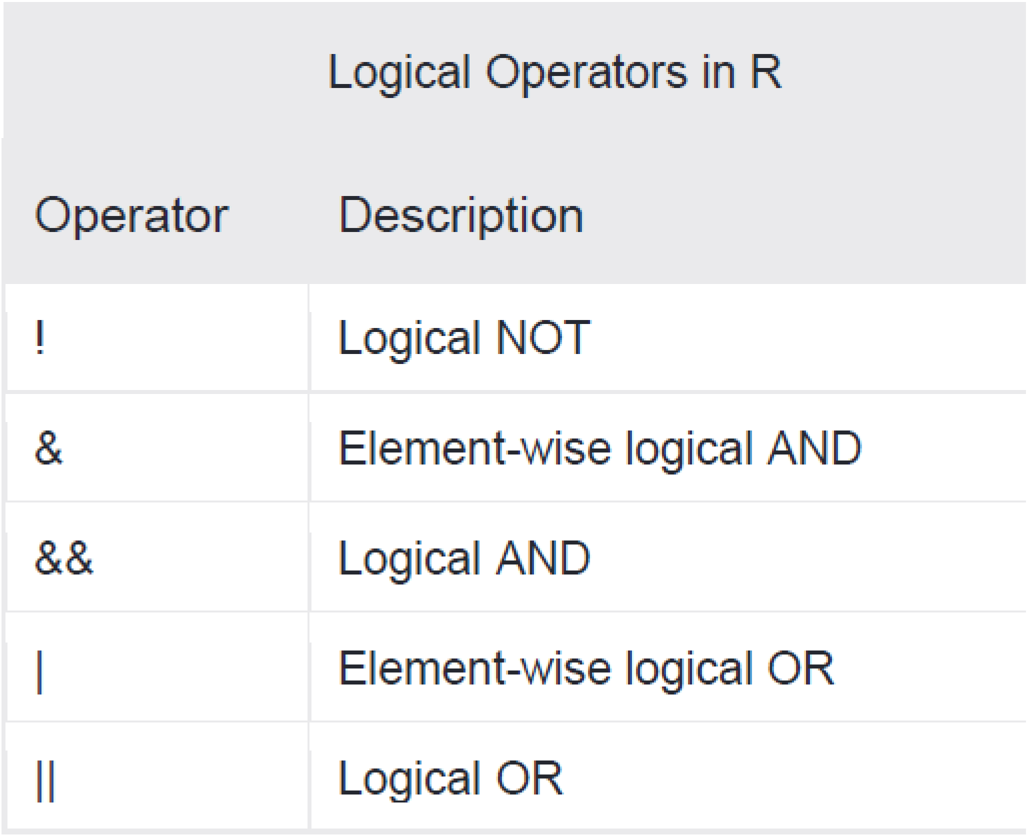
```
> x <- 5
> y <- 16
```



```
> x<y
[1] TRUE

> x>y
[1] FALSE

> x<=5
[1] TRUE

> y>=20
[1] FALSE

> y == 16
[1] TRUE

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

Logical Operations in R

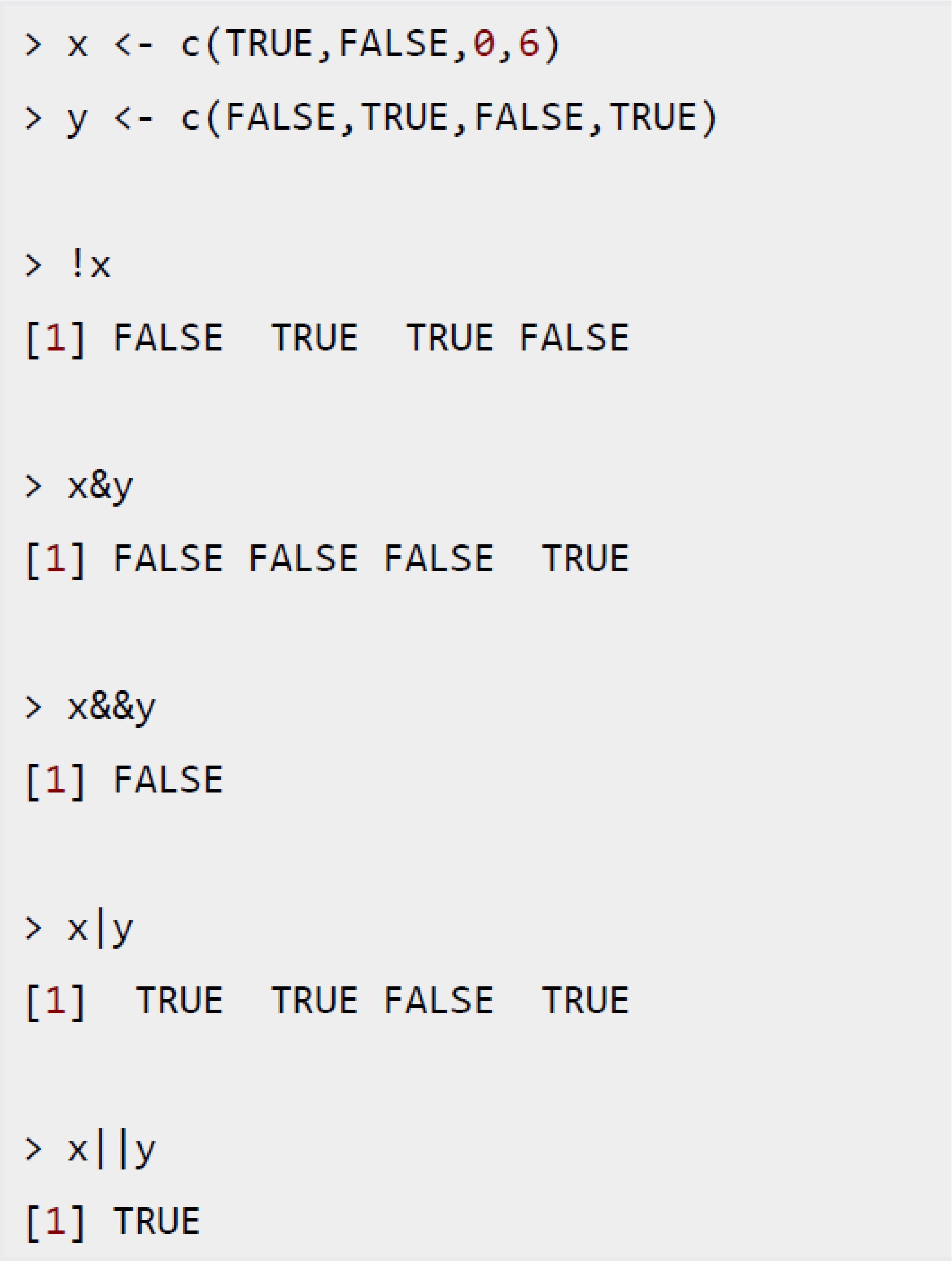
Logical operators are used to carry out Boolean operations like AND, OR etc.



Logical Operators in R

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>!</td><td>Logical NOT</td></tr>
<tr><td>&</td><td>Element-wise logical AND</td></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>|</td><td>Element-wise logical OR</td></tr>
<tr><td>||</td><td>Logical OR</td></tr>
</table>

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.

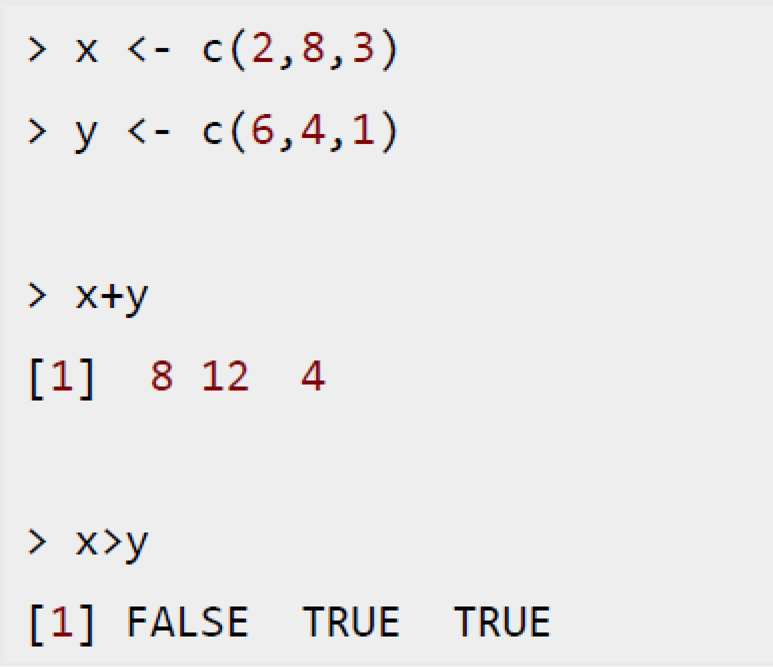


```
> x <- c(TRUE,FALSE,0,6)
> y <- c(FALSE,TRUE,FALSE,TRUE)

> !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
```

Vector

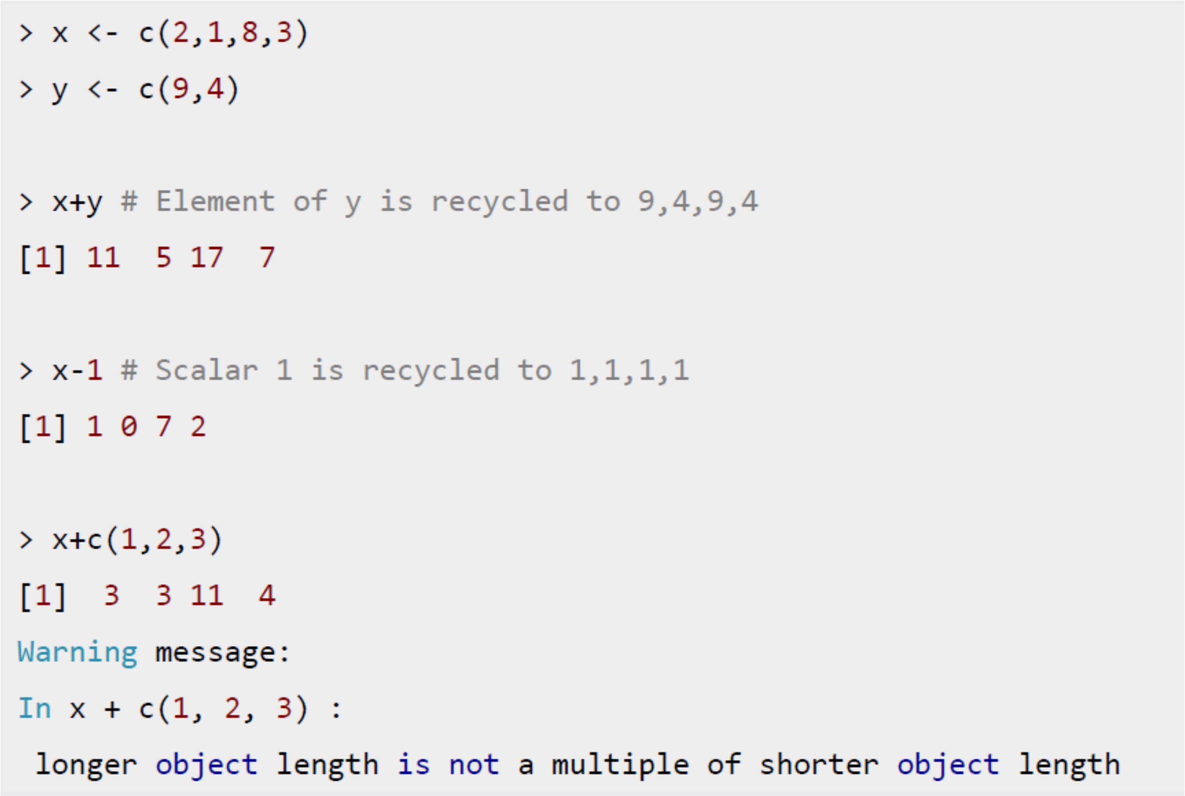
Lets create a vector containing three numeric values. This will be an example of Numeric Vector

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.



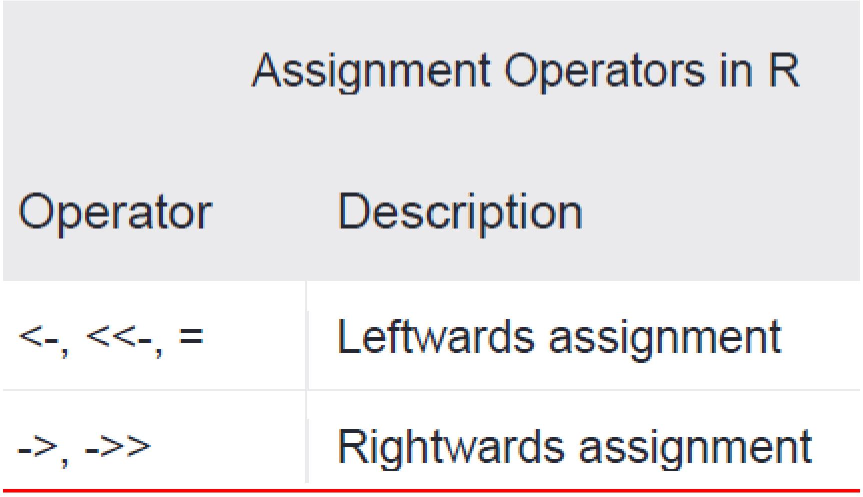
```
> x <- c(2,1,8,3)
> y <- c(9,4)

> 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
```

<u>Vectors Vs List</u>

Vector:	List:
Vector: In many programming languages, a vector is a dynamic array that can resize itself automatically. Vectors typically provide constant-time access to individual elements and amortized constant-time for appending new elements.	List: A list is a more generic term and can refer to various data structures. In some languages, a list is synonymous with an array or a vector, while in others, it might refer to a linked list. Linked lists have nodes where each node contains a value and a reference (or link) to the next node.
Vectors often allocate contiguous memory, making them efficient for random access to elements.	Lists can use contiguous or non-contiguous memory depending on the type. Linked lists, for example, use non-contiguous memory as each element points to the next one.
Vectors are generally more efficient when it comes to random access to elements due to their contiguous memory allocation.	Lists, especially linked lists, can be more efficient for inserting or deleting elements in the middle, as it doesn't require shifting elements like in a vector.
Vectors are often more rigid in size, but some languages provide resizable vectors. Once a vector reaches its capacity, it may need to allocate a new, larger chunk of memory and copy elements over.	Lists, especially linked lists, can easily grow or shrink without the need for reallocation. Each element points to the next one, so inserting or removing elements is more straightforward.
In many languages, vectors have specific syntax and methods/functions for operations like appending, accessing elements, and resizing.	Lists can be more generic, and the term might be used interchangeably with arrays or other data structures. Syntax and methods can vary widely depending on the language and the specific type of list (e.g., linked list, array list).

Assigning Variables

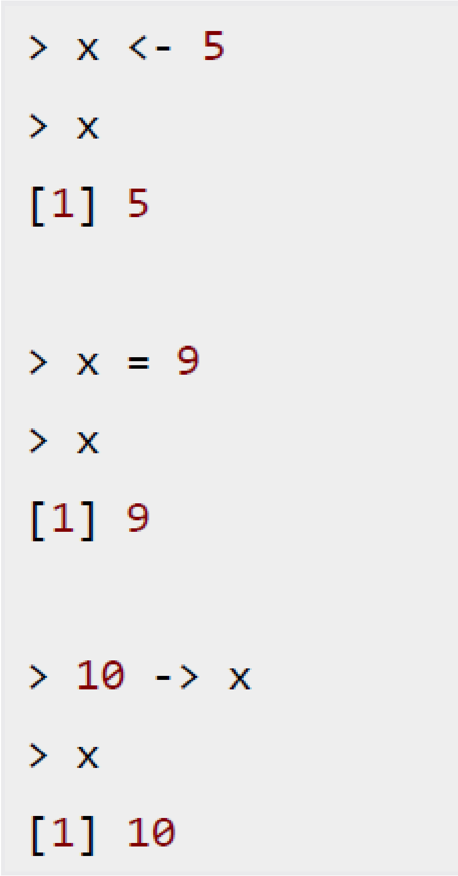
These operators are used to assign values to variables



Assignment Operators in R

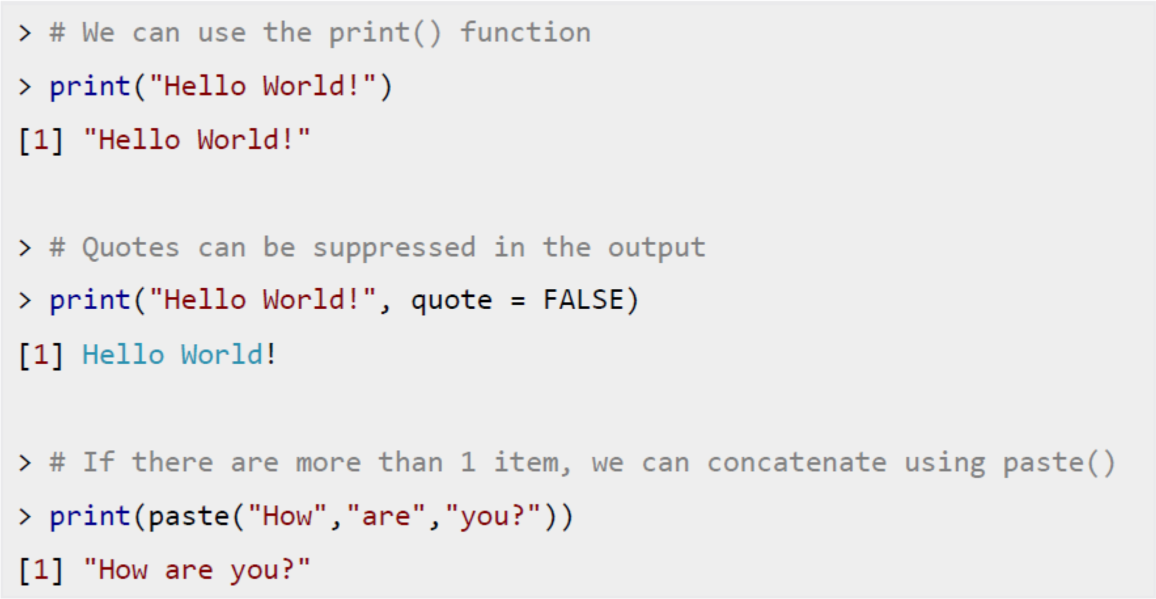
Operator	Description
<-, <<-, =	Leftwards assignment
->, ->>	Rightwards assignment

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



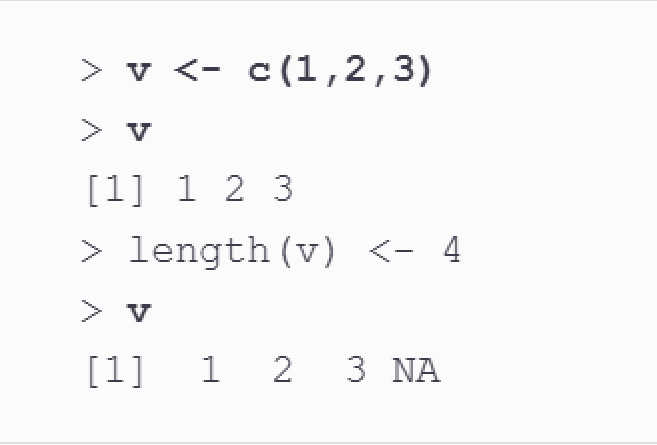
```
> # We can use the print() function
> print("Hello World!")
[1] "Hello World!"

> # 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?"
```

```
# Using assign() function
variable_name <- "z"
assign(variable_name, 15)

# Using the equal sign =
y = 5
```

Special Numbers

NA

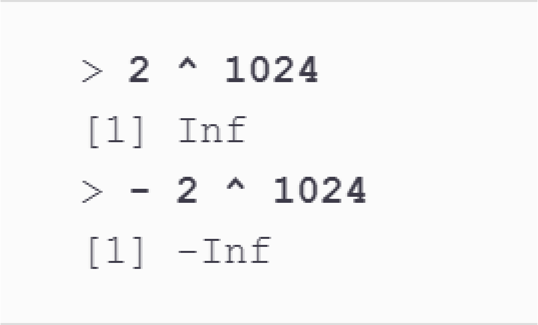
In R, the NA values are used to represent missing values. (NA stands for “not available.”) You may encounter NA values in text loaded into R (to represent missing values) or in data loaded from databases (to replace NULL values).
If you expand the size of a vector (or matrix or array) beyond the size where values were defined, the new spaces will have the value NA:



```
> v <- c(1,2,3)
> v
[1] 1 2 3
> length(v) <- 4
> v
[1]  1  2  3 NA
```

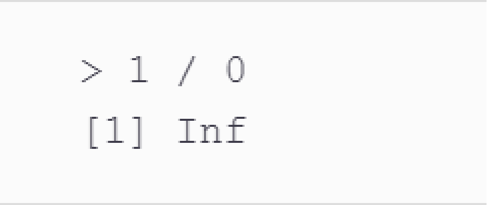
Inf and -Inf

If a computation results in a number that is too big, R will return Inf for a positive number and -Inf for a negative number (meaning positive and negative infinity, respectively):



```
> 2 ^ 1024
[1] Inf
> - 2 ^ 1024
[1] -Inf
```

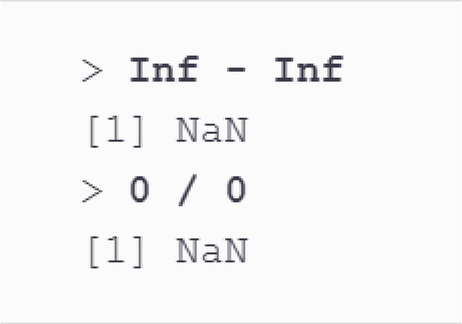
This is also the value returned when you divide by 0:



```
> 1 / 0
[1] Inf
```

NaN

Sometimes, a computation will produce a result that makes little sense. In these cases, R will often return NaN (meaning “not a number”):



```
> Inf - Inf
[1] NaN
> 0 / 0
[1] NaN
```

NULL

Additionally, there is a null object in R, represented by the symbol NULL. (The symbol NULL always points to the same object.) NULL is often used as an argument in functions to mean that no value was assigned to the argument. Additionally, some functions may return NULL. Note that NULL is not the same as NA, Inf, -Inf, or NaN.

Changing classes in R Programming

as. Functions: The **as.** functions are used to coerce or convert objects from one class to another. For example:

```
# Convert numeric to character
x <- 123
x_char <- as.character(x)

# Convert character to numeric
y <- "456"
y_num <- as.numeric(y)
```

as() Function: The **as()** function is a more general way to coerce objects to a specified class:

```
# Convert numeric to character
x <- 123
x_char <- as(x, "character")

# Convert character to numeric
y <- "456"
y_num <- as(y, "numeric")
# Convert numeric to character
x <- 123
x_char <- as(x, "character")

# Convert character to numeric
y <- "456"
y_num <- as(y, "numeric")
```

factor() Function: If you want to convert a vector to a factor, you can use the **factor()** function:

```
# Convert a character vector to a factor
colors <- c("red", "green", "blue")
factor_colors <- factor(colors)
```

as.Date() Function: To convert a character or numeric vector to a Date object, you can use **as.Date()**:

```
# Convert character to Date
```

date_str <- "2024-03-06"

date_obj <- as.Date(date_str)

matrix() Function: If you want to convert a vector to a matrix, you can use the **matrix()** function:

# Convert a numeric vector to a matrix

vec <- 1:6

mat <- matrix(vec, nrow = 2, ncol = 3)

R Packages

Packages are bundles of code and / or data that can be written by anyone in the R community, but must comply with some requirements:

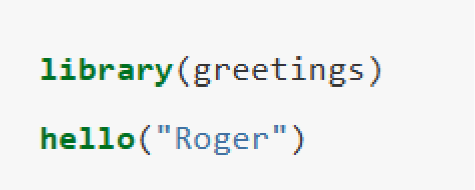
- All functions and data should be documented
- These help files have a very specific formatting (in a .Rd file)
- The file structure is also rigid
- Once packaged, the file can be uploaded to CRAN, github, Bioconductor, …

What is an R package?

Packages are bundles of code and / or data that can be written by anyone in the R community, but must comply with some requirements:

- Two files are mandatory: DESCRIPTION and NAMEFILE
- If sent to CRAN, a technical check will be conducted and feedback will be provided. If any error or warning appears when compiling the package will not be accepted

- This section will cover how to build R packages using R and RStudio. Using RStudio for this section will be critical as RStudio includes a number of tools that make building R packages *much* simpler.
- For the purposes of demonstration in this section, we will be building a package called greetings that has a single function called hello(). The hello() function takes a single argument called name (which is required) and makes a plot containing a message directed at name. For example,



```
library(greetings)
hello("Roger")
```

Admittedly, this is not a useful package, but it allows us to demonstrate all of the necessary ingredients for building a simple R package.

Prerequisites

Before starting you **must** install two additional packages:

- devtools - this provides many additional tools for building packages
- roxygen2 - this provides tools for writing documentation

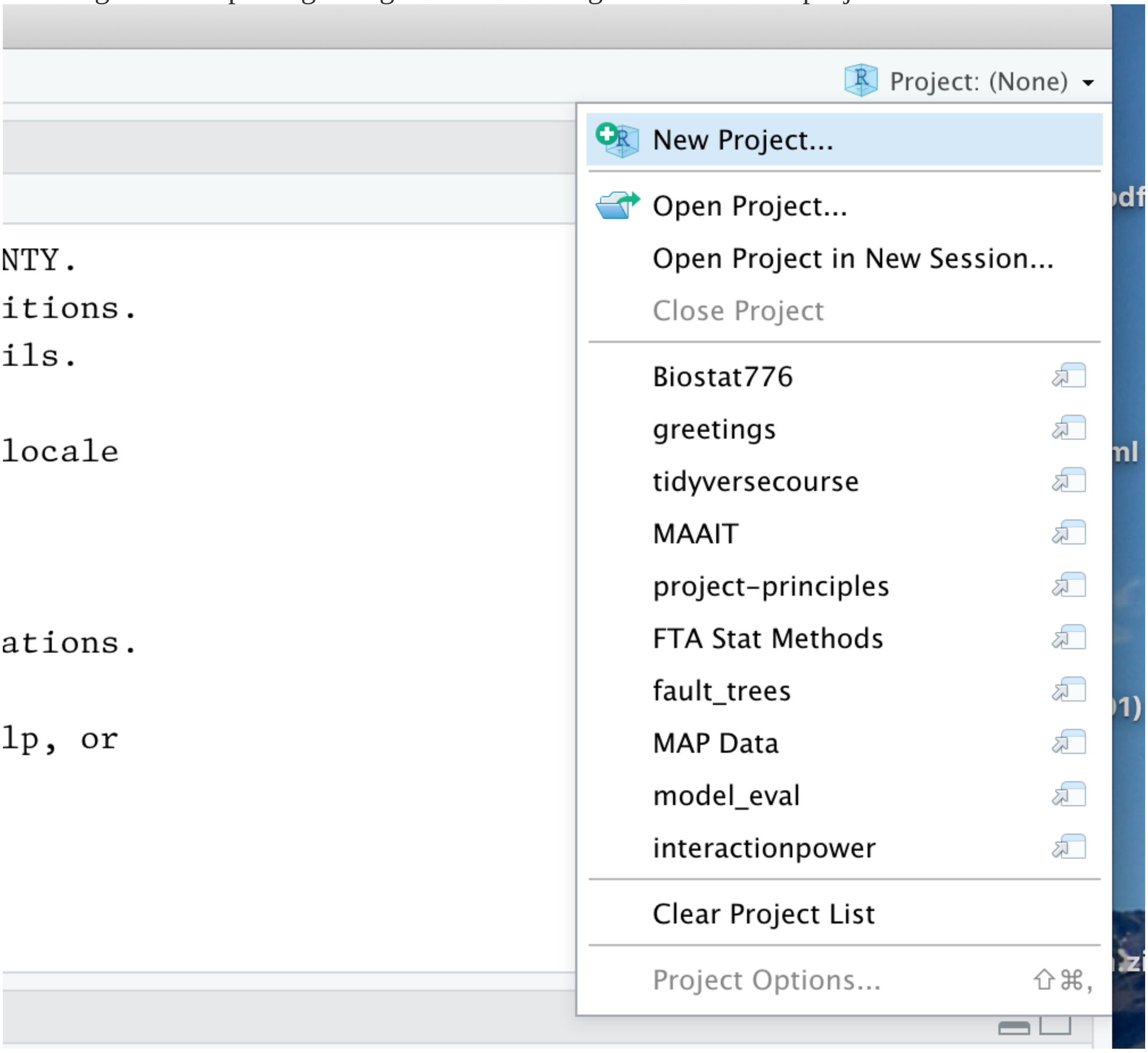
You can do this by calling

```
install.packages(c("devtools", "roxygen2"))
```

or use the “Install Packages…” option from the “Tools” menu in RStudio.

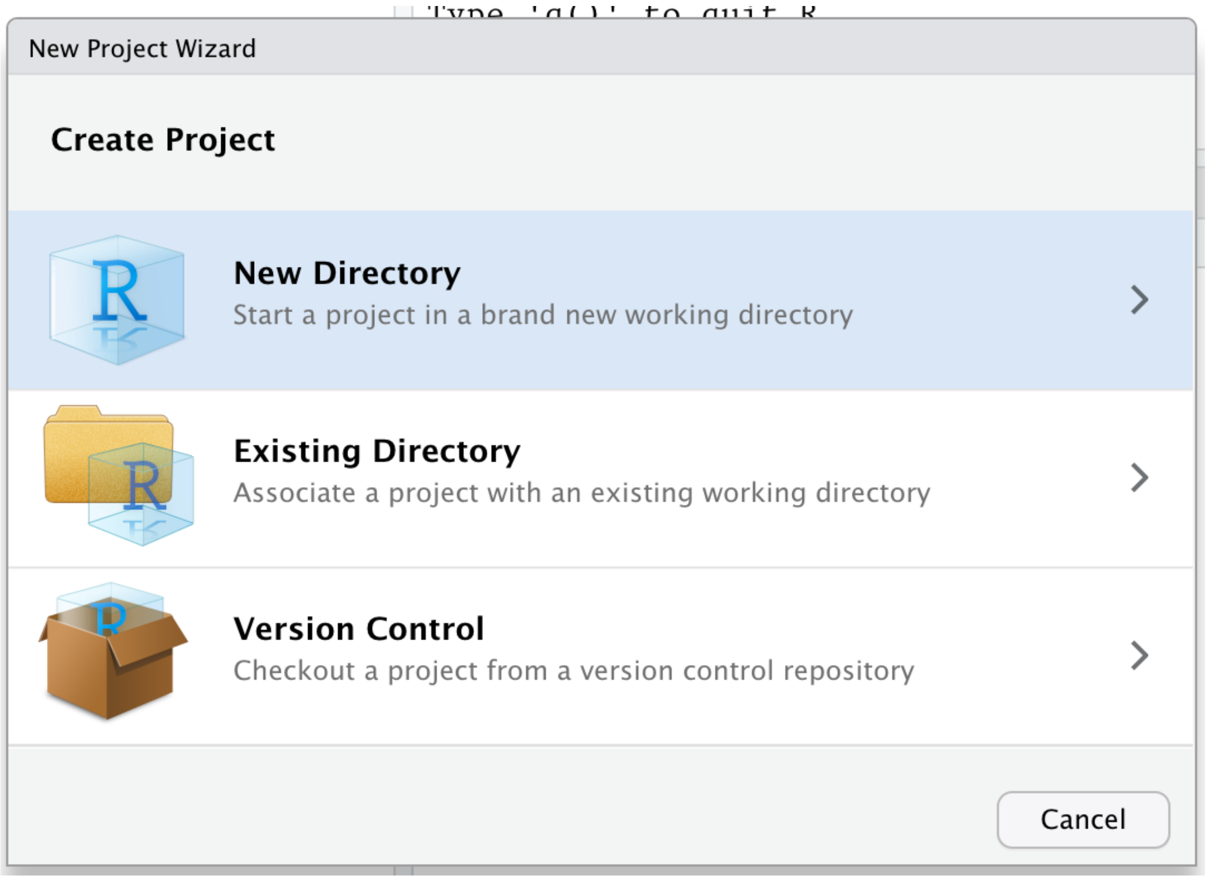
Create a New R Package Project

Creating a new R packages begins with creating a new RStudio project.



New Proejct

You should choose **New Directory** as this will be a brand new project (not an existing project).



Create Project

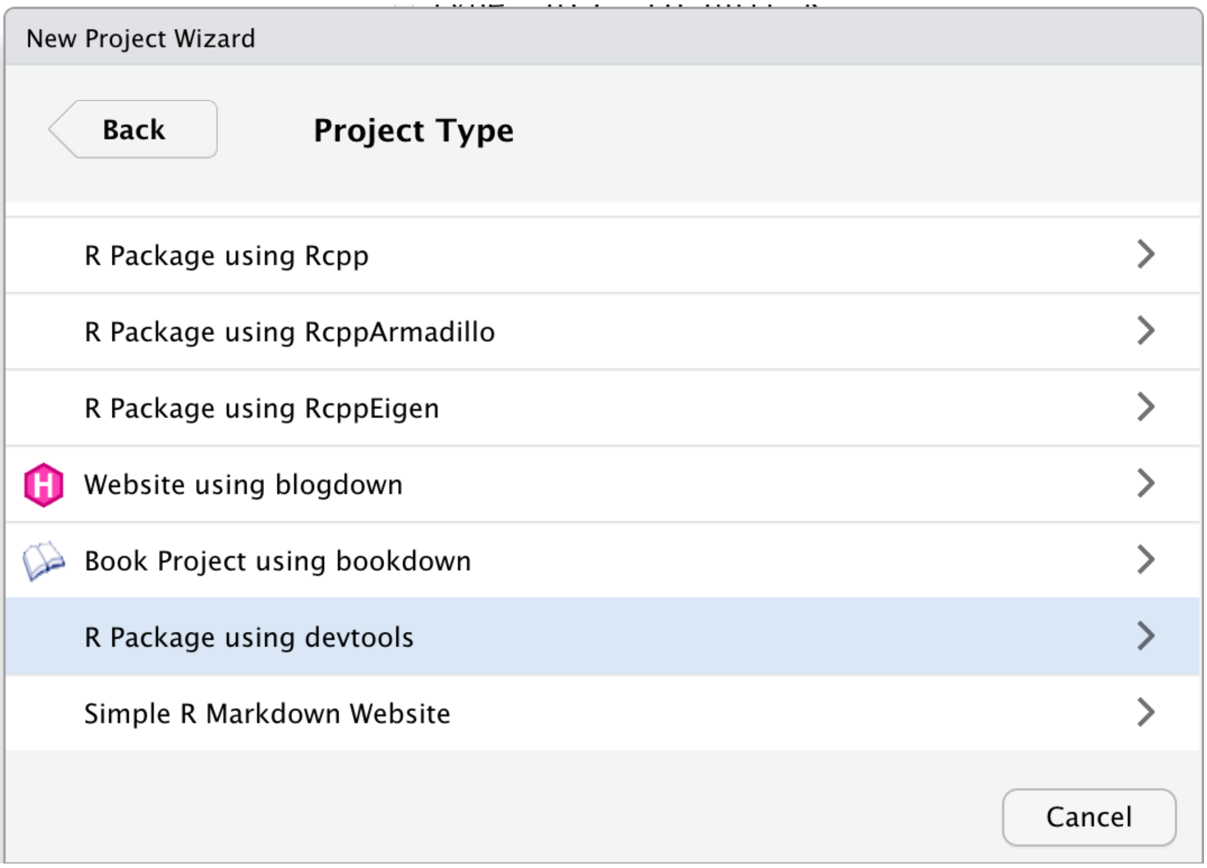
New Directory
Start a project in a brand new working directory

Existing Directory
Associate a project with an existing working directory

Version Control
Checkout a project from a version control repository

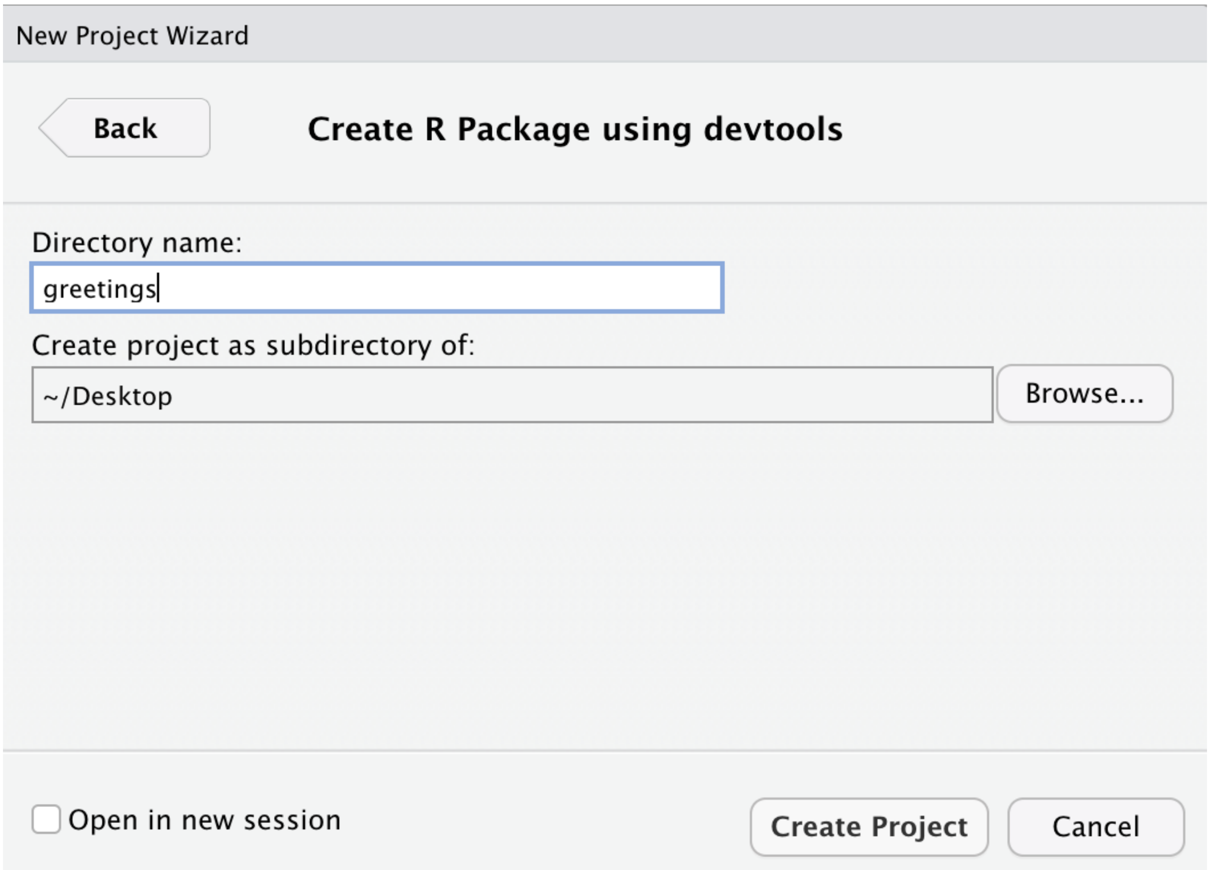
New Directory

Next, you should choose the “Project Type” as **R Package using devtools** (you may need to scroll down a little in that menu).


New Project Wizard
Back
Project Type
R Package using Rcpp
R Package using RcppArmadillo
R Package using RcppEigen
Website using blogdown
Book Project using bookdown
R Package using devtools
Simple R Markdown Website
Cancel

R Package using devtools

Finally, you should specify the name of your package. For this example, we will use **greetings** as the name of the package. Also you should double check the name of the sub-directory listed under “Create project as subdirectory of” is a directory that you can find.



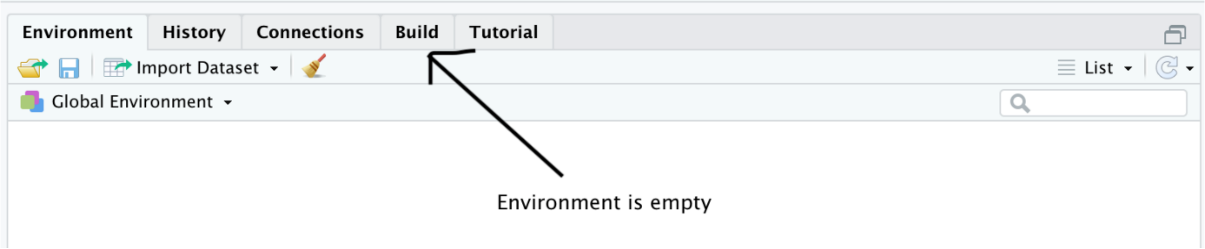
Directory name: greetings

Create project as subdirectory of: ~/Desktop

☐ Open in new session

Package Name

Click "Create Project" and allow R and RStudio to restart. You should get a brand new session. You will also see a window with a series of tabs. One of those tabs will be called **Build** and that will be important as we build our package.

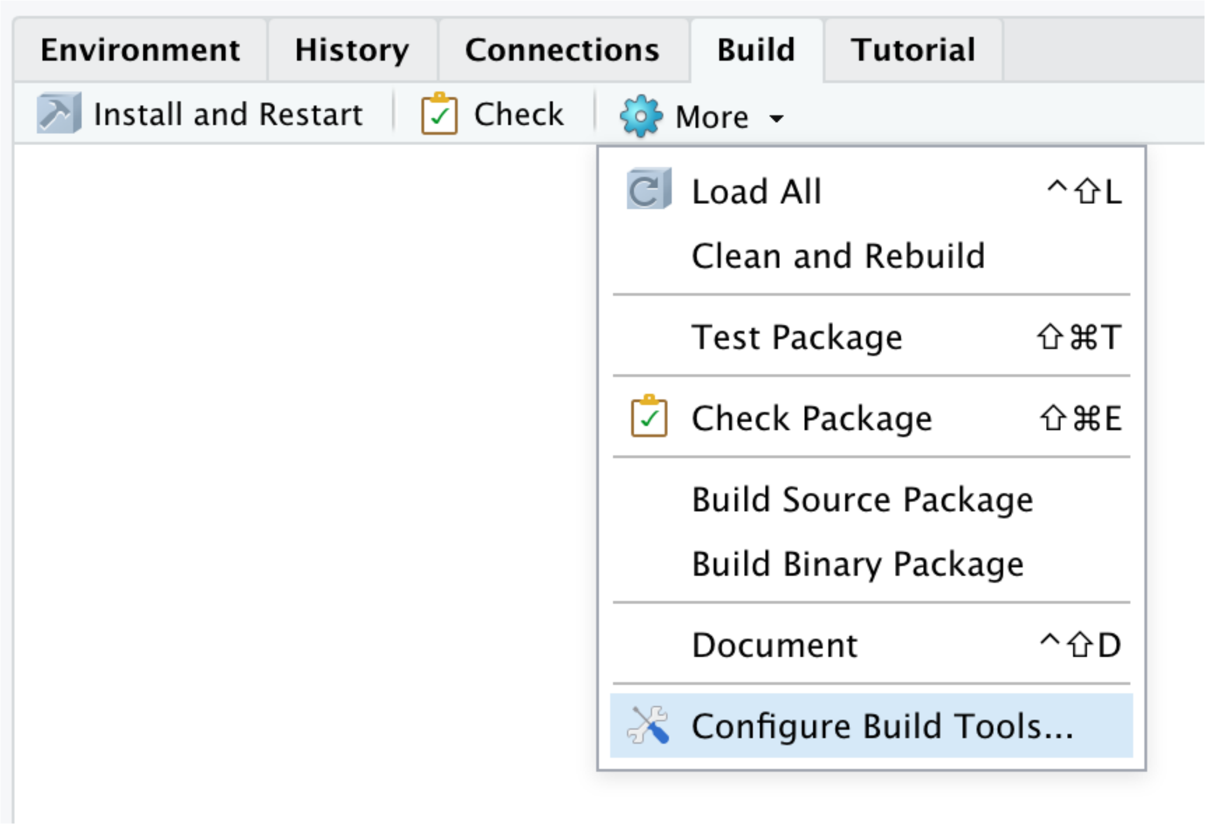


Environment is empty

Build Menu

27.2 Configure Build Tools

The next step after creating a new project is to configure your build tools. Click on the **Build** tab and then **More** and then **Configure Build Tools…**.

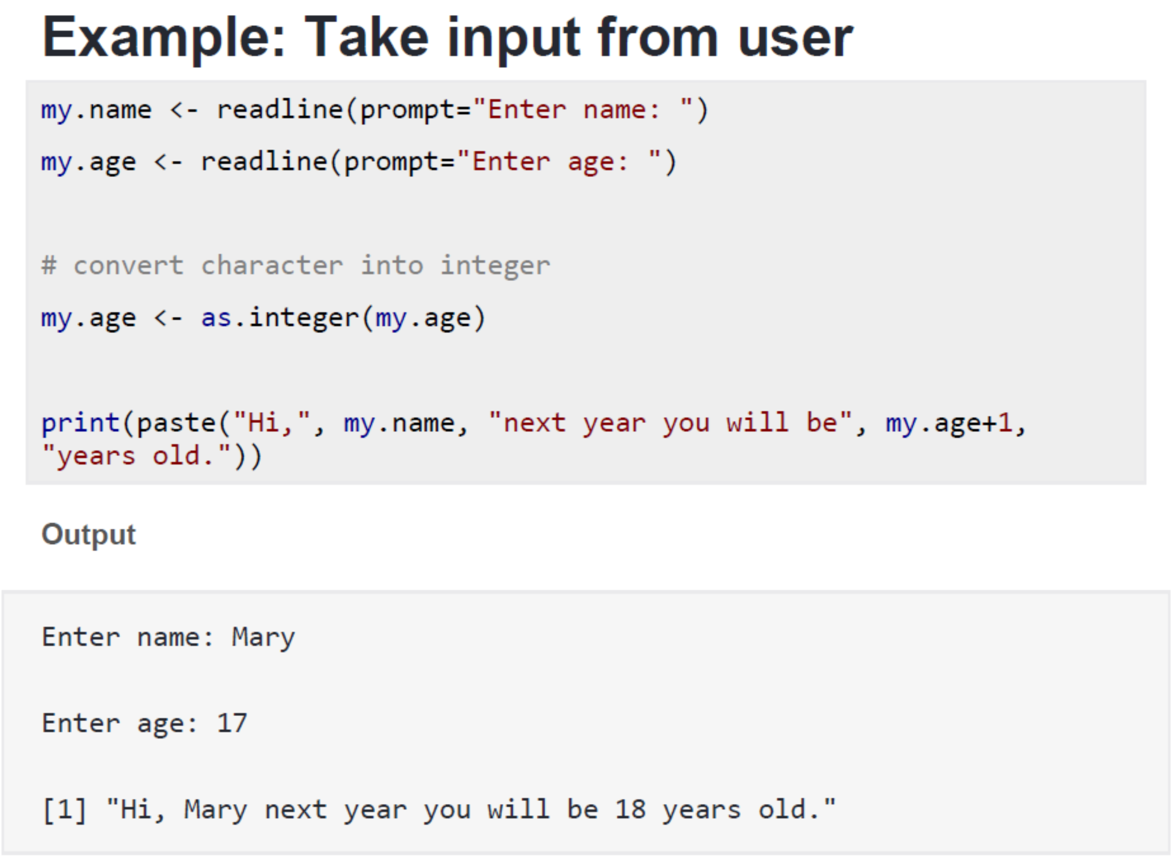


Input in R

Input of Data from Terminal: The scan function is used to take data from the user at the terminal.

Input of Data through R Objects: There are many types of R-objects including Vectors, Lists, Matrices, Arrays, Factors and Data Frames.

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

```r
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.

Output in R

print() Function: Print cannot combine two or more strings, variables, a string and a variable.

cat() Function: The cat() function is an alternative to print that lets you combine multiple items into a continuous output.

Using print() Function

One of the most fundamental ways to display output in R is by employing the print() function. It is versatile and can display various data types, such as vectors, matrices, data frames, etc. The print() function takes the object you want to display as an argument and outputs it to the console.

Here's an example that demonstrates how to use the print() function to display a simple text message:

```r
# Displaying a text message using the print() function

print("Hello, World!")
```

Output:

```
[1] "Hello, World!"
```

Besides text, the print() function can also display the contents of variables, such as vectors. Here's an example:

```
# Creating a vector

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


# Displaying the contents of the vector using the print() function

print(my_vector)
```

Output:

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

The print() function is the cornerstone for output display in R and is immensely useful for debugging and data analysis. Although more sophisticated methods exist for producing output, understanding and utilizing print() effectively is crucial for anyone working with R.

Using paste() Function

The paste() function in R is particularly useful for concatenating and displaying strings and combining strings with the values of variables. This function takes multiple arguments and concatenates them into a single string. The sep parameter can specify a separator between the concatenated elements. By default, the separator is a space.

Here's an example demonstrating the use of the paste() function to concatenate and display strings:

```
# Concatenating and displaying strings using the paste() function

name <- "Alice"

greeting <- paste("Hello,", name, "! How are you?")

print(greeting)
```

Output:

```
[1] "Hello, Alice ! How are you?"
```

The paste() function can also concatenate and display numerical values and strings. Below is an example:

```
# Concatenating and displaying numbers with strings using the paste() function

x <- 5

y <- 10
```

```
result <- paste("The sum of", x, "and", y, "is", x + y, ".")

print(result)
```

Output:

```
[1] "The sum of 5 and 10 is 15 ."
```

The paste() function is an essential tool for crafting custom messages and formatted strings in R. Combining paste() with print() allows for greater control over the structure and content of the output, enabling users to generate more informative and readable results.

Difference betwwen paste() and paste0() function

The paste() and paste0() functions in R are both used to concatenate or combine strings. They work similarly but have a key difference regarding the use of separators.

1. **paste():**
   This function concatenates vectors after converting to character. By default, it separates the combined strings with a space.

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

2. **paste0():**
   This function also concatenates vectors after converting to character. However, it does not insert any separators between the combined strings by default.

```
paste0("Hello", "World")  # Output: "HelloWorld"
```

So, the essential difference between the two is that paste() includes a space between the strings to be concatenated by default, while paste0() does not. You can modify this behavior by specifying a different separator for paste(), or by using the sep argument in either function. For example, paste("Hello", "World", sep = "-") would result in "Hello-World".

Using sprintf() Function

The sprintf() function in R is inspired by the C programming language's printf() function. It allows you to format strings with placeholders, making it useful for creating structured and well-formatted outputs. The sprintf() function takes a format string as its first argument, followed by any number of arguments to replace the placeholders in the format string.

Here's an example demonstrating the use of the sprintf() function to format a string with placeholders:

```
# Using sprintf() to format a string

name <- "John"

age <- 30

formatted_string <- sprintf("My name is %s and I am %d years old.", name, age)
```

```
print(formatted_string)
```

Output:

```
[1] "My name is John and I am 30 years old."
```

The %s is a placeholder for a string, while %d is a placeholder for an integer. Several other placeholders are available, such as %f for floating-point numbers.

Another example involves using sprintf() to format floating-point numbers:

```
# Using sprintf() to format a floating-point number
pi_value <- 3.141592653589793
formatted_pi <- sprintf("The value of pi to 4 decimal places is %.4f.", pi_value)
print(formatted_pi)
```

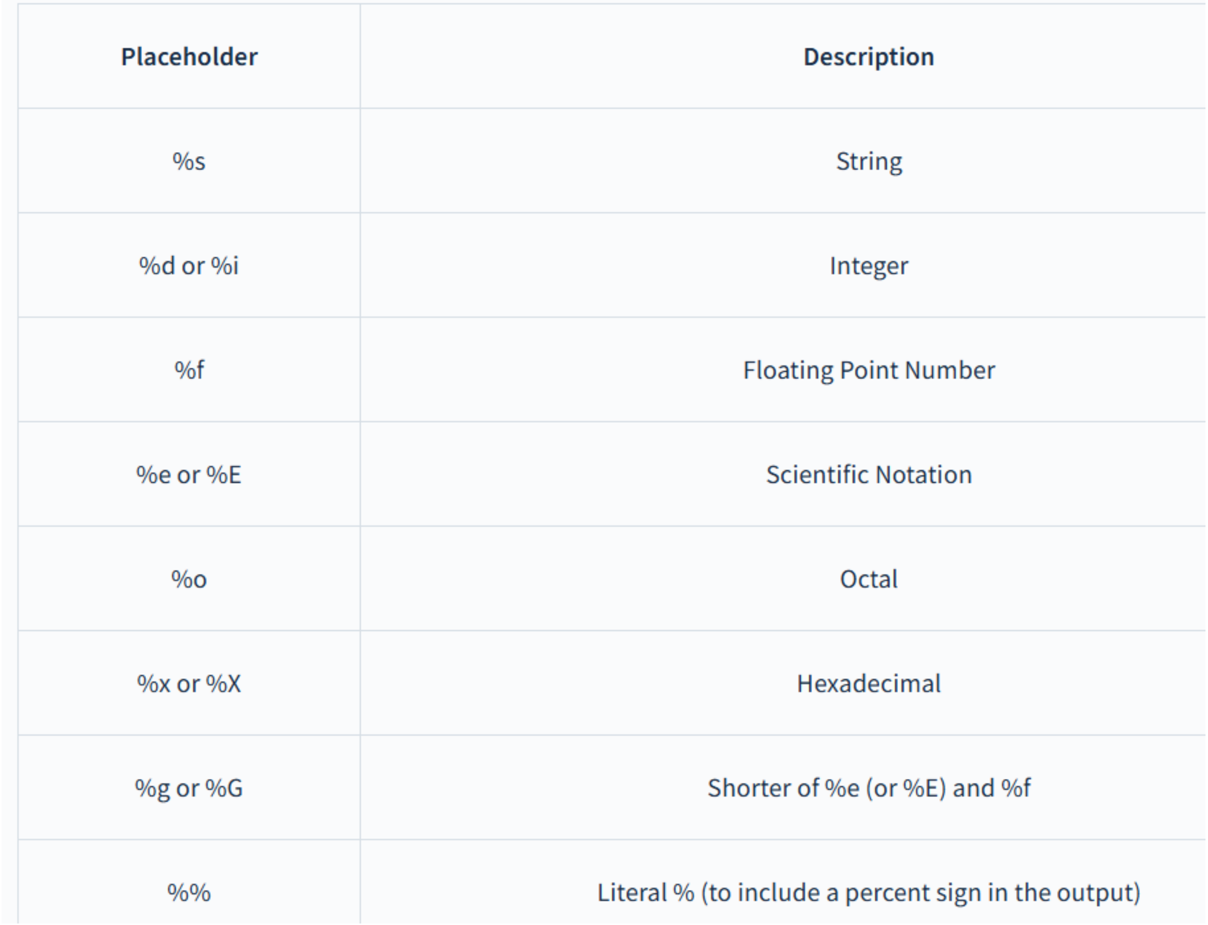
Output:

```
[1] "The value of pi to 4 decimal places is 3.1416."
```

n this example, %.4f is a placeholder that formats a floating-point number to 4 decimal places.

The sprintf() function is invaluable when you need to format strings and numbers in a specific way, particularly for generating structured output and reports.

The sprintf() function uses placeholders to specify the type of data that should be inserted into a string. Below is a table of common placeholders:



Placeholder	Description
%s	String
%d or %i	Integer
%f	Floating Point Number
%e or %E	Scientific Notation
%o	Octal
%x or %X	Hexadecimal
%g or %G	Shorter of %e (or %E) and %f
%%	Literal % (to include a percent sign in the output)

You can use these placeholders within the format string of the sprintf() function. For example:

```
sprintf("Hello %s, your score is %d", "Alice", 95)
```

Using cat() Function

The cat() function in R is another method for displaying output but more flexible than the print() function. It allows you to concatenate and print objects to the console, files, or other connections and is often used for creating more customized outputs. Unlike print(), the cat() function does not display extra information such as vector indices, making the output cleaner for presentation purposes.

Here’s an example demonstrating the use of the cat() function to display a customized message:

```
# Using the cat() function to display a customized message

name <- "Bob"

cat("Hello,", name, "!\n")
```

Output:

```
Hello, Bob !
```

Notice that you can include special characters such as "\n" for a new line within cat().

Another example involves using cat() to print the elements of a vector:

```
# Using the cat() function to print the elements of a vector

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

cat("Elements in my_vector:", my_vector, "\n")
```

Output:

```
Elements in my_vector: 1 2 3 4 5
```

The cat() function is useful when you need more control over the output formatting, especially when creating output for external audiences or writing data to files. By understanding and leveraging the capabilities of cat(), you can create polished and professional output in your R programs.

In R, as in many programming languages, there are certain special characters known as escape sequences that are used to represent certain special characters that you can't type into the code directly. Two of these are "\n" and "\t".

1. **"\n"**:
   This is the newline character. It inserts a line break, meaning anything after it will be on the next line when the string is printed or written to a file.

```
print("Hello\nWorld")

# Output:

# [1] "Hello

# World"
```

The output is on two lines because of the "\n". 2. **"\t"**: This is the tab character. It inserts a tab space, which is wider than a regular space.

```
print("Hello\tWorld")

# Output:

# [1] "Hello   World"
```

In addition to "\n" and "\t", there are several other escape sequences used in R strings. Here are some of them:

1. "\\":
   This is the escape character itself. If you need to include a literal backslash in a string, you use two backslashes.
2. **"\"":**
   This is used to include a literal double quote in a string. Normally, a double quote would end the string, but when preceded by a backslash, it is interpreted as part of the string.

3. **"\b"**:
   This represents a backspace.
4. **"\r"**:
   This represents a carriage return, which moves the cursor to the beginning of the current line.
5. **"\f"**:
   This represents a form feed, which is used to start a new page. It's not commonly used nowadays, but can still be found in some applications.
6. **"\a"**:
   This represents a bell or alert. In some systems, this can cause the computer to beep.
7. **"\v"**:
   This represents a vertical tab, which is used to align characters vertically.
8. **"\u"** or **"\U"**:
   These are used to represent Unicode characters. They are followed by a series of hex digits that specify the Unicode code point. For example, "\u03B1" represents the Greek letter alpha.

Using message() Function

In R, the message() function primarily displays informational messages, warnings, or diagnostic information about the code being executed. Unlike the print() and cat() functions, message() sends its output to the standard error stream, which means that the output from this function can be separated from the normal output. This can be useful for logging and for situations where you want to display additional information that is not part of the main results of your script.

Here's an example demonstrating the use of the message() function to display an informational message:

```
# Using the message() function to display an informational message
x <- 10
y <- 0

# Displaying a message before performing a division
message("Performing division...")

# Checking for division by zero
if (y != 0) {
  result <- x / y
  print(result)
} else {
  message("Division by zero is not allowed.")
}
```

Output:

```
Performing division...

Division by zero is not allowed.
```

Note that the messages "Performing division..." and "Division by zero is not allowed." are informational and do not constitute the main output of the program.

Inbuilt Functions in R

Mathematical Functions: R can also be used as a calculator along with facility to use many mathematical functions. Ex: sqrt, abs, floor, ceiling etc.

Trigonometric Functions: R provides the user an ability to compute the result using different trigonometric functions. Ex: sin, cos, tan etc.

Logarithmic Functions: R has an extensive facility to provide log of a number with proper specification of the base . Ex : log with base 10 and natural base

Date and Time Functions: Dates and times have special classes in R that allow for numerical and statistical calculations.

Sequence Function: A sequence is a set of related numbers, events, date etc. that follow each other in a particular order. R has a number of facilities for generating commonly used sequences of numbers.

Repeat Function: Function rep is used to replicates the values in a vector. It is a very powerful feature in R which helps the user to create a set of values in an easy manner

- Mathematical Functions
- Statistical Probability Functions
- String Functions

Mathematical Functions

R functions enable mathematical operations on numbers, like **finding the highest** or **lowest number, computing square roots, absolute values, rounding**, and more.

1. min() and max()

It returns the smallest value in a vector or collection of values.

Example:

```
values <- c(3, 12, 15, 1, 18, 21)

min_value <- min(values)

print(min_value)
```

Output:

```
[1] 1
```

Similarly, the max() function returns the largest value in a vector or collection of values.

Example:

```
values <- c(3, 12, 15, 1, 18, 21)

max_value <- max(values)

print(max_value)
```

Output:

```
[1] 21
```

2. sum()

The **sum()** function in R calculates the sum of all the values in a vector.

Example:

```
values <- c(3, 12, 15, 1, 18, 21)

total_sum <- sum(values)

print(total_sum)
```

Output:

```
[1] 70
```

3. mean()

It computes the average (mean) of a vector, i.e., It adds up all the values present in the vector and divides the total by the number of values.

Example:

```
values <- c(3, 12, 15, 1, 18, 21)

avg <- mean(values)

print(avg)
```

Output:

4. sqrt()

It computes the square root of a number or a vector of numbers.

Example:

```
x <- 36
```

```
sq_root <- sqrt(x)

print(sq_root)
```

Output:

```
[1] 6
```

5. abs()

It returns the absolute value of a number or a vector of numbers.

Example:

```
x <- -15

abs_value <- abs(x)

print(abs_value)
```

Output:

```
[1] 15
```

6. ceiling()

The ceiling() function rounds up a number to the nearest integer greater than or equal.

Example:

```
x <- 8.7

ce_value <- ceiling(x)

print(ce_value)
```

Output:

```
[1] 9
```

7. floor()

The floor() function rounds down a number to the nearest integer less than or equal.

```
x <- 5.8

floor_value <- floor(x)

print(floor_value)
```

Output:

```
[1] 5
```

8. trunc()

It removes all decimal places from a number and reduces it to its integer portion.

Example:

Output:

```
[1] 4
```

9. round()

The round() function rounds an integer to the number of decimal places provided.

Example:

```
x <- 3.14266

rnd_value <- round(x, digits = 2)

print(rnd_value)
```

Output:

```
[1] 3.14
```

10. cos(), sin(), tan()

The cos(), sin(), and tan() functions compute the cosine, sine, and tangent of an angle in radians, respectively.

Example:

```
angle <- 45

cosine <- cos(angle)

sine <- sin(angle)

tangent <- tan(angle)

print(cosine)

print(sine)

print(tangent)
```

Output:

```
> print(cosine)

[1] 0.525322

> print(sine)

[1] 0.8509035
```

```
> print(tangent)

[1] 1.619775
```

11. log() and log10()

It computes the natural logarithm of a number or a vector of numbers.

Output:

```
[1] 2.302585
```

Similarly, the log10() function computes the base-10 logarithm of a number or a vector of numbers.

Example:

```
log10_val <- log10(100)

print(log10_val)
```

Output:

```
[1] 2
```

12. exp()

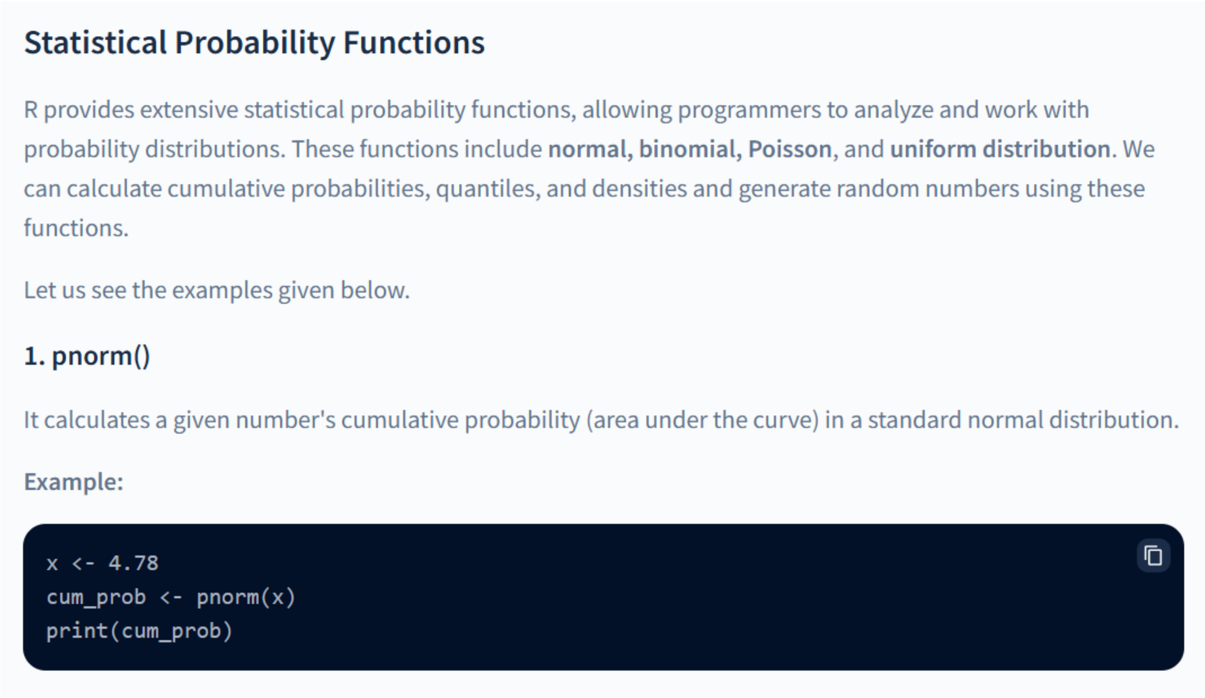
The exp() function calculates the exponential value (e raised to the power of x) for a number or a vector of numbers.

Example:

```
x <- 2

exp_value <- exp(x)

print(exp_value)
```

Output:

```
[1] 7.389056
```



Statistical Probability Functions

R provides extensive statistical probability functions, allowing programmers to analyze and work with probability distributions. These functions include **normal, binomial, Poisson**, and **uniform distribution**. We can calculate cumulative probabilities, quantiles, and densities and generate random numbers using these functions.

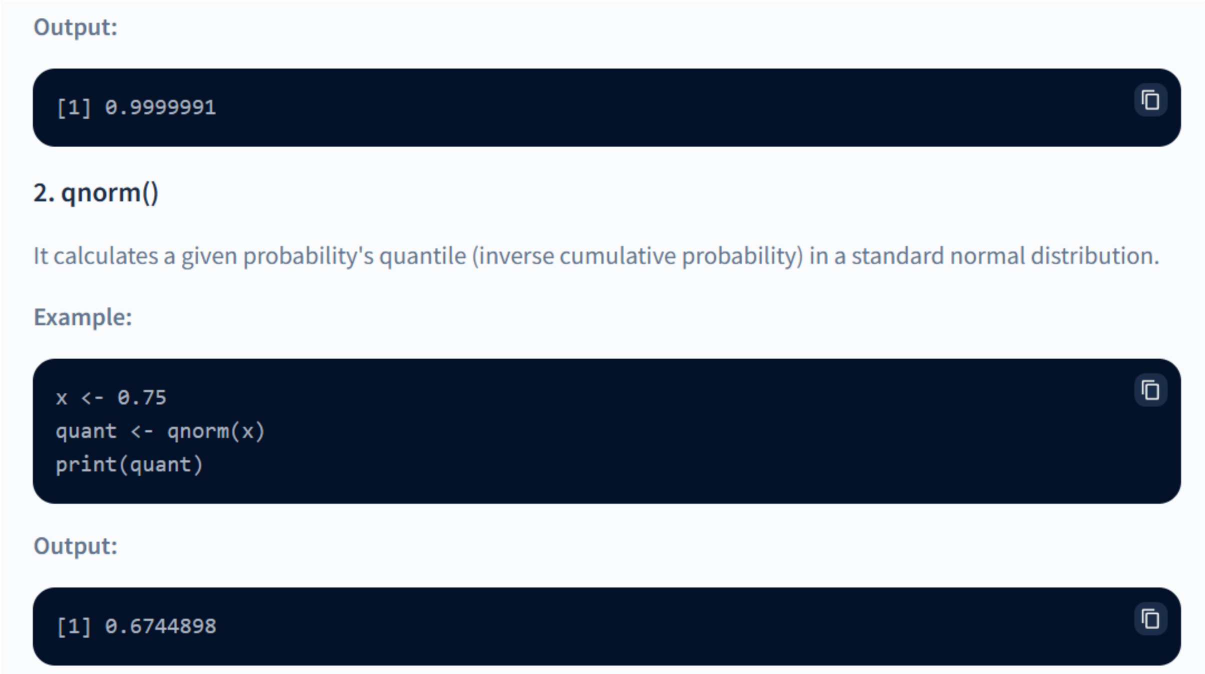
Let us see the examples given below.

1. pnorm()

It calculates a given number's cumulative probability (area under the curve) in a standard normal distribution.

Example:

```
x <- 4.78
cum_prob <- pnorm(x)
print(cum_prob)
```



Output:

```
[1] 0.9999991
```

2. qnorm()

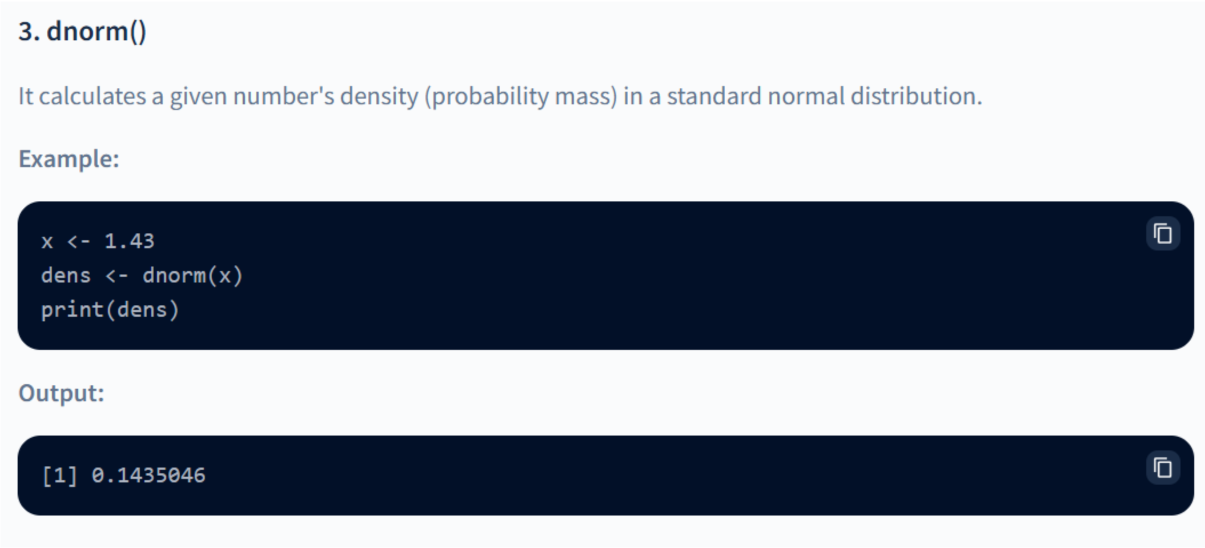
It calculates a given probability's quantile (inverse cumulative probability) in a standard normal distribution.

Example:

```
x <- 0.75
quant <- qnorm(x)
print(quant)
```

Output:

```
[1] 0.6744898
```



3. dnorm()

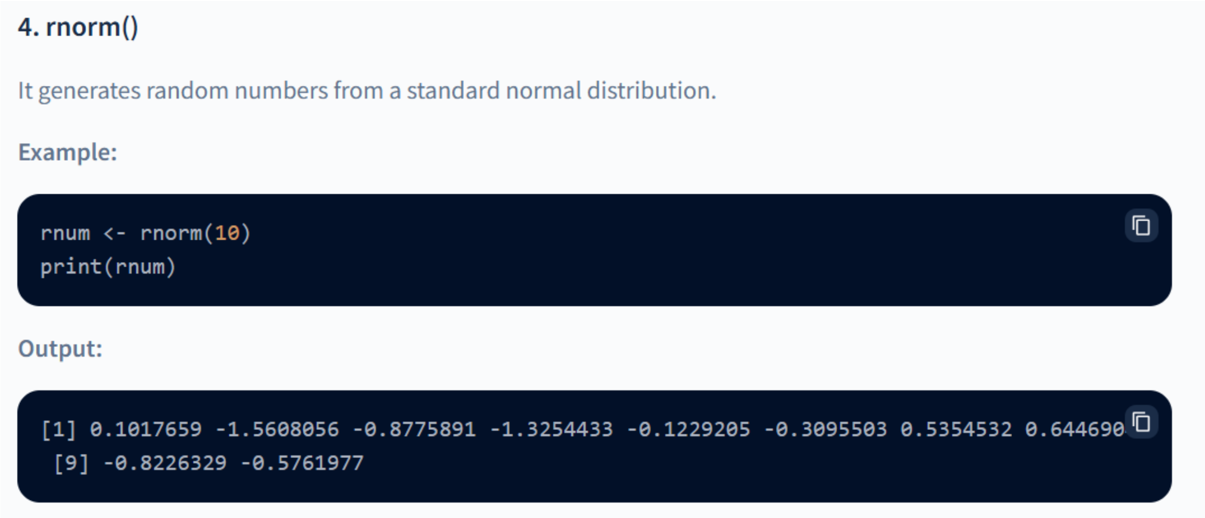
It calculates a given number's density (probability mass) in a standard normal distribution.

Example:

```
x <- 1.43
dens <- dnorm(x)
print(dens)
```

Output:

```
[1] 0.1435046
```



4. rnorm()

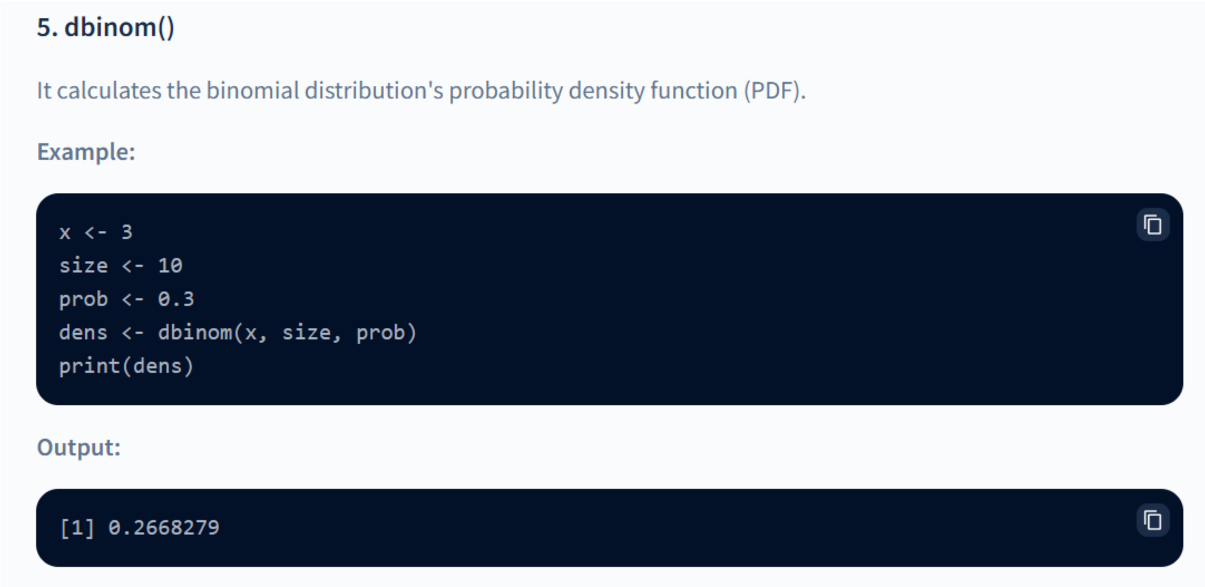
It generates random numbers from a standard normal distribution.

Example:

```
rnum <- rnorm(10)
print(rnum)
```

Output:

```
[1] 0.1017659 -1.5608056 -0.8775891 -1.3254433 -0.1229205 -0.3095503 0.5354532 0.644690
 [9] -0.8226329 -0.5761977
```



5. dbinom()

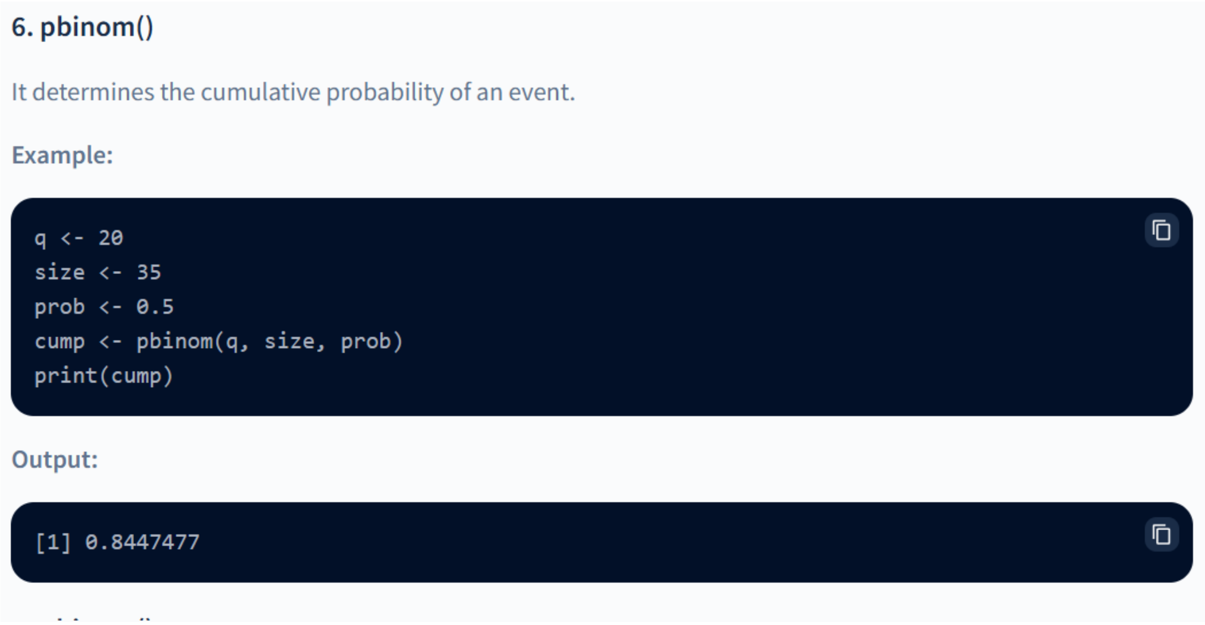
It calculates the binomial distribution's probability density function (PDF).

Example:

```
x <- 3
size <- 10
prob <- 0.3
dens <- dbinom(x, size, prob)
print(dens)
```

Output:

```
[1] 0.2668279
```



6. pbinom()

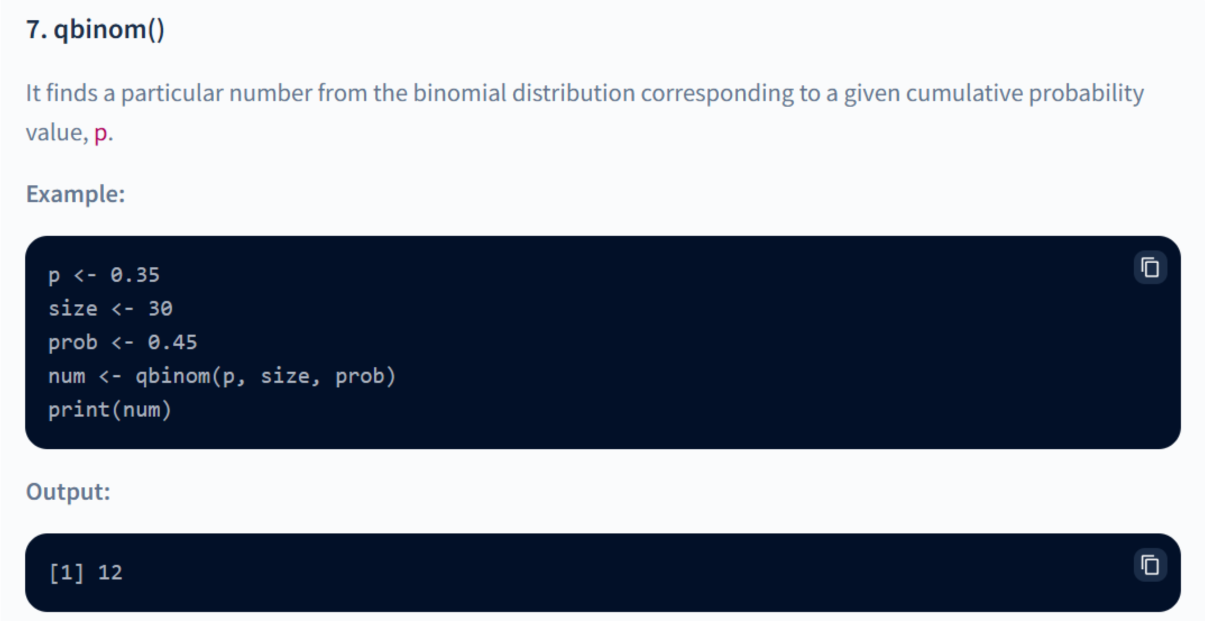
It determines the cumulative probability of an event.

Example:

```
q <- 20
size <- 35
prob <- 0.5
cump <- pbinom(q, size, prob)
print(cump)
```

Output:

```
[1] 0.8447477
```



7. qbinom()

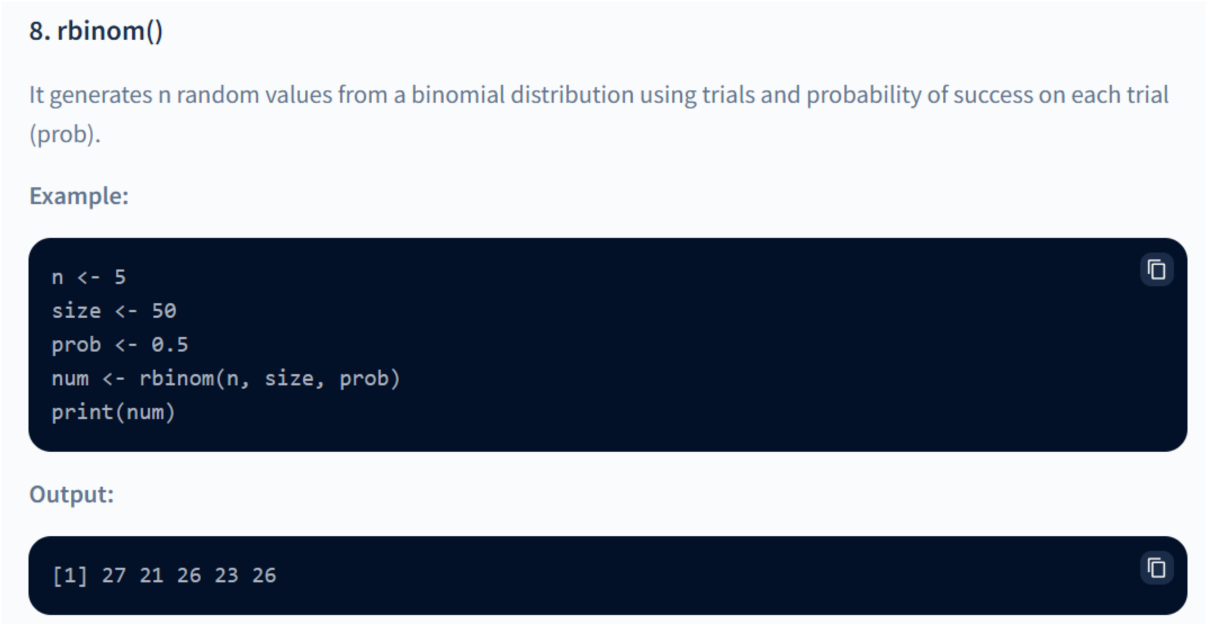
It finds a particular number from the binomial distribution corresponding to a given cumulative probability value, p.

Example:

```
p <- 0.35
size <- 30
prob <- 0.45
num <- qbinom(p, size, prob)
print(num)
```

Output:

```
[1] 12
```



8. rbinom()

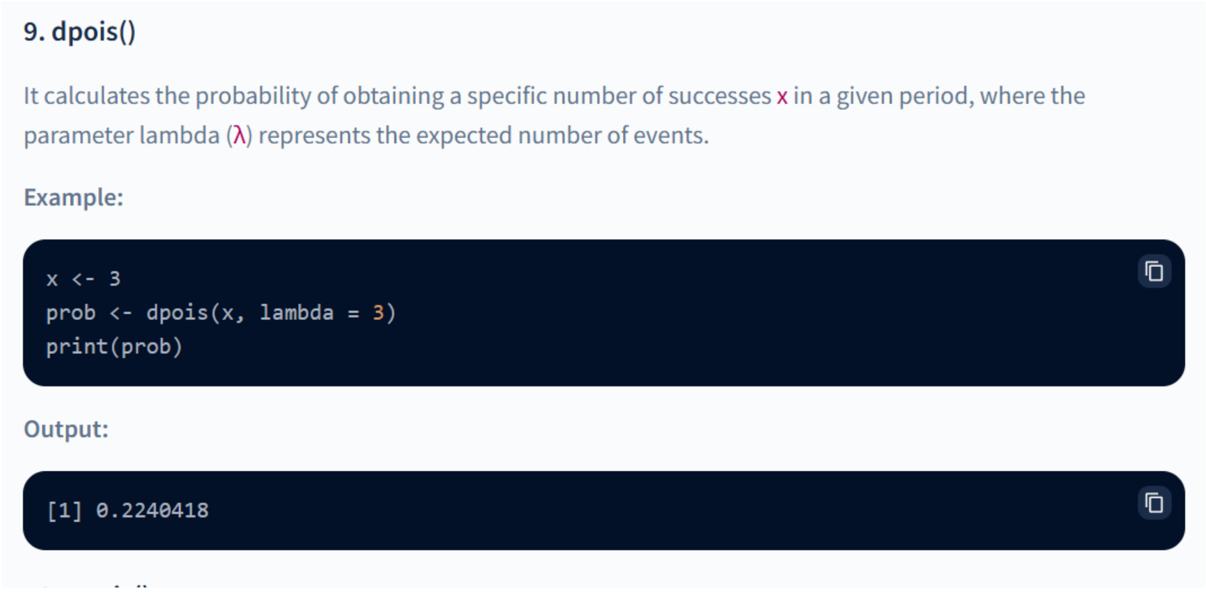
It generates n random values from a binomial distribution using trials and probability of success on each trial (prob).

Example:

```
n <- 5
size <- 50
prob <- 0.5
num <- rbinom(n, size, prob)
print(num)
```

Output:

```
[1] 27 21 26 23 26
```



9. dpois()

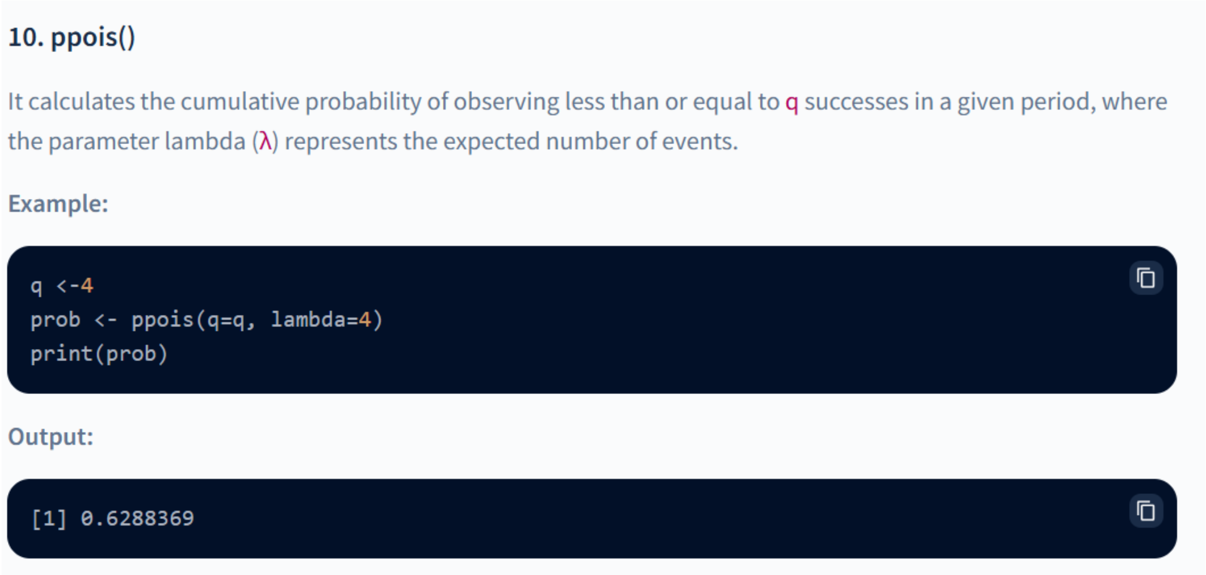
It calculates the probability of obtaining a specific number of successes x in a given period, where the parameter lambda (λ) represents the expected number of events.

Example:

```
x <- 3
prob <- dpois(x, lambda = 3)
print(prob)
```

Output:

```
[1] 0.2240418
```



10. ppois()

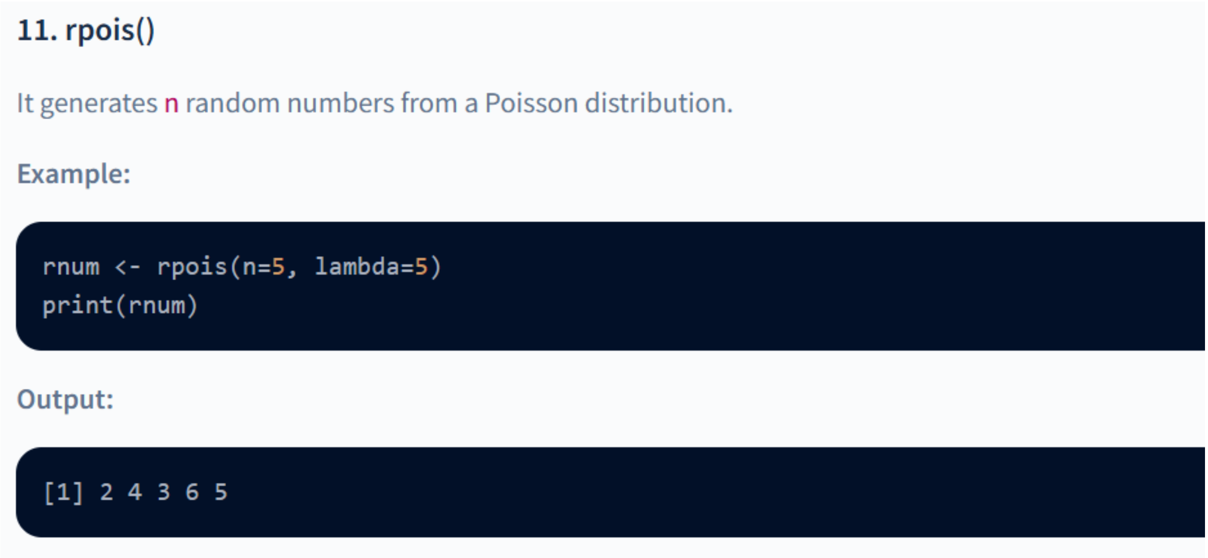
It calculates the cumulative probability of observing less than or equal to q successes in a given period, where the parameter lambda (λ) represents the expected number of events.

Example:

```
q <-4
prob <- ppois(q=q, lambda=4)
print(prob)
```

Output:

```
[1] 0.6288369
```



11. rpois()

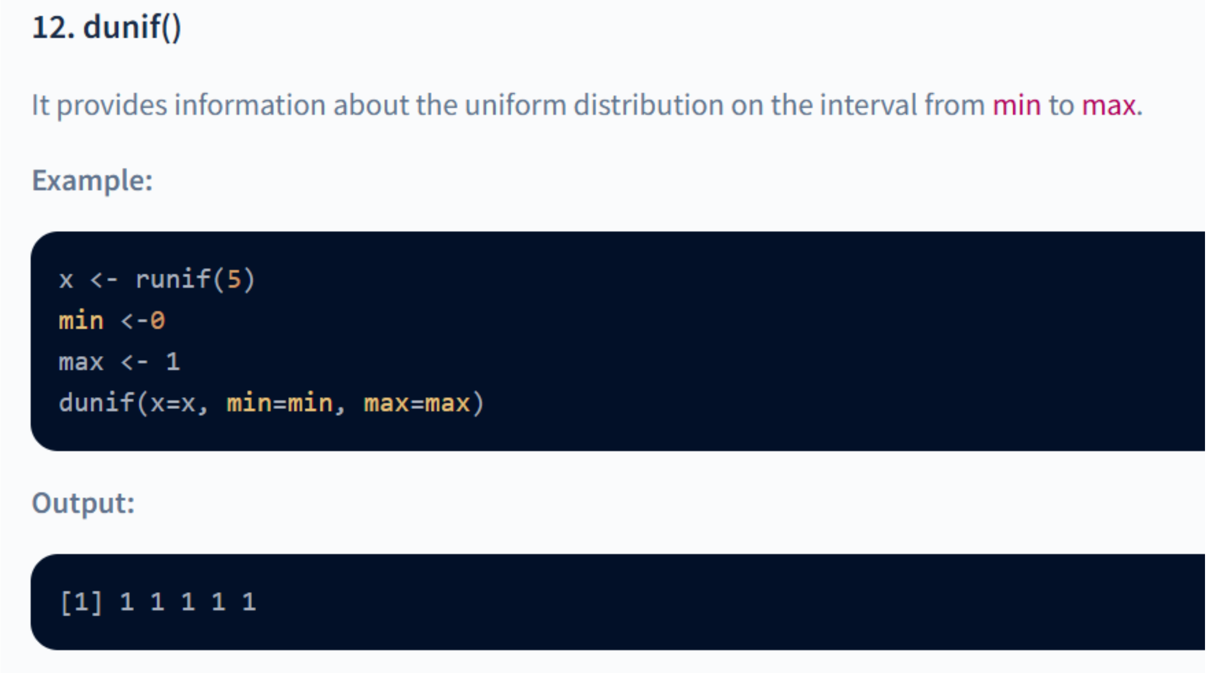
It generates n random numbers from a Poisson distribution.

Example:

```
rnum <- rpois(n=5, lambda=5)
print(rnum)
```

Output:

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



12. dunif()

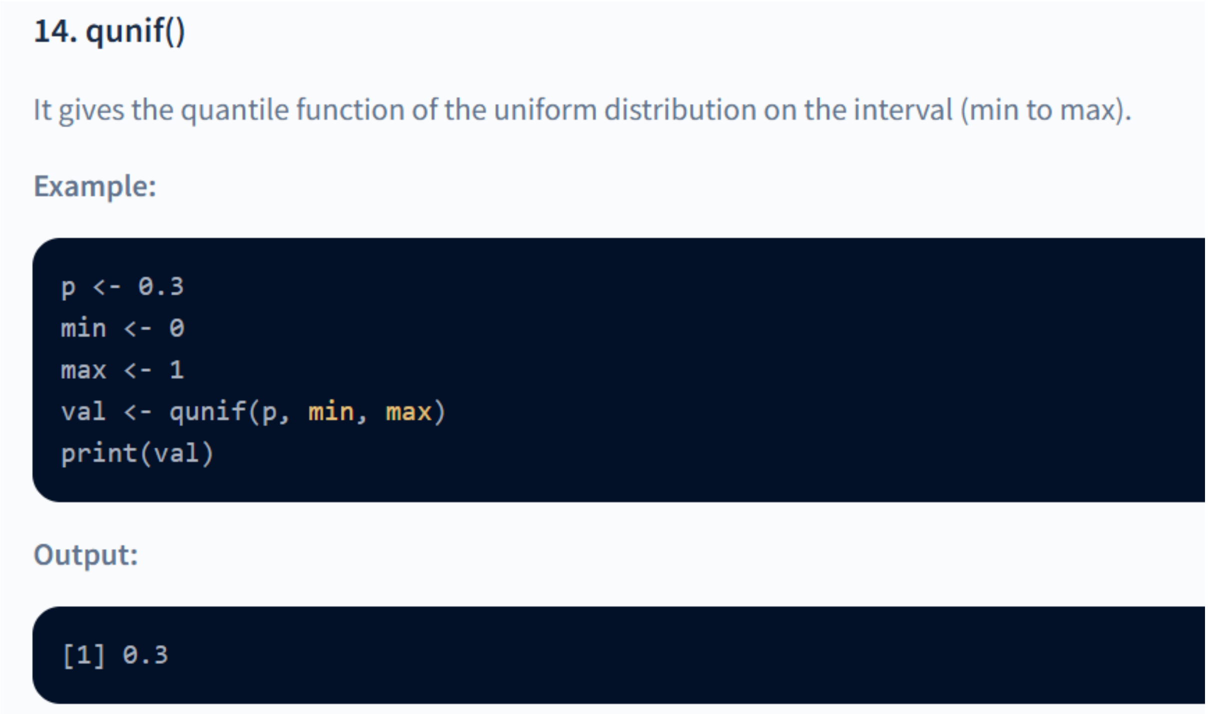
It provides information about the uniform distribution on the interval from min to max.

Example:

```
x <- runif(5)
min <-0
max <- 1
dunif(x=x, min=min, max=max)
```

Output:

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



14. qunif()

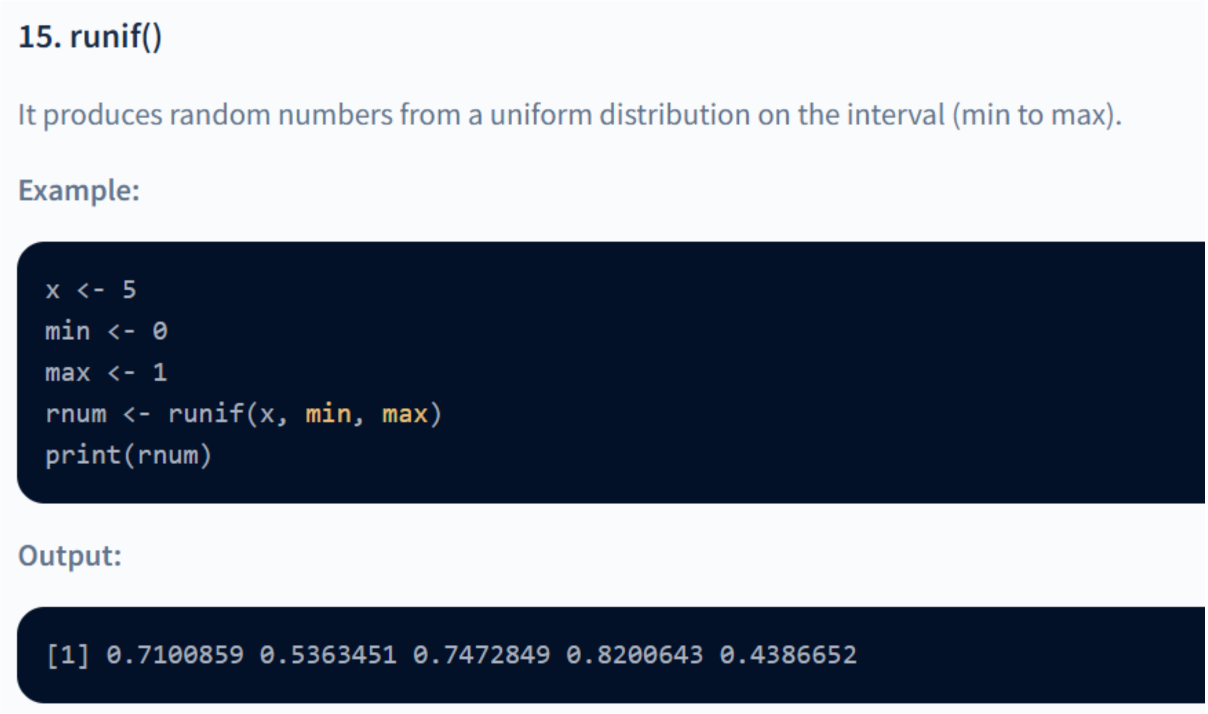
It gives the quantile function of the uniform distribution on the interval (min to max).

Example:

```
p <- 0.3
min <- 0
max <- 1
val <- qunif(p, min, max)
print(val)
```

Output:

```
[1] 0.3
```



15. runif()

It produces random numbers from a uniform distribution on the interval (min to max).

Example:

```
x <- 5
min <- 0
max <- 1
rnum <- runif(x, min, max)
print(rnum)
```

Output:

```
[1] 0.7100859 0.5363451 0.7472849 0.8200643 0.4386652
```

Strings

Creating a String: String in R is written within a pair of single quote or double quotes. Concatenating Strings: The paste() function concatenates several strings together. It creates a new string by joining the given strings end to end.

Formatting of Strings: Strings can be formatted to a specific style according to the requirement of the user using format() function.

Counting number of character: nchar() function is used to count the number of characters including spaces in a string.

Change case: The functions toupper() and tolower() functions are used to change the case of characters of a string.

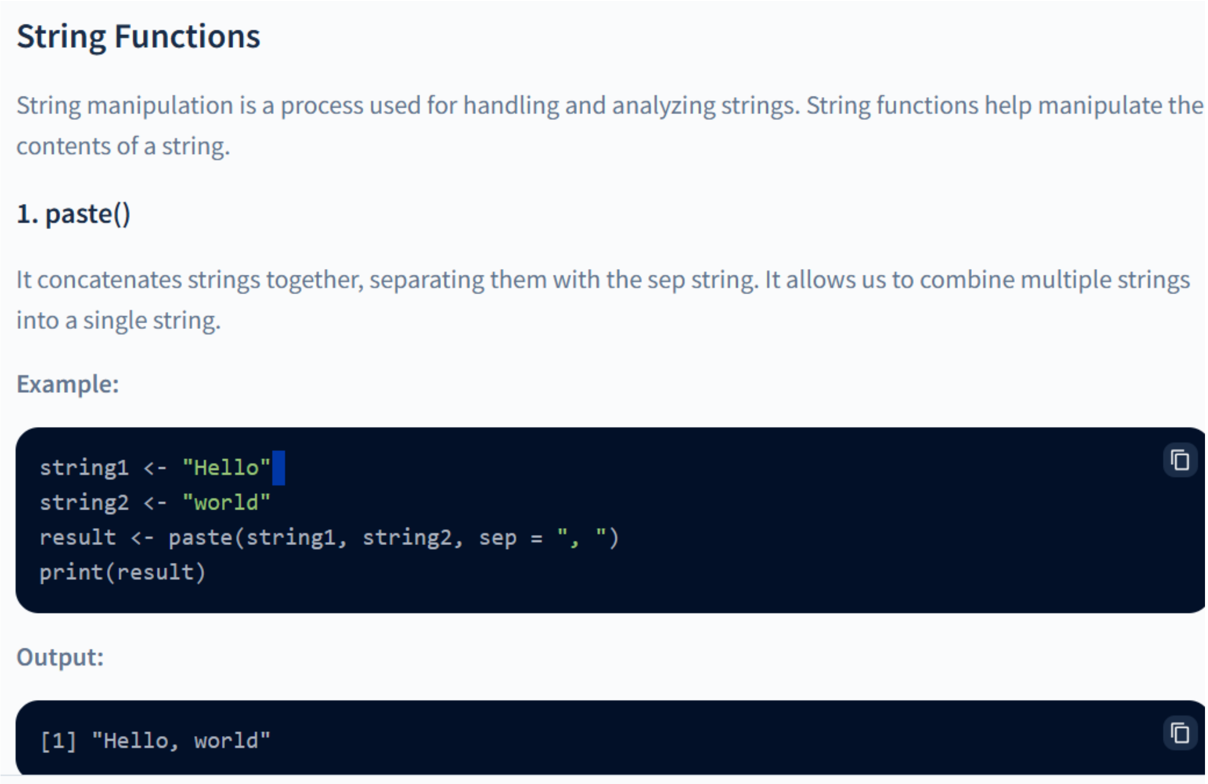
Extracting parts of a string: The substring() or substr() function extracts parts of a string depending on the index position of the string.

Searching Matches: The grep() function is used for searching the matches.

Changing String to expression: The eval() function evaluates an expression only and not a string..

Split the Elements of Vector: The function strsplit() is used to split the elements of a character vector into substrings according to the matches to substring split within them.

Replace substring with other: sub (substring) and gsub (global substring) functions are used if we want to replace one substring with another with in a string.



String Functions

String manipulation is a process used for handling and analyzing strings. String functions help manipulate the contents of a string.

1. paste()

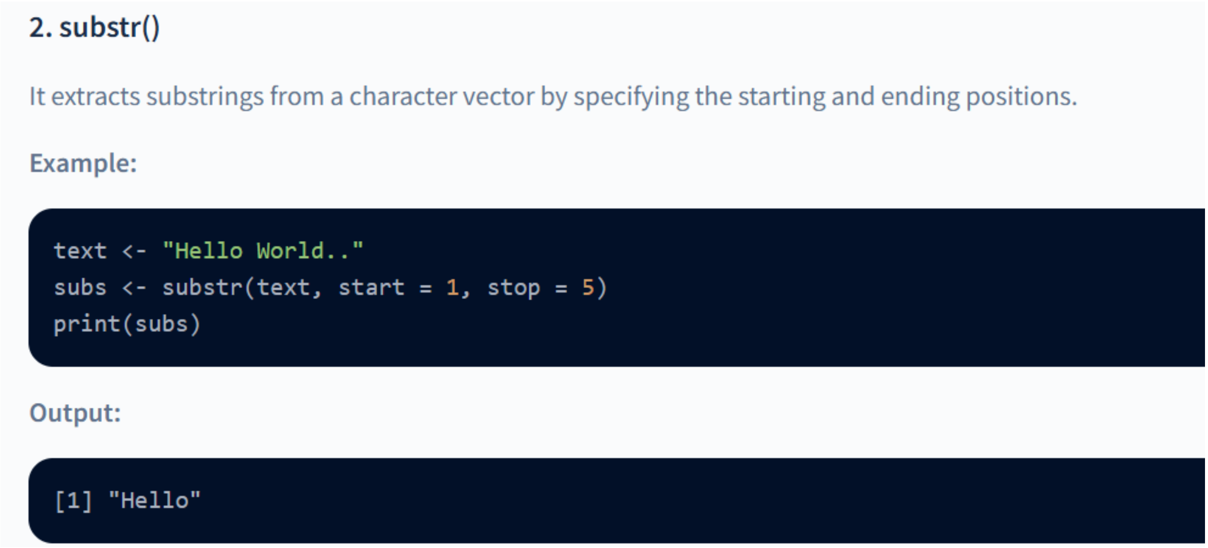
It concatenates strings together, separating them with the sep string. It allows us to combine multiple strings into a single string.

Example:

```
string1 <- "Hello"
string2 <- "world"
result <- paste(string1, string2, sep = ", ")
print(result)
```

Output:

```
[1] "Hello, world"
```



2. substr()

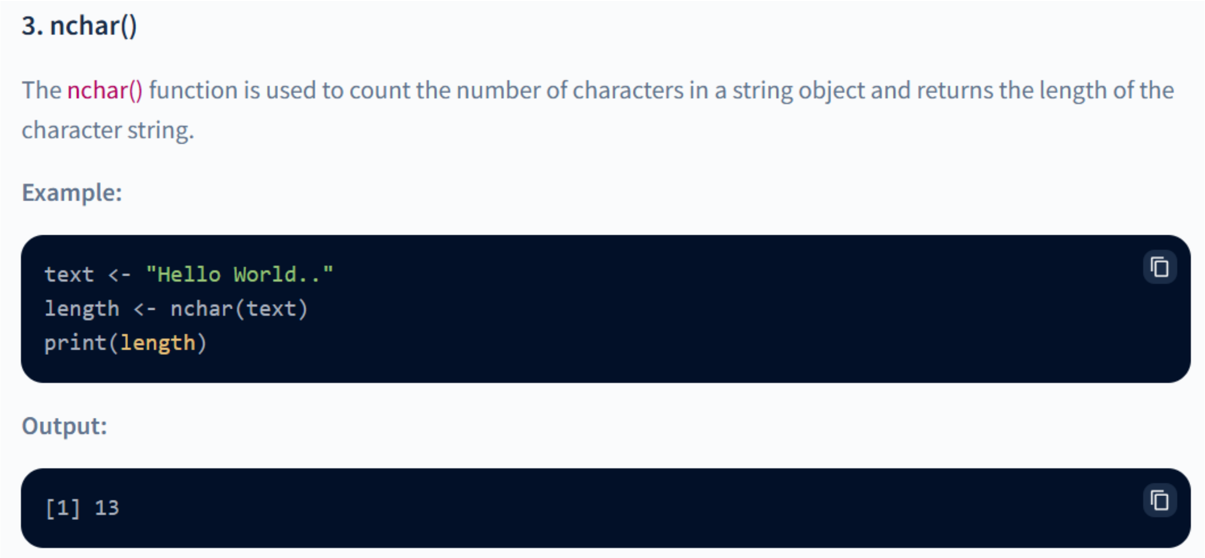
It extracts substrings from a character vector by specifying the starting and ending positions.

Example:

```
text <- "Hello World.."
subs <- substr(text, start = 1, stop = 5)
print(subs)
```

Output:

```
[1] "Hello"
```



3. nchar()

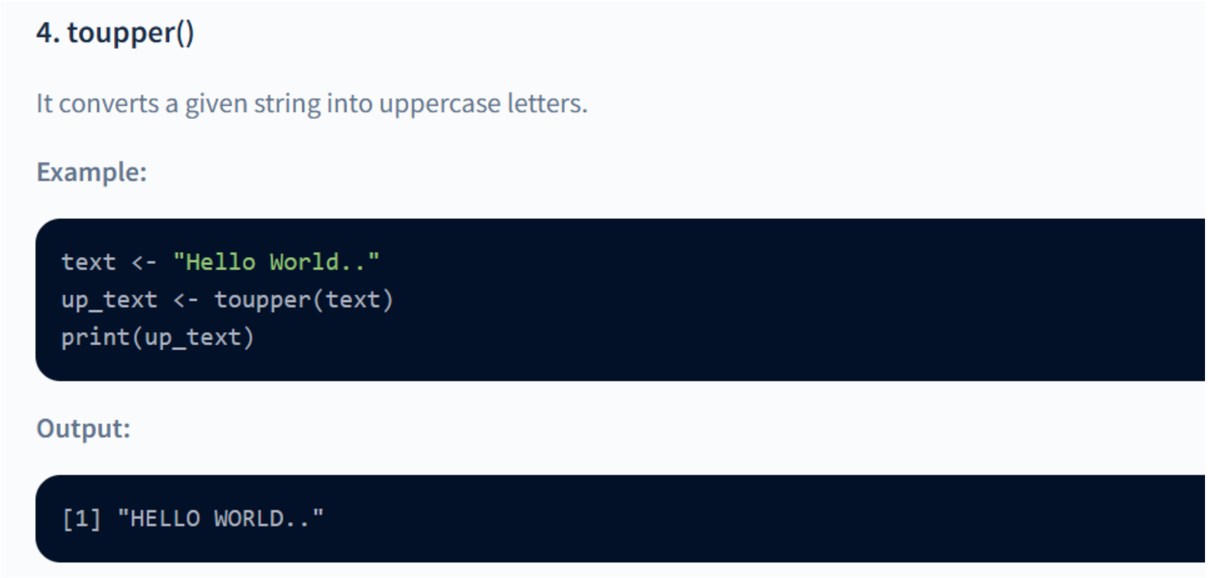
The nchar() function is used to count the number of characters in a string object and returns the length of the character string.

Example:

```
text <- "Hello World.."
length <- nchar(text)
print(length)
```

Output:

```
[1] 13
```



4. toupper()

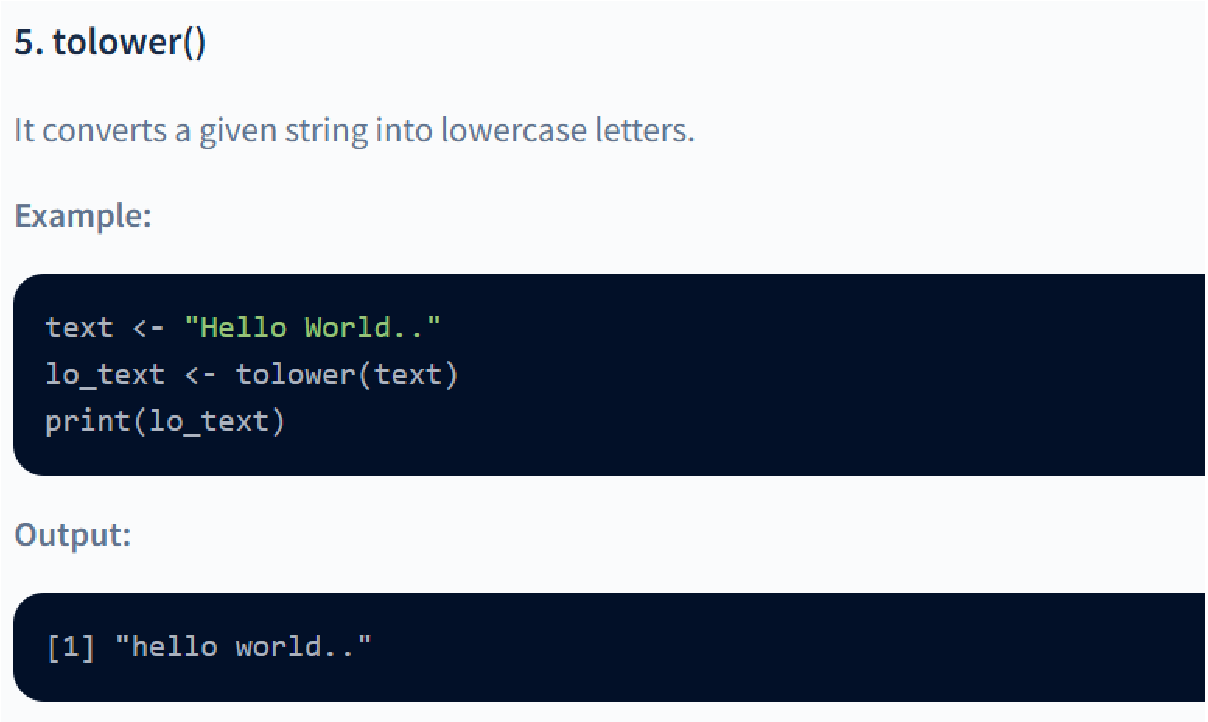
It converts a given string into uppercase letters.

Example:

```
text <- "Hello World.."
up_text <- toupper(text)
print(up_text)
```

Output:

```
[1] "HELLO WORLD.."
```



5. tolower()

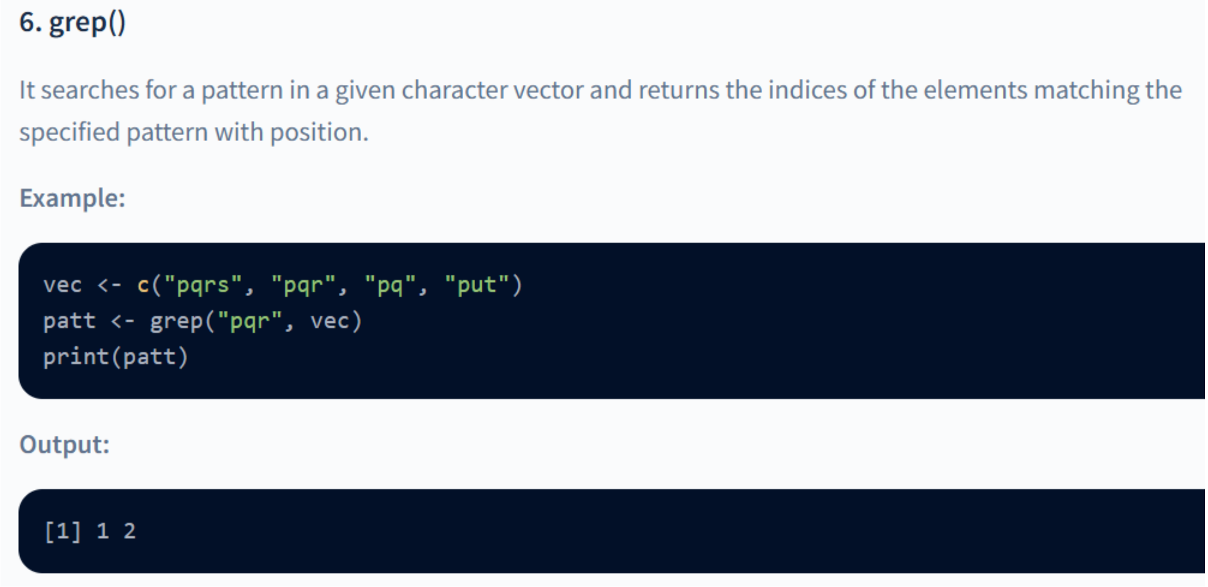
It converts a given string into lowercase letters.

Example:

```
text <- "Hello World.."
lo_text <- tolower(text)
print(lo_text)
```

Output:

```
[1] "hello world.."
```



6. grep()

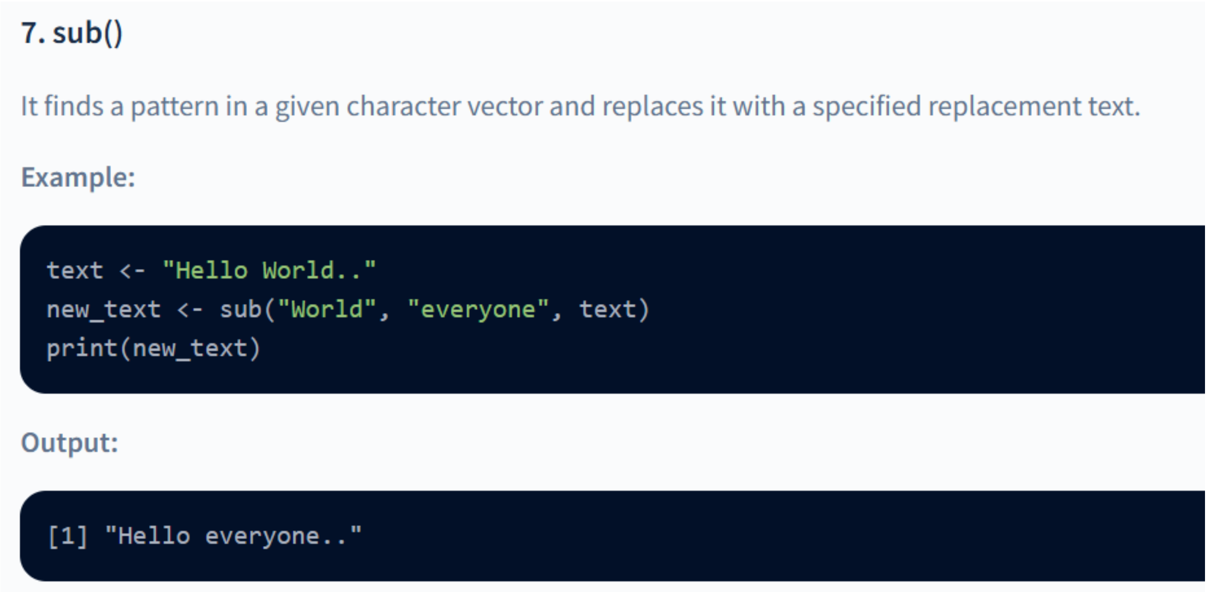
It searches for a pattern in a given character vector and returns the indices of the elements matching the specified pattern with position.

Example:

```
vec <- c("pqrs", "pqr", "pq", "put")
patt <- grep("pqr", vec)
print(patt)
```

Output:

```
[1] 1 2
```



7. sub()

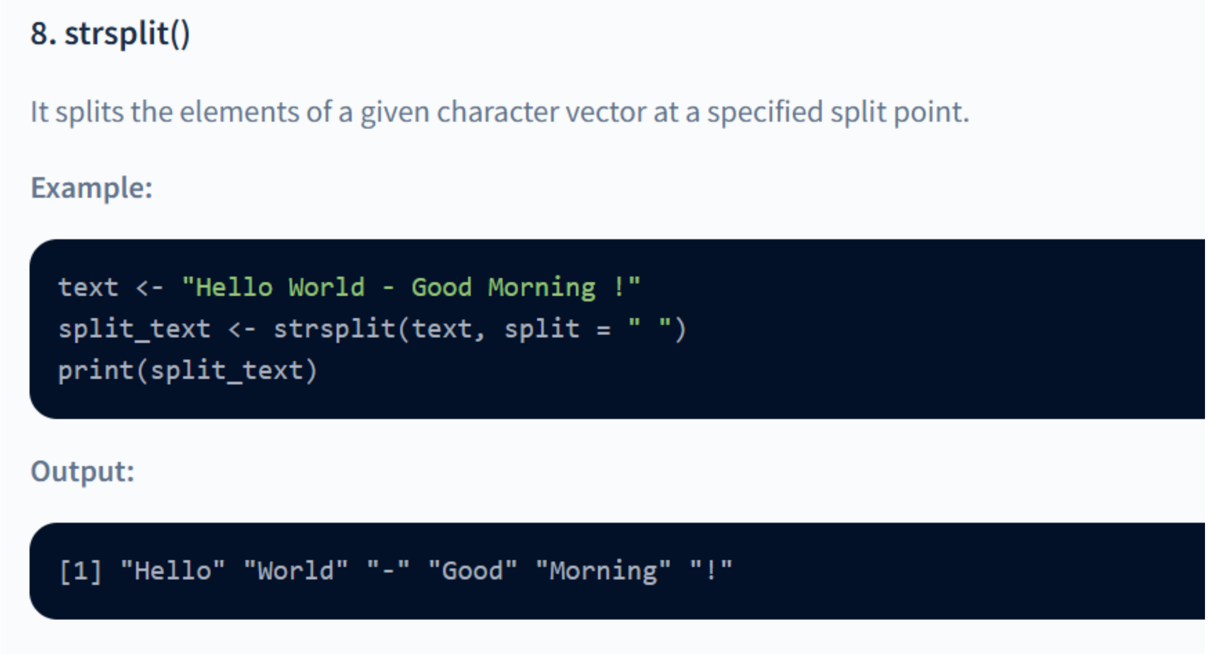
It finds a pattern in a given character vector and replaces it with a specified replacement text.

Example:

```
text <- "Hello World.."
new_text <- sub("World", "everyone", text)
print(new_text)
```

Output:

```
[1] "Hello everyone.."
```



8. strsplit()

It splits the elements of a given character vector at a specified split point.

Example:

```
text <- "Hello World - Good Morning !"
split_text <- strsplit(text, split = " ")
print(split_text)
```

Output:

```
[1] "Hello" "World" "-" "Good" "Morning" "!"
```

Packages in R

R packages are a collection of R functions, complied code and sample data. They are stored under a directory called "library" in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

Packages in R

packages in library ‘C:/Program Files/R/R-3.2.2/library’:

base   The R Base Package

```
boot            Bootstrap Functions (Originally by Angelo Canty
                for S)
class           Functions for Classification
cluster         "Finding Groups in Data": Cluster Analysis
                Extended Rousseeuw et al.
codetools       Code Analysis Tools for R
compiler        The R Compiler Package
datasets        The R Datasets Package
foreign         Read Data Stored by 'Minitab', 'S', 'SAS',
                'SPSS', 'Stata', 'Systat', 'Weka', 'dBase', ...
graphics        The R Graphics Package
grDevices       The R Graphics Devices and Support for Colours
                and Fonts
grid            The Grid Graphics Package
KernSmooth      Functions for Kernel Smoothing Supporting Wand
                & Jones (1995)
lattice         Trellis Graphics for R
MASS            Support Functions and Datasets for Venables and
                Ripley's MASS
Matrix          Sparse and Dense Matrix Classes and Methods
methods         Formal Methods and Classes
mgcv            Mixed GAM Computation Vehicle with GCV/AIC/REML
                Smoothness Estimation
nlme            Linear and Nonlinear Mixed Effects Models
nnet            Feed-Forward Neural Networks and Multinomial
                Log-Linear Models
parallel        Support for Parallel computation in R
rpart           Recursive Partitioning and Regression Trees
spatial         Functions for Kriging and Point Pattern
                Analysis
splines         Regression Spline Functions and Classes
stats           The R Stats Package
stats4          Statistical Functions using S4 Classes
survival        Survival Analysis
tcltk           Tcl/Tk Interface
tools           Tools for Package Development
utils           The R Utils Package
```

UNIT -II

UNIT-II Data Types & Functions Classes: 11

Data Types of R-Vectors, Matrices, Arrays, Lists, Factors, Data Frame-Basic Expressions Vectors – sequences, lengths, names, indexing vectors, vector recycling and repetition, Matrices and Arrays – creating arrays and matrices, rows, columns, dimensions, indexing arrays, Arithmetic. Decision-Making Structures – Loops, User- Defined Functions, User-Defined Package, Reports using Rmarkdown, Conditional – if and else, vectorized if,
multiple selection, Loops – repeat loops, while loops, for loops, Advanced looping – replication, looping over lists, looping over arrays, Multiple – input apply, instant vectorization.

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

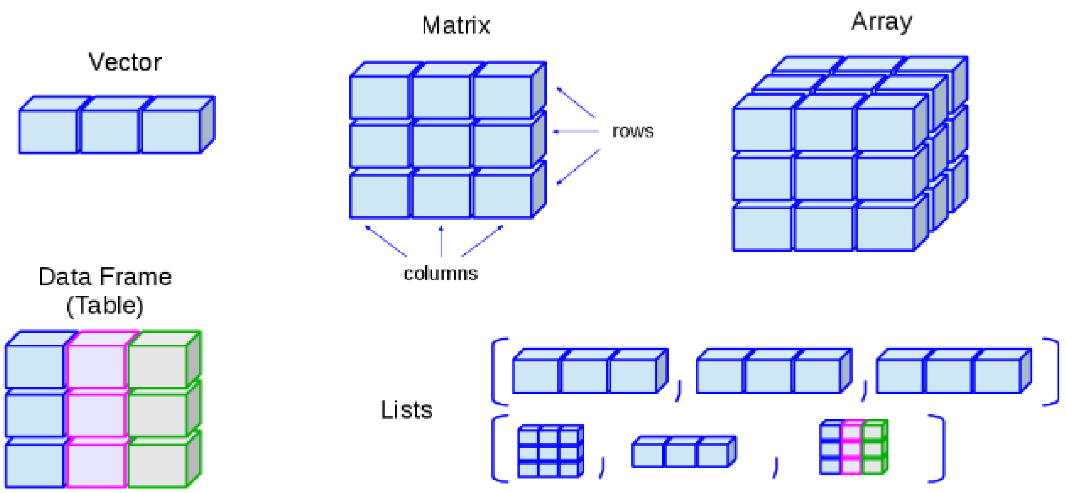
□ Arrays

□ 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.

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.



Vector

We start by looking at vectors. We have already seen vectors several times before. Vectors are a way to store one type of data in a certain variable. A simple vector we can create is a variable with a number in it. For example, if we assign the number 10 to the variable h and then check if h is a vector with the function is.vector() we will see that the result is TRUE.

```
h <- (10)
is.vector(h)

## [1] TRUE
```

Furthermore, we have already seen that if we want to store multiple numbers in a vector then we can use the c() function.

```
i <- c(10, 17, 25, 41)
is.vector(i)
```

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

Furthermore, we can also create vectors by using the “:” sign. For example, if we want to create a vector with the numbers 1 to 50 (or even more) you can imagine that it will take quite some time if we have to enter them ourselves with the c() function. If we want to do this we can also type the following code:

```
j <- 1:50
j
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

And this is also an example of a vector.

We can also make vectors with only characters. For example, if we want to make a vector with the names of students we can use the c() function again.

```
k <- c("Peter", "Sarah", "Michiel", "Jimmy")
is.vector(k)
```

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

Matrix

We can create matrices in R by using the matrix() function. If we want to create a simple 2 by 4 matrix with the numbers 1 through 8 and we want to assign it to the variable example_matrix we can type the following code:

```
example_matrix <- matrix(1:8, nrow = 2, ncol = 4)
example_matrix
```

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

In the code above we created a vector with the numbers 1 to 8 by using the “:” sign and we also see 2 other arguments, namely nrow = 2 and ncol = 4. The nrow and ncol represent the number of rows and columns. For example, we could have also made a 4 by 2 matrix (4 rows and 2 columns) with the numbers 1 through 8. This can be done by specifying nrow = 4 and ncol = 2.

```
example_matrix2 <- matrix(1:8, nrow = 4, ncol = 2)
example_matrix2
```

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

In both examples, we see that the numbers 1 through 8 are filled column-wise. So if we look at example_matrix2 we see that the numbers 1 through 4 are placed in column 1 first and then the numbers 5 through 8 are placed in the 2nd column. An alternative would be to place the numbers 1 through 8 per row and we can do that by specifying a byrow argument.

```
matrixA <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
matrixA
```

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

Now we see that the numbers are filled in per row. If we hadn’t given the byrow = TRUE argument the numbers 1 through 3 would be placed in the first column instead of the first row.

We can also multiply matrices. For example, if we create another 3 by 3 matrix with the numbers 10 through 18 and call it matrixB:

```
matrixB <- matrix(10:18, nrow = 3, ncol = 3, byrow = TRUE)
matrixB
```

```
##      [,1] [,2] [,3]
## [1,]   10   11   12
## [2,]   13   14   15
## [3,]   16   17   18
```

Then we can multiply the matrices by using the * sign.

```
matrixA * matrixB
```

```
##      [,1] [,2] [,3]
## [1,]   10   22   36
## [2,]   52   70   90
## [3,]  112  136  162
```

The result is an element-wise multiplication of the matrices. This means that all numbers in the rows are multiplied with each other. So 10 is obtained by 1 * 10, 22 is obtained by 2 * 11, and so on. If we want matrix multiplication as we may remember it from linear algebra:

we can use the %*% sign.

```
matrixA %*% matrixB
```

```
##      [,1] [,2] [,3]
## [1,]   84   90   96
## [2,]  201  216  231
## [3,]  318  342  366
```

Finally, we can also test if something is a matrix by using the is.matrix() function. For example, if we do this with matrixA:

```
is.matrix(matrixA)
```

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

Array

An array is a data structure that can hold multi-dimensional data. In R, the array is objects that can hold two or more than two-dimensional data. For example, in square matrices can contain two rows and two columns and dimension can take five. Arrays can store the values having only a similar kind of data types. The data can be more than one dimensional, where there are rows and columns and dimensions of some length.

Creation of an Array

The array() function will create an array which takes a vector, which is the numbers and dimension('dim') in the argument.

Let's see an example below where two vectors named 'array1' and array2' are created.

```
vector1 =  c (5, 10, 15,20)

vector2 =  c (25, 30, 35, 40, 45, 50,55,60)
```

You can take two vectors above as an input to an array where the dimension is considered as 4 * 4, and two matrices or dimensional data is created.

```
final = array (c (array1, array2),dim =c(4,4,3))
print (final)
```

The output of the above code is below :

```
, , 1

     [,1] [,2] [,3] [,4]
[1,]    5   25   45    5
[2,]   10   30   50   10
[3,]   15   35   55   15
[4,]   20   40   60   20

, , 2

     [,1] [,2] [,3] [,4]
[1,]   25   45    5   25
[2,]   30   50   10   30
[3,]   35   55   15   35
[4,]   40   60   20   40

, , 3

     [,1] [,2] [,3] [,4]
[1,]   45    5   25   45
[2,]   50   10   30   50
[3,]   55   15   35   55
[4,]   60   20   40   60
```

How to access array elements with indexing

Using indexing, you can access elements in your arrays. Similar to other programming languages, you can do this by specifying the indices of the elements in square brackets. If you are working with multidimensional arrays, you can not only output individual elements but also entire rows or columns:

```
examplearray <- array(1:6, dim = c(2, 3))
# Access the element in the first row and the second column
element <- examplearray[1, 2]
# Access the first row
row <- examplearray[, 1]
# Access the first column
column <- examplearray[1, ]
```

How to do calculations with arrays

With R arrays, you can apply **various mathematical functions** to entire datasets. For example, you can calculate the sum of two arrays. This would be like adding two matrices together. Arrays should have the same dimensions or length. You can find out the length of an R array with the length() function.

```
array1 <- array(1:4, dim = c(2,2))

array2 <- array(5:8, dim = c(2,2))

result <- array1 + array2
```

In addition to the arithmetic operations that you can implement using operators in R, various functions in R are defined to work with arrays. These can help you complete different types of calculations. For example, if you want to calculate the average of all elements in an array, you can use the R command mean():

```
average <- mean(array1)
```

You can also apply a range of functions to the dimension of your choice using the R array function apply(array, MARGIN, FUN). This function accepts several parameters:

- array: Array to be considered/used
- MARGIN: Dimension that the function should be applied to, with 1 representing rows and 2 representing columns
- FUN: Vector function that returns a scalar result

Below is an example of how the function apply() can be used:

```
# Create an array

testarray <- array(1:6, dim = c(2,3))

# Use apply()

average_columns <- apply(array, MARGIN = 2, FUN = mean)

# Display the results

print(average_columns)
```

List

Lists are very similar to vectors. The only difference between vectors and lists is that we can store multiple data types within lists as opposed to vectors. To illustrate this difference we will take a look at the following example:

```
l <- c(1, 2, 3, "4")
l
```

```
## [1] "1" "2" "3" "4"
```

In this example, we tried to create a vector with the numbers 1 to 3 (numeric data type) and a character “4”. If we look at the output we see that R has also made the numbers 1 to 3 characters. The reason for this is that vectors are only able to store one data type. If we want to store multiple data types in a variable we can use lists.

So if we want to store the numbers 1 to 3 as numeric and the number 4 as a character we can do that by using the list() function to create a list.

```
l2 <- list(c(1, 2, 3), "4")
l2
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "4"
```

he output of the list now consists of 2 parts [[1]] and [[2]]. The first part contains our numbers 1 through 3 as numeric data type and the second part contains our character “4”.

We can also create lists by combining vectors of the same data type. For example, suppose we have 3 students in a class, we have the grades of a test, and whether the students passed or failed the test. Then we can create individual vectors with the names of the students, grades, and pass (TRUE) or fail (FALSE). Then, we can use the list() function to combine these vectors in a list.

```
names <- c("Sarah", "Hugo", "James")
grades <- c(5, 8, 9)
pass <- c(FALSE, TRUE, TRUE)

class1 <- list(names, grades, pass)
class1
```

```
## [[1]]
## [1] "Sarah" "Hugo"  "James"
##
## [[2]]
## [1] 5 8 9
##
## [[3]]
## [1] FALSE  TRUE  TRUE
```

Our list now consists of 3 parts and we can see that it contains the names of the students, the grades, and pass (TRUE) or failed (FALSE).

Factors

The factor data type factor is commonly used in statistical analyses. For example, if we have a dataset with Social Economic Status (SES), it may be coded as “Low”, “Average” and “High”. The only problem is that we can’t use that for statistical analysis because everything has to be coded as numbers if we want to be able to use it for analysis. The factor data type can be used for this. For example, if we have the variable SES with “low”, “average” and “high” and we look at what data type it is, we can see that it is a character because “low”, “average” and “high” are all written in double parentheses.

```
SES <- c("Low", "Average", "High")
class(SES)
## [1] "character"
```

Now what we can do is change this variable to a factor (Note: We can overwrite variables by assigning something else to the same variable). We can do this by using the as.factor() function.

```
SES <- as.factor(SES)
class(SES)
## [1] "factor"
levels(SES)
## [1] "Average" "High"    "Low"
```

Now we see thatthe SES variable has become a factor data type. This has assigned numbers or levels to the categories of low, average, and high and in this way, we can use them for statistical analysis. In addition to the as.factor function, we also have several other as. functions and these can change data types to other data types whenever that is possible.

For example, if we have a vector with the numbers 1 to 7 and this is stored as a character data type (in parentheses " "), then we can change it to the numeric data type by using the as.numeric() function.

```
f <- c("1", "2", "3", "4", "5", "6", "7")
f <- as.numeric(f)
f
## [1] 1 2 3 4 5 6 7
class(f)
## [1] "numeric"
```

But as mentioned earlier, we can only do that if it is logical, if we try to do it with, for example:

```
as.numeric("This is an example")
## Warning: NAs introduced by coercion
## [1] NA
```

Then we see a red error message because R cannot assign values to text.

We can use the as.numeric() function for logical data types as well. Accordingly, TRUE will be encoded as 1 and FALSE will be encoded as 0.

```
g <- c(TRUE, FALSE, TRUE, TRUE)
as.numeric(g)
## [1] 1 0 1 1
```

Certain functions in R will already do this automatically. For example, if we want to know the sum of the variable g with 3 times TRUE in it and one FALSE in it then R will automatically do this and return the result 3.

```
sum(g)
## [1] 3
```

Dataframe

We can also create data frames ourselves with the data.frame() function, but this is rarely done in practice. Generally, data frames are loaded by using, for example, SPSS or excel files. Later in the book, we will discuss data frames and loading data in greater detail. For the moment it’s useful to see a data frame once and know that we can create one similarly as we did with lists.

```
names <- c("Sarah", "Hugo", "James")
grades <- c(5, 8, 9)
pass <- c(FALSE, TRUE, TRUE)
example_dataframe <- data.frame(names, grades, pass)
example_dataframe
##   names grades  pass
## 1 Sarah      5 FALSE
## 2  Hugo      8  TRUE
## 3 James      9  TRUE
```

If we compare the output of this data frame to that of a list we see that with the list we only had one column of data and that they were separated with [[1]], [[2]], and so on. In contrast, a data frame can have multiple columns and rows of data. Again, we can also test if something is a data frame by using the is.data.frame() function.

```
is.data.frame(example_dataframe)
## [1] TRUE
```

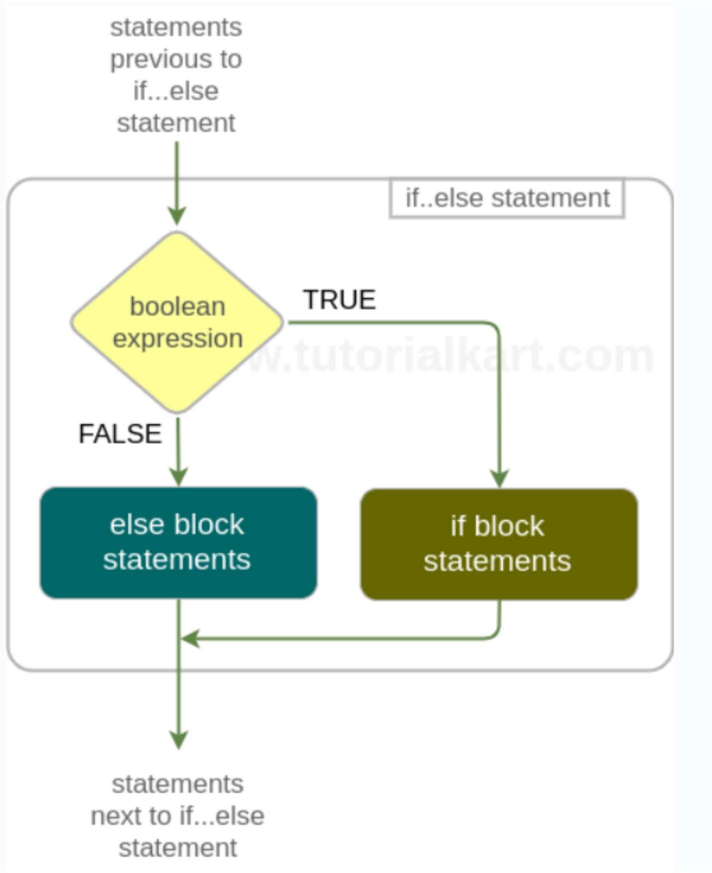
And the output shows that this is indeed a data frame.

Decision making statements in R

Decision making is a prime feature of any programming language. It allows us to make a decision, based on the result of a condition. Decision making is involved in order to change the sequence of the execution of statements, depending upon certain conditions.
A set of statements is provided to the program, along with a condition. Statements get executed only if the condition stands true, and, optionally, an alternate set of statements is executed if the condition becomes false.

R If Else

In this tutorial, we shall learn about R if…else statement, its Syntax and the Execution Flow in and around the if…else statement with an R Example Script.
if…else statement is an extension of if statement with an else block. So, if the condition provided to the if statement is true, then the statements in the if statement block are executed, else the statements in the else block are executed.
Following is a flow diagram depicting the flow of execution around and in an if..else statement.



Syntax – R if-else

The syntax of R if else statement is

```
if(boolean_expression){
    if_block_statements
 } else {
    else_block_statements
 }
```

The boolean_expression is any expression that evaluates to a boolean value. And if the boolean value = TRUE, execution flow enters the if block, else execution flow enters the else block.

Example 1 – R If-Else

In this example, we will write two if-else statements. For the first if-else statement, the condition evaluates to TRUE and therefore if block will be executed. For the second if-else statement, the condition evaluates to FALSE and therefore else block will be executed.

```
# R if..else statement Example

# for TRUE condition
```

```
a = 6

if(a==6){
   print ("Condition a==6 is TRUE")
   print ("This is second statement in if block")
} else{
   print ("Condition a==6 is FALSE")
   print ("This is second statement in else block")
}

# for FALSE condition
b = 7

if(b==6){
   print ("Condition b==6 is TRUE")
   print ("This is second statement in if block")
} else{
   print ("Condition b==6 is FALSE")
   print ("This is second statement in else block")
}
```

Output

```
$ Rscript r_if_else_example.R
[1] "Condition a==6 is TRUE"
[1] "This is second statement in if block"
[1] "Condition b==6 is FALSE"
[1] "This is second statement in else block"
```

Vectorised if-else

A Vectorized if-then-else: The ifelse() Function

In addition to the usual if-then-else construct found in most languages, R also includes a vectorized version, the ifelse() function. The form is as follows:

```
ifelse(b,u,v)
```

where b is a Boolean vector, and u and v are vectors.

The return value is itself a vector; element i is u[i] if b[i] is true, or v[i] if b[i] is false. The concept is pretty abstract, so let’s go right to an example:

```
> x <- 1:10
> y <- ifelse(x %% 2 == 0,5,12)  # %% is the mod operator

> y

 [1] 12  5 12  5 12  5 12  5 12  5
```

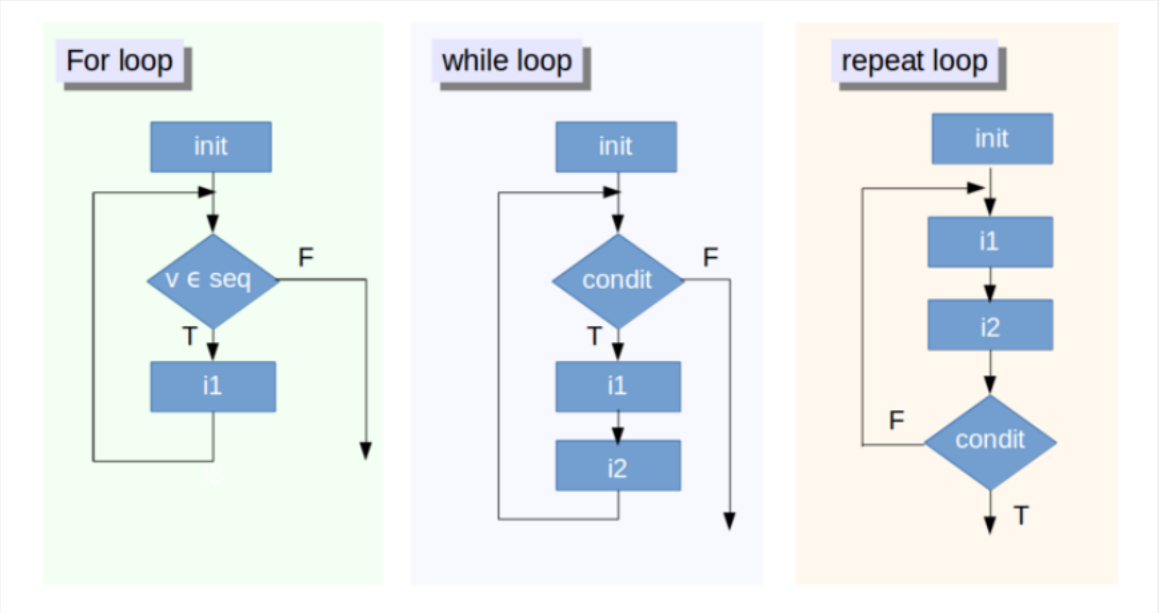
Loops : Repeat loop ,While loop & For loop

An Introduction to Loops in R

According to the R base manual, among the control flow commands, the loop constructs are for, while and repeat, with the additional clauses break and next.

Remember that control flow commands are the commands that enable a program to branch between alternatives, or to "take decisions", so to speak.

You can always see these control flow commands by invoking ?Control at the RStudio command line.



For Loops in R

The next sections will take a closer look at each of these structures that are shown in the figure above. We will start our discussion with the structure on the left, and we will continue the next sections by gradually moving to the structures on the right.

For a video introduction to for loops and a follow-up exercise, try **the For Loop chapter** in our intermediate R course.

For Loops Explained

This loop structure, made of the rectangular box ‘init’ (or initialization), the diamond or rhombus decision, and the rectangular box i1 is executed a known number of times.

In flowchart terms, rectangular boxes mean something like “do something which does not imply decisions”. Rhombi or diamonds, on the other hand, are called “decision symbols” and therefore translate into questions which only have two possible logical answers, namely, True (T) or False (F).

Note that, to keep things simple, other possible symbols have been omitted from the figure.

One or more instructions within the initialization rectangle are followed by the evaluation of the condition on a variable which can assume values within a specified sequence. In the figure, this is represented by the diamond: the symbols mean “does the variable v’s current value belong to the sequence seq?”.

In other words, you are testing whether v’s current value is within a specified range. You normally define this range in the initialization, with something like 1:100 to ensure that the loop starts.

If the condition is not met and the resulting outcome is False, the loop is never executed. This is indicated by the loose arrow on the right of the for loop structure. The program will then execute the first instruction found after the loop block.

If the condition is verified, an instruction -or block of instructions- i1 is executed. And perhaps this block of instructions is another loop. In such cases, you speak of a nested loop.

Once this is done, the condition is then evaluated again. This is indicated by the lines going from i1 back to the top, immediately after the initialization box. In R -and in Python, it is possible to express this in plain English, by asking whether our variable belongs to a range of values or not.

Note that in other languages, for example in C, the condition is made more explicit with the use of a logical operator, such as greater or less than, equal to, …

Here is an example of a simple for loop:

```
# Create a vector filled with random normal values

u1 <- rnorm(30)

print("This loop calculates the square of the first 10 elements of vector u1")


# Initialize `usq`

usq <- 0.


for (i in 1:10) {

   # i-th element of `u1` squared into `i`-th position of `usq`

   usq[i] <- u1[i] * u1[i]
```

```
  print(usq[i])
}

print(i)
```

The for block is contained within curly braces. These can be placed either immediately after the test condition or beneath it, preferably followed by an indentation. None of this is compulsory, but the curly braces definitely enhance the readability of your code and allow to spot the loop block and potential errors within it easily.

Note that the vector of the squares, usq, is initialized. This would not be necessary in plain RStudio code, but in the markup version, knitr would not compile because a reference to the vector is not found before its use in the loop, thus throwing an error within RStudio. For more information on knitr, go to **the R Markdown page**.

Nesting for Loops

Now that you know for loops can also be nested, you're probably wondering why and when you would be using this in your code.

Well, suppose you wish to manipulate a bi-dimensional array by setting its elements to specific values.

Then you might do something like this:

```
# Insert your own integer here
my_int <- 42

nr <- as.integer(my_int)

# Create a `n` x `n` matrix with zeros
mymat <- matrix(0, nr, nr)

# For each row and for each column, assign values based on position
# These values are the product of two indexes
for (i in 1:dim(mymat)[1]) {
  for (j in 1:dim(mymat)[2]) {
    mymat[i, j] = i * j
  }
}
```

You have two nested for loops in the code chunk above and thus two sets of curly braces, each with its own block and governed by its own index. That is, i runs over the lines and j runs over the columns.

What was produced?

Well, you made the very familiar multiplication table that you should know by heart.

You can also choose an integer and then produce a table according to your choice: you can assign an integer to a variable if the table is square or to two variables if the table is rectangular. This variable will then serve as upper bounds to the indexes i and j.

```
# Show the first 10x10 chunk or the first `nr` x `nr` chunk
if (nr > 10) {
   mymat[1:10, 1:10]
} else
   mymat[1:nr, 1:nr]
```

Note that to prevent the user from deluging the screen with huge tables, you put a condition at the end to print the first 10 x 10 chunk, only if the user asked for an integer greater than 10. Else, an n x n chunk will be printed.

The complete code looks like this:

```
# Insert your own integer here
my_int <- 42

nr <- as.integer(my_int)

# Create a `n` x `n` matrix with zeroes
mymat <- matrix(0, nr, nr)
# For each row and for each column, assign values based on position
# These values are the product of two indexes
for (i in 1:dim(mymat)[1]) {
   for (j in 1:dim(mymat)[2]) {
      mymat[i, j] = i * j
   }
}

# Show the first 10x10 chunk or the first `nr` x `nr` chunk
if (nr > 10) {
   mymat[1:10, 1:10]
} else
   mymat[1:nr, 1:nr]
```

For Loops: Popular but not Always Perfect for your Use

The for loop is by far the most popular, and its construct implies that the number of iterations is fixed and known in advance, as in cases like “generate the first 200 prime numbers” or “enlist the 10 most important customers”.

But what if you do not know or control the number of iterations, and one or several conditions may occur which are not predictable beforehand?

For example, you may want to count the number of clients living in an area identified by a specified postal code, or the number of clicks on a web page banner within the last two days, or similar unforeseen events.

In cases like these, the while loop and its cousin repeat may come to the rescue…

While Loops

The while loop, set in the middle of the figure above, is made of an initialization block as before, followed by a logical condition. This condition is typically expressed by the comparison between a control variable and a value, by using greater than, less than or equal to, but any expression that evaluates to a logical value, True or False, is legitimate.

If the result is False (F), the loop is never executed, as indicated by the loose arrow on the right of the figure. The program will then execute the first instruction it finds after the loop block.

If it is True (T), the instruction or block of instructions i1 is executed next.

Note that an additional instruction or block of instructions i2 was added: this serves as an update for the control variable, which may alter the result of the condition at the start of the loop, but this is not necessary. Or maybe you want to add an increment to a counter to keep trace of the number of iterations executed. The iterations cease once the condition evaluates to false.

The format is while(cond) expr, where cond is the condition to test and expr is an expression.

For example, the following loop asks the user with a User Defined Function or UDF to enter the correct answer to the universe and everything question. It will then continue to do so until the user gets the answer right:

```
# Your User Defined Function
```

```
readinteger <- function(){
  n <- readline(prompt="Please, enter your ANSWER: ")
}

response <- as.integer(readinteger())

while (response!=42) {
  print("Sorry, the answer to whatever the question MUST be 42");
  response <- as.integer(readinteger());
}
```

As a start, use a user defined function to get the user input before entering the loop. This loop will continue as long as the answer is not the expected 42.

In other words, you do this because otherwise, R would complain about the missing expression that was supposed to provide the required True or False -and in fact, it does not know ‘response’ before using it in the loop. You also do this because, if you answer right at first attempt, the loop will not be executed at all.

Repeat Loops

The repeat loop is located at the far right of the flow chart that you find above. This loop is similar to the while loop, but it is made so that the blocks of instructions i1 and i2 are executed at least once, no matter what the result of the condition.

Adhering to other languages, one could call this loop “repeat until” to emphasize the fact that the instructions i1 and i2 are executed until the condition remains False (F) or, equivalently, becomes True (T), thus exiting; but in any case, at least once.

As a variation of the previous example, you may write:

```
readinteger <- function(){
  n <- readline(prompt="Please, enter your ANSWER: ")
}

repeat {
  response <- as.integer(readinteger());
  if (response == 42) {
    print("Well done!");
    break
  } else print("Sorry, the answer to whatever the question MUST be 42");

}
```

After the now familiar input function, you have the repeat loop whose block is executed at least once and that will terminate whenever the if condition is verified.

Note that you had to set a condition within the loop upon which to exit with the clause break. This clause introduces us to the notion of exiting or interrupting cycles within loops.

Interruption and Exit Loops in R

So how do you exit from a loop?

In other terms, aside from the “natural” end of the loop, which occurs either because you reached the prescribed number of iterations (for) or because you met a condition (while, repeat), can you stop or interrupt the loop?

And if yes, how?

The break statement responds to the first question: you have seen this in the last example.

Break your Loops with break

When the R interpreter encounters a break, it will pass control to the instruction immediately after the end of the loop (if any). In the case of nested loops, the break will permit to exit only from the innermost loop.

Here’s an example.

This chunk of code defines an m x n matrix of zeros and then enters a nested for loop to fill the locations of the matrix, but only if the two indexes differ. The purpose is to create a lower triangular matrix, that is a matrix whose elements below the main diagonal are non-zero. The others are left untouched to their initialized zero value.

When the indexes are equal and thus the condition in the inner loop, which runs over the column index j is fulfilled, a break is executed and the innermost loop is interrupted with a direct jump to the instruction following the inner loop. This instruction is a print() instruction. Then, control gets to the outer for condition (over the rows, index i), which is evaluated again.

If the indexes differ, the assignment is performed and the counter is incremented by 1. At the end, the program prints the counter ctr, which contains the number of elements that were assigned.

```
# Make a lower triangular matrix (zeroes in upper right corner)

m = 10

n = 10


# A counter to count the assignment

ctr = 0
```

```
# Create a 10 x 10 matrix with zeroes
mymat = matrix(0, m, n)

for (i in 1:m) {
   for (j in 1:n) {
      if (i == j) {
         break
      } else {
         # you assign the values only when i<>j
         mymat[i, j] = i * j
         ctr = ctr + 1
      }
   }
   print(i * j)
}

# Print how many matrix cells were assigned
print(ctr)
```

The Use of next in Loops

next discontinues a particular iteration and jumps to the next cycle. In fact, it jumps to the evaluation of the condition holding the current loop.

In other languages, you may find the (slightly confusing) equivalent called “continue”, which means the same: wherever you are, upon the verification of the condition, jump to the evaluation of the loop.

A simpler example of keeping the loop ongoing while discarding a particular cycle upon the occurrence of a condition is:

```
m = 20

for (k in 1:m) {
   if (!k %% 2)
      next
```

```
    print(k)
}
```

This piece of code prints all uneven numbers within the interval 1:m (here m=20). In other words, all integers except the ones with non zero remainder when divided by 2 (thus the use of the modulus operand %%), as specified by the if test, will be printed.

Numbers whose remainder is zero will not be printed, as the program jumps to the evaluation of the i in 1:m condition and ignores any instruction that might follow. In this case, print(k) is ignored.

Wrapping up the Use of Loops in R

1. Try to put as little code as possible within the loop by taking out as many instructions as possible. Remember, anything inside the loop will be repeated several times and perhaps it is not needed.
2. Be careful when you use repeat: make sure that a termination is explicitly set by testing a condition or you can end up in an infinite loop.
3. It is better to use one or more function calls within the loop if a loop is getting (too) big. The function calls will make it easier for other users to follow the code. But the use of a nested for loop to perform matrix or array operations is probably a sign that you didn't implemented things in the best way for a matrix-based language like R.
4. It is not recommended to "grow" variable or dataset by using an assignment on every iteration. In some languages like Matlab, a warning error is issued: you may continue, but you are invited to consider alternatives. A typical example will be shown in the next section.

Advanced looping – Replication ,Looping over lists ,looping over arrays

The rep Function and its Applications

In this section, I will introduce the rep function and provide a basic use case in the context of creating a data frame from the simulated data we have generated. The rep function replicates the values in a given object (a vector or a function) a specified number of times. The rep function uses the following syntax and arguments:

```rep(x, ...)```

Where "x" is the value to be replicated or the vector containing values to be replicated, and "…" denotes additional arguments, such as "times=" (which specifies the number of times to replicate "x"), "each=" (which specifies the number of times to repeat each element of "x"), and "len=" (which specifies the desired length of an output vector). Below, I go over some very basic examples of each of these uses of the rep function.

A. Examples of simple replication of individual values and vectors

The simplest use of the replicate function is to replicate a text or numeric value a set number of times. To begin, let’s replicate the number 5 ten times:

```
rep(5, 10)
```

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

As you see above, entering a value n such that the sytax reads “rep(x,n)” defaults to the same meaning as the syntax “rep(x, times=n)”. Aside from individual values, we can also replicate a text label in the same manner:

```
rep("Kitty Cat", 10)
```

```
##  [1] "Kitty Cat" "Kitty Cat" "Kitty Cat" "Kitty Cat" "Kitty Cat"

##  [6] "Kitty Cat" "Kitty Cat" "Kitty Cat" "Kitty Cat" "Kitty Cat"
```

One possible use for replicating individual values or text labels is populating columns for continuous or categorical data in a data frame. I explore this use in section 5.2 B below.

Aside from replicating individual text or numeric values, the rep function is commonly used to replicate (elements within) vectors. Below, I create a simple vector of seven numeric values and then replicate that vector, as a whole, 3 times.

```
ExampleRepVector<-c(1,2,5,7,7,8,10)

rep(ExampleRepVector, 3)
```

```
##  [1]  1  2  5  7  7  8 10  1  2  5  7  7  8 10  1  2  5  7  7  8 10
```

We can also replicate each element of this vector a set number of times. Below, I first replicate each element of the vector twice. Next, I replicate each element twice, while also replicating the vector as a whole (with each element present twice) three times.

```
rep(ExampleRepVector, each=2)
```

```
##  [1]  1  1  2  2  5  5  7  7  7  7  8  8 10 10
```

```
rep(ExampleRepVector, times=3, each=2)
```

```
##  [1]  1  1  2  2  5  5  7  7  7  7  8  8 10 10  1  1  2  2  5  5  7  7  7

## [24]  7  8  8 10 10  1  1  2  2  5  5  7  7  7  7  8  8 10 10
```

Finally, let’s see what happens when we specify an output vector of a different length than the number of terms in the original vector (7):

20 terms:

```
rep(ExampleRepVector, len=20)
```

```
##  [1]  1  2  5  7  7  8 10  1  2  5  7  7  8 10  1  2  5  7  7  8
```

5 terms:

```
rep(ExampleRepVector, len=5)
```

```
## [1] 1 2 5 7 7
```

When outputting to a vector longer than the original vector, the original vector is replicated until the output vector has been filled. In our case, filling the 20 term output vector requires the original vector to be replicated twice, with an incomplete 3rd replication of only the first six terms.

When outputting to a vector shorter than the original vector, the output vector is filled before all of the values of the original vector can be replicated. In our case, the original vector has 7 values and the output vector only has room for 5, so the last 2 values are not replicated

Loop over a list

Looping over a list is just as easy and convenient as looping over a vector. There are again two different approaches here:

```
primes_list <- list(2, 3, 5, 7, 11, 13)


# loop version 1
for (p in primes_list) {
  print(p)
}


# loop version 2
for (i in 1:length(primes_list)) {
  print(primes_list[[i]])
}
```

Notice that you need double square brackets - [[ ]] - to select the list elements in loop version 2.

Suppose you have a list of all sorts of information on New York City: its population size, the names of the boroughs, and whether it is the capital of the United States. We've already defined a list nyc containing this information

Looping over Array

```
myarray <- c(1:10)


# creating a multidimensional array
multiarray <- array(myarray, dim = c(3, 2))


# using a for loop
for(x in multiarray){
  print(x)
}
```

Multiple -Input apply

Looping on the Command Line

Writing for and while loops is useful when programming but not particularly easy when working interactively on the command line. Multi-line expressions with curly braces are just not that easy to sort through when working on the command line. R has some functions which implement looping in a compact form to make your life easier.

- lapply(): Loop over a list and evaluate a function on each element
- sapply(): Same as lapply but try to simplify the result
- apply(): Apply a function over the margins of an array
- tapply(): Apply a function over subsets of a vector
- mapply(): Multivariate version of lapply

An auxiliary function split is also useful, particularly in conjunction with lapply.

lapply()

Watch a video of this section

The lapply() function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)
3. and returns a list (the l is for “list”).

This function takes three arguments: (1) a list X; (2) a function (or the name of a function) FUN; (3) other arguments via its ... argument. If X is not a list, it will be coerced to a list using as.list().

The body of the lapply() function can be seen here.

```
> lapply
function (X, FUN, ...)
{
  FUN <- match.fun(FUN)
  if (!is.vector(X) || is.object(X))
     X <- as.list(X)
  .Internal(lapply(X, FUN))
}
<bytecode: 0x7f79498e5528>
<environment: namespace:base>
```

Note that the actual looping is done internally in C code for efficiency reasons.

It’s important to remember that lapply() always returns a list, regardless of the class of the input.

Here’s an example of applying the mean() function to all elements of a list. If the original list has names, the the names will be preserved in the output.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3


$b
[1] 0.1322028
```

Notice that here we are passing the mean() function as an argument to the lapply() function. Functions in R can be used this way and can be passed back and forth as arguments just like any other object. When you pass a function to another function, you do not need to include the open and closed parentheses () like you do when you are *calling* a function.

Here is another example of using lapply().

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5


$b
[1] 0.248845


$c
[1] 0.9935285


$d
```

```
[1] 5.051388
```

You can use lapply() to evaluate a function multiple times each with a different argument. Below, is an example where I call the runif() function (to generate uniformly distributed random variables) four times, each time generating a different number of random numbers.

```
> x <- 1:4

> lapply(x, runif)

[[1]]

[1] 0.02778712


[[2]]

[1] 0.5273108 0.8803191


[[3]]

[1] 0.37306337 0.04795913 0.13862825


[[4]]

[1] 0.3214921 0.1548316 0.1322282 0.2213059
```

When you pass a function to lapply(), lapply() takes elements of the list and passes them as the *first argument* of the function you are applying. In the above example, the first argument of runif() is n, and so the elements of the sequence 1:4 all got passed to the n argument of runif().

Functions that you pass to lapply() may have other arguments. For example, the runif() function has a min and max argument too. In the example above I used the default values for min and max. How would you be able to specify different values for that in the context of lapply()?

Here is where the ... argument to lapply() comes into play. Any arguments that you place in the ... argument will get passed down to the function being applied to the elements of the list.

Here, the min = 0 and max = 10 arguments are passed down to runif() every time it gets called.

```
> x <- 1:4

> lapply(x, runif, min = 0, max = 10)

[[1]]

[1] 2.263808
```

```
[[2]]
[1] 1.314165 9.815635


[[3]]
[1] 3.270137 5.069395 6.814425


[[4]]
[1] 0.9916910 1.1890256 0.5043966 9.2925392
```

So now, instead of the random numbers being between 0 and 1 (the default), the are all between 0 and 10.

The lapply() function and its friends make heavy use of *anonymous* functions. Anonymous functions are like members of Project Mayhem—they have no names. These are functions are generated “on the fly” as you are using lapply(). Once the call to lapply() is finished, the function disappears and does not appear in the workspace.

Here I am creating a list that contains two matrices.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4


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

Suppose I wanted to extract the first column of each matrix in the list. I could write an anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) { elt[,1] })
$a
[1] 1 2
```

```
$b
[1] 1 2 3
```

Notice that I put the function() definition right in the call to lapply(). This is perfectly legal and acceptable. You can put an arbitrarily complicated function definition inside lapply(), but if it's going to be more complicated, it's probably a better idea to define the function separately.

For example, I could have done the following.

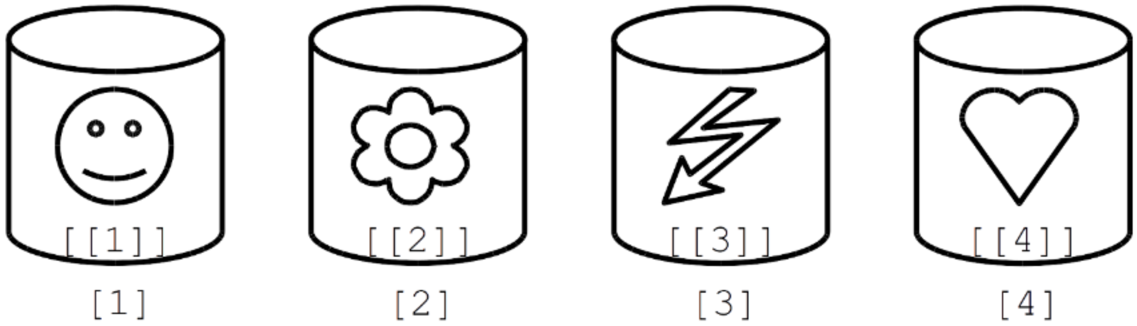
```
> f <- function(elt) {
+       elt[, 1]
+ }
> lapply(x, f)
$a
[1] 1 2


$b
[1] 1 2 3
```

Now the function is no longer anonymous; it's name is f. Whether you use an anonymous function or you define a function first depends on your context. If you think the function f is something you're going to need a lot in other parts of your code, you might want to define it separately. But if you're just going to use it for this call to lapply(), then it's probably simpler to use an anonymous function.

UNIT 3

Understanding lists

Up to now, we have been working with *atomic* data objects (vector, matrix, array). In contrast, lists, data frames, and functions are *recursive* data objects. Recursive data objects have more flexibility in combining diverse data objects into one object. A list provides the most flexibility. Think of a list object as a collection of "bins" that can contain any R object. Lists are very useful for collecting results of an analysis or a function into one data object where all its contents are readily accessible by indexing.



is a schematic representation of a list of length four. The first bin [1] contains a smiling face [[1]], the second bin [2] contains a flower [[2]], the third bin [3] contains a lightning bolt [[3]], and the fourth bin [[4]] contains a heart [[4]]. When indexing a list object, single brackets [··] indexes the *bin*, and double brackets [[··]] indexes the bin *contents*. If the bin has a name, then $*name* also indexes the contents.

Creating lists

We start right away with constructing a few simple lists for illustration. While vectors can be created using c(), lists are most often constructed using the function list().

A vector: Vector of length 2.

```
c(3, 5)

## [1] 3 5
```

A list: List containing values of the same types/classes.

```
list(3, 5)

## [[1]]

## [1] 3

##

## [[2]]

## [1] 5
```

This list with two elements ([[1]], [[2]]) contains two vectors, each of length one, indicated by [1] (first vector element).

Automatic Variables

Automatic variables in R are typically created within functions and exist only within the scope of that function. They are automatically destroyed when the function exits, thus freeing up memory. Here's a simple example:

```
my_function <- function(x) {

  y <- x + 2  # 'y' is an automatic variable

  return(y)
}


result <- my_function(5)

print(result)  # Outputs: 7
```

In this example, y is an automatic variable. It is created when my_function is called and is destroyed when the function finishes execution.

Recursive Variables

Recursive variables are used in the context of recursive functions, where a function calls itself. Recursion is a powerful technique for solving problems that can be broken down into simpler sub-problems of the same type. A classic example of recursion is the computation of the factorial of a number.

Here’s an example of a recursive function in R:

```
factorial_recursive <- function(n) {

  if (n <= 1) {

    return(1)

  } else {

    return(n * factorial_recursive(n - 1))  # Recursive call

  }

}


result <- factorial_recursive(5)

print(result)  # Outputs: 120
```

In this example, factorial_recursive is a recursive function that uses a recursive variable n in its calls. Each call to factorial_recursive creates a new instance of the variable n.

List Dimensions

Lists in R do not have dimensions in the same way that arrays or matrices do. Instead, they are collections of elements indexed by their position or by names if the elements are named. Here’s an example of creating and manipulating lists:

```
# Creating a list

my_list <- list(

  numbers = c(1, 2, 3),

  names = c("Alice", "Bob", "Charlie"),
  matrix = matrix(1:9, nrow=3)
)

# Accessing list elements
print(my_list$numbers)  # Access by name
print(my_list[[1]])     # Access by position

# Accessing elements within elements
print(my_list$matrix[1, 2])  # Access an element within the matrix in the list
```

Arithmetic with Lists

Unlike vectors or matrices, lists do not directly support arithmetic operations. To perform arithmetic on the elements within a list, you typically need to extract those elements, perform the arithmetic, and then reassign the results if needed.

Here are some examples of performing arithmetic operations on elements within a list:

```
# Example list
list1 <- list(
  a = c(1, 2, 3),
  b = c(4, 5, 6)
)

# Performing arithmetic on elements
sum_a_b <- list1$a + list1$b
print(sum_a_b)  # Outputs: 5 7 9

# Applying a function to each element in a list using lapply
squared_list <- lapply(list1, function(x) x^2)
print(squared_list)
# Outputs:
# $a
# [1] 1 4 9
```

```
#
# $b
# [1] 16 25 36

# Using mapply to apply a function to multiple list elements
sum_lists <- mapply("+", list1$a, list1$b)
print(sum_lists)  # Outputs: 5 7 9
```

Numeric Indexing

Using numeric indices, you can access elements by their position in the list.

```
# Creating a list
my_list <- list(a = 1:3, b = letters[1:3], c = matrix(1:9, nrow = 3))

# Accessing elements
print(my_list[[1]])  # Access the first element (outputs: 1 2 3)
print(my_list[[2]])  # Access the second element (outputs: "a" "b" "c")

# Accessing elements within elements
print(my_list[[3]][1, 2])  # Access the element in the first row, second column of the matrix (outputs: 4)
```

Character Indexing

Using character indices, you can access elements by their names.

```
# Accessing elements
print(my_list[["a"]])  # Access element named "a" (outputs: 1 2 3)
print(my_list[["b"]])  # Access element named "b" (outputs: "a" "b" "c")
```

$ Operator

The $ operator is a convenient shorthand for accessing named elements.

```
# Accessing elements
print(my_list$a)  # Access element named "a" (outputs: 1 2 3)
print(my_list$b)  # Access element named "b" (outputs: "a" "b" "c")
```

Logical Indexing

Logical vectors can be used to index lists, though it’s more common with vectors and matrices.

```
# Logical indexing example
logical_index <- c(TRUE, FALSE, TRUE)

# Accessing elements based on logical indexing
print(my_list[logical_index])  # Outputs elements a and c
```

Modifying List Elements

You can also modify elements of a list using indexing.

```
# Modifying an element
my_list[[1]] <- 10:12
print(my_list$a)  # Outputs: 10 11 12

# Adding a new element
my_list[["d"]] <- "new element"
print(my_list$d)  # Outputs: "new element"

# Removing an element
my_list[["a"]] <- NULL
print(my_list)  # Element "a" is removed
```

Nested Lists

For nested lists, you can use multiple indexing steps.

```
# Creating a nested list
nested_list <- list(
  outer = list(inner = list(a = 1, b = 2))
)

# Accessing nested elements
print(nested_list[[1]][[1]][["a"]])  # Access the element "a" within the nested list (outputs: 1)
```

Convert List to Vector

To convert List to Vector in R, call unlist() function and pass the list as argument to the function.

In this tutorial, we will learn how to convert a list to vector in R, using unlist() function, with the help of example program.

Example

In the following program, we create a list with three elements, and convert it into vector using unlist().

Example.R

```
x <- list(18, 25, 33)
y <- unlist(x)
print(y)
```

Output

```
[1] 18 25 33
```

List with Different Datatypes

If list contains elements of different datatype, then lesser datatypes would be promoted to the highest datatype.

In the following program, we take a list with logical, double and character elements. When we convert this list to vector, logical and double values would be promoted to character datatype.

Example.R

```
x <- list(TRUE, 25, "Apple")
print(x)

y <- unlist(x)
print(y)
```

Output

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

[[2]]
[1] 25
```

```
[[3]]
[1] "Apple"

[1] "TRUE"  "25"    "Apple"
```

Converting Numeric Vectors to Lists

```
# Numeric vector
numeric_vector <- c(1, 2, 3, 4, 5)

# Convert to list
numeric_list <- as.list(numeric_vector)

# Print the list
print(numeric_list)
# Outputs:
# [[1]]
# [1] 1
#
# [[2]]
# [1] 2
#
# [[3]]
# [1] 3
#
# [[4]]
# [1] 4
#
# [[5]]
# [1] 5
```

Converting Character Vectors to Lists

```
# Character vector
char_vector <- c("a", "b", "c", "d")

# Convert to list
char_list <- as.list(char_vector)

# Print the list
print(char_list)
# Outputs:
# [[1]]
# [1] "a"
#
# [[2]]
# [1] "b"
#
# [[3]]
# [1] "c"
#
# [[4]]
# [1] "d"
```

Converting Logical Vectors to Lists

```
# Logical vector
logical_vector <- c(TRUE, FALSE, TRUE)

# Convert to list
logical_list <- as.list(logical_vector)

# Print the list
print(logical_list)
# Outputs:
```

```
# [[1]]
# [1] TRUE
#
# [[2]]
# [1] FALSE
#
# [[3]]
# [1] TRUE
```

Converting Complex Vectors to Lists

```
# Complex vector
complex_vector <- c(1+1i, 2+2i, 3+3i)

# Convert to list
complex_list <- as.list(complex_vector)

# Print the list
print(complex_list)
# Outputs:
# [[1]]
# [1] 1+1i
#
# [[2]]
# [1] 2+2i
#
# [[3]]
# [1] 3+3i
```

Converting Factors to Lists

Factors in R are used to represent categorical data and can also be converted to lists.

```
# Factor
factor_vector <- factor(c("high", "medium", "low"))
```

```
# Convert to list
factor_list <- as.list(factor_vector)

# Print the list
print(factor_list)
# Outputs:
# [[1]]
# [1] high
# Levels: high low medium
#
# [[2]]
# [1] medium
# Levels: high low medium
#
# [[3]]
# [1] low
# Levels: high low medium
```

Combining Lists with Different Structures

You can also combine lists that contain different types of elements, such as vectors, matrices, and other lists.

```
# Creating lists with different structures
list1 <- list(a = 1:3, b = matrix(1:4, nrow = 2))
list2 <- list(c = "hello", d = list(x = 10, y = 20))

# Combining lists
combined_list <- c(list1, list2)

# Print the combined list
print(combined_list)
# Outputs:
```

```
# $a
# [1] 1 2 3
#
# $b
#      [,1] [,2]
# [1,]    1    3
# [2,]    2    4
#
# $c
# [1] "hello"
#
# $d
# $d$x
# [1] 10
#
# $d$y
# [1] 20
```

Creating and Assigning NULL

You can explicitly assign NULL to variables or list elements.

```
# Assigning NULL to a variable
x <- NULL
print(x)  # Outputs: NULL

# Assigning NULL to a list element
my_list <- list(a = 1, b = 2, c = NULL)
print(my_list)
# Outputs:
# $a
# [1] 1
#
```

```
# $b
# [1] 2
#
# $c
# NULL
```

Checking for NULL

To check if a variable or list element is NULL, you can use the is.null() function.

```
# Checking for NULL
x <- NULL
print(is.null(x))  # Outputs: TRUE

y <- 5
print(is.null(y))  # Outputs: FALSE

my_list <- list(a = 1, b = NULL)
print(is.null(my_list$b))  # Outputs: TRUE
```

Removing Elements from Lists Using NULL

Assigning NULL to a list element effectively removes that element from the list.

```
# Removing list elements by assigning NULL
my_list <- list(a = 1, b = 2, c = 3)
my_list$b <- NULL
print(my_list)
# Outputs:
# $a
# [1] 1
#
# $c
# [1] 3
```

Using NULL in Function Arguments

NULL can be used as a default value for function arguments.

```
# Function with NULL default argument
my_function <- function(x = NULL) {
  if (is.null(x)) {
    return("No value provided")
  } else {
    return(x)
  }
}


print(my_function())        # Outputs: "No value provided"
print(my_function(10))      # Outputs: 10
```

In R programming, pairlists are a type of data structure primarily used internally to represent the arguments of a function or the components of a language object. They are not as commonly used directly by users as other data structures like vectors, lists, or data frames. However, understanding pairlists can be useful, especially when dealing with more advanced R programming concepts such as the manipulation of language objects or environments.

What is a Pairlist?

A pairlist is a recursive list where each element is a pair (a key-value pair). The key is a name (often NULL), and the value can be any R object, including another pairlist. Pairlists are primarily used internally in R to represent function arguments and other similar structures.

Creating Pairlists

You can create a pairlist using the pairlist() function.

```
# Creating a pairlist
pl <- pairlist(a = 1, b = 2, c = 3)


# Print the pairlist
print(pl)
# Outputs:
# $a
# [1] 1
#
# $b
# [1] 2
#
# $c
# [1] 3
```

Accessing Elements in Pairlists

You can access elements in a pairlist similarly to how you would access elements in a list, using the $ operator or double square brackets [[ ]].

```
# Accessing elements in a pairlist
print(pl$a)       # Outputs: 1
print(pl[["b"]])  # Outputs: 2
```

Manipulating Pairlists

You can add, modify, or remove elements in a pairlist.

```
# Adding an element to a pairlist
pl$d <- 4
print(pl)
# Outputs:
# $a
# [1] 1
#
# $b
# [1] 2
#
# $c
# [1] 3
#
# $d
# [1] 4

# Modifying an element in a pairlist
pl$a <- 10
print(pl$a)  # Outputs: 10
```

```
# Removing an element from a pairlist
pl$b <- NULL
print(pl)
# Outputs:
# $a
# [1] 10
#
# $c
# [1] 3
#
# $d
# [1] 4
```

Data Frames _ Creating Data Frames , Indexing Data Frames, Basic Data Frame Manipulation

Data Frames is a tables where data is organized into rows and columns. This is the essence of data frame–it is a table of similar objects, and for each object we know multiple distinct types of information. It turns out to be a very powerful way to represent data, and data frames are incorporated not just into R but also in many other analysis frameworks.

Creating Data Frames

You can create data frames using the data.frame() function, converting other data structures, or by reading data from external files.

Using data.frame()

```
# Creating a data frame from vectors
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Height = c(5.5, 6.0, 5.8)
)

# Print the data frame
```

```
print(df)
# Outputs:
#      Name Age Height
# 1   Alice  25    5.5
# 2     Bob  30    6.0
# 3 Charlie  35    5.8
```

Indexing Data Frames

Indexing in data frames can be done using numeric, character, or logical indices.

Numeric Indexing

```
# Accessing rows and columns by numeric index
print(df[1, 2])      # Outputs: 25 (element in the 1st row and 2nd column)
print(df[1, ])       # Outputs the entire 1st row
print(df[, 2])       # Outputs the entire 2nd column
print(df[1:2, 1:2])  # Outputs the first 2 rows and first 2 columns
```

Character Indexing

```
# Accessing columns by column names
print(df["Name"])    # Outputs the "Name" column
print(df[, "Age"])   # Outputs the "Age" column
print(df[c("Name", "Age")])  # Outputs the "Name" and "Age" columns
```

Logical Indexing

```
# Logical indexing to filter rows
print(df[df$Age > 25, ])  # Outputs rows where Age is greater than 25
```

Basic Data Frame Manipulation

Adding Columns

```
# Adding a new column
df$Weight <- c(130, 150, 160)

# Print the updated data frame
print(df)
# Outputs:
```

```
#      Name Age Height Weight
# 1   Alice  25    5.5    130
# 2     Bob  30    6.0    150
# 3 Charlie  35    5.8    160
```

Adding Rows

```
# Adding a new row using rbind
new_row <- data.frame(Name = "David", Age = 40, Height = 6.2, Weight = 180)
df <- rbind(df, new_row)

# Print the updated data frame
print(df)
# Outputs:
#      Name Age Height Weight
# 1   Alice  25    5.5    130
# 2     Bob  30    6.0    150
# 3 Charlie  35    5.8    160
# 4   David  40    6.2    180
```

Removing Columns

```
# Removing a column by setting it to NULL
df$Weight <- NULL

# Print the updated data frame
print(df)
# Outputs:
#      Name Age Height
# 1   Alice  25    5.5
# 2     Bob  30    6.0
# 3 Charlie  35    5.8
# 4   David  40    6.2
```

Removing Rows

```
# Removing rows using negative indexing
df <- df[-c(1, 2), ]


# Print the updated data frame
print(df)
# Outputs:
#      Name Age Height
# 3 Charlie  35    5.8
# 4   David  40    6.2
```

Modifying Elements

```
# Modifying specific elements
df[1, 2] <- 36  # Change the Age of the first row
print(df)
# Outputs:
#      Name Age Height
# 3 Charlie  36    5.8
# 4   David  40    6.2
```

Functions

Writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere "user" to a developer who creates new functionality for R. Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions. Functions are also often written when code must be shared with others or the public.

The writing of a function allows a developer to create an interface to the code, that is explicitly specified with a set of parameters. This interface provides an abstraction of the code to potential users. This abstraction simplifies the users' lives because it relieves them from having to know every detail of how the code operates. In addition, the creation of an interface allows the developer to communicate to the user the aspects of the code that are important or are most relevant.

Functions in R

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions. This is very handy for the various apply functions, like lapply() and sapply().

- Functions can be nested, so that you can define a function inside of another function

If you’re familiar with common language like C, these features might appear a bit strange. However, they are really important in R and can be useful for data analysis.

Creating Functions

You create a function in R using the function keyword. The basic syntax for defining a function is:

```
function_name <- function(arguments) {
  # Function body
  # Return a value
}
```

Example: Simple Function

```
# Define a function that adds two numbers
add <- function(x, y) {
  return(x + y)
}

# Call the function
result <- add(3, 5)
print(result)  # Outputs: 8
```

Example: Function with Default Arguments

```
# Define a function with a default argument
greet <- function(name = "World") {
  return(paste("Hello", name))
}

# Call the function with and without argument
print(greet())        # Outputs: "Hello World"
print(greet("Alice")) # Outputs: "Hello Alice"
```

Calling Functions

Calling a function in R is straightforward. You use the function name followed by parentheses, including any required arguments inside the parentheses.

```
# Call a function with required arguments
result <- add(10, 20)
print(result)  # Outputs: 30

# Call a function with named arguments
result <- add(y = 5, x = 15)
print(result)  # Outputs: 20
```

Passing Functions as Arguments

In R, you can pass functions as arguments to other functions. This is useful for creating higher-order functions.

Example: Function as an Argument

```
# Define a function that applies another function to its arguments
apply_function <- function(func, a, b) {
  return(func(a, b))
}

# Call apply_function with add as an argument
result <- apply_function(add, 4, 6)
print(result)  # Outputs: 10
```

Advanced Function Features

Returning Multiple Values

You can return multiple values from a function using a list.

```
# Define a function that returns multiple values
multi_return <- function(a, b) {
  sum <- a + b
  product <- a * b
  return(list(sum = sum, product = product))
}

# Call the function and capture the result
result <- multi_return(3, 5)
```

```
print(result)
# Outputs:
# $sum
# [1] 8
#
# $product
# [1] 15

# Accessing the returned values
print(result$sum)      # Outputs: 8
print(result$product)  # Outputs: 15
```

Types of Scope in R

1. **Global Scope**: Variables defined in the global environment can be accessed from anywhere in the R script or session. These variables are not confined to any function or block.
2. **Local Scope**: Variables defined within a function are local to that function. They cannot be accessed outside the function. Once the function execution is complete, local variables are destroyed.
3. **Lexical Scope**: R uses lexical scoping, which means that the value of a variable is determined by the environment in which the function was defined, not where it was called. This allows for more flexible and powerful programming.

Global Scope

Variables defined in the global environment are accessible anywhere in the script or console.

```
# Global variable
x <- 10

# Function accessing global variable
my_function <- function() {
  return(x + 5)
}

# Call the function
```

```
print(my_function())  # Outputs: 15
```

Local Scope

Variables defined within a function are not accessible outside of that function.

```
# Function with local variable
my_function <- function() {
  y <- 5  # Local variable
  return(y + 10)
}

# Call the function
print(my_function())  # Outputs: 15

# Trying to access the local variable outside the function
# print(y)  # Error: object 'y' not found
```

Lexical Scope

In R, functions can access variables from their environment (the environment in which they were defined), allowing for nested functions and closures.

```
# Outer function
outer_function <- function(a) {
  # Inner function
  inner_function <- function(b) {
    return(a + b)  # Accesses 'a' from the outer function
  }

  return(inner_function)
}

# Create a function that adds 5
add_five <- outer_function(5)

# Call the inner function
```

```
print(add_five(10))  # Outputs: 15
```

Variable Masking

If a variable is defined in both the global and local scopes, the local variable will mask the global variable within that function.

```
# Global variable
x <- 10

# Function with a local variable of the same name
my_function <- function() {
  x <- 5  # Local variable masks the global variable
  return(x)
}

# Call the function
print(my_function())  # Outputs: 5

# Access the global variable
print(x)  # Outputs: 10
```

Global Scope: Variables in the global environment are accessible anywhere in the code.

Local Scope: Variables defined within a function are not accessible outside that function.

Lexical Scope: R allows functions to access variables from their defining environment, enabling nested functions.

Variable Masking: Local variables can mask global variables of the same name within a function.

Modifying Global Variables: Use the <<- operator to modify global variables from within a function, but use it cautiously.

In R, strings are a fundamental data type used to represent text. This guide covers constructing and printing strings, formatting numbers, working with special characters, changing case, and extracting substrings.

Constructing Strings

You can create strings in R using single quotes (') or double quotes (").

```
# Creating strings
```

```
string1 <- "Hello, World!"
string2 <- 'Welcome to R programming.'

# Print strings
print(string1)  # Outputs: [1] "Hello, World!"
print(string2)  # Outputs: [1] "Welcome to R programming."
```

Printing Strings

You can use the print() function or cat() to print strings. The cat() function is useful for concatenating and displaying strings.

```
# Using print
print(string1)  # Outputs: [1] "Hello, World!"

# Using cat
cat(string2, "\n")  # Outputs: Welcome to R programming.
```

Formatting Numbers

R provides several functions to format numbers as strings.

Using sprintf()

The sprintf() function formats numbers with specified precision and other formatting options.

```
# Formatting numbers with sprintf
formatted_number <- sprintf("The value of pi is approximately %.2f", pi)
print(formatted_number)  # Outputs: "The value of pi is approximately 3.14"
```

Using format()

The format() function can format numbers as well, allowing for control over decimal places and scientific notation.

```
# Formatting numbers with format
formatted_value <- format(1234.5678, nsmall = 2, big.mark = ",")
print(formatted_value)  # Outputs: "1,234.57"
```

Special Characters

Special characters can be included in strings using escape sequences.

- \": Double quote
- \': Single quote

- \\: Backslash
- \n: New line
- \t: Tab

```
# Using special characters
string_with_specials <- "She said, \"Hello!\"\nThis is a new line."
cat(string_with_specials)
# Outputs:
# She said, "Hello!"
# This is a new line.
```

Changing Case

You can change the case of strings using toupper() for uppercase and tolower() for lowercase.

```
# Changing case
uppercase_string <- toupper("Hello, World!")
lowercase_string <- tolower("Welcome to R Programming.")

print(uppercase_string)  # Outputs: "HELLO, WORLD!"
print(lowercase_string)   # Outputs: "welcome to r programming."
```

Extracting Substrings

You can extract substrings using the substr() or substring() functions.

Using substr()

```
# Extracting substrings with substr
text <- "Hello, World!"
sub_string <- substr(text, start = 1, stop = 5)
print(sub_string)  # Outputs: "Hello"
```

Using substring()

The substring() function is useful for extracting substrings based on character positions.

```
# Extracting substrings with substring
sub_string <- substring(text, 8, 12)  # Extracts from position 8 to 12
print(sub_string)  # Outputs: "World"
```

UNIT-IV

General Statistics

Introduction

Any significant application of R includes statistics or models or graphics. This chapter addresses the statistics. Some recipes simply describe how to calculate a statistic, such as relative frequency. Most recipes involve statistical tests or confidence intervals. The statistical tests let you choose between two competing hypotheses; that paradigm is described next. Confidence intervals reflect the likely range of a population parameter and are calculated based on your data sample.

What is Statistics?

Statistics is an area of applied mathematics concerned with data collection, analysis, interpretation,
and presentation.
This area of mathematics deals with understanding how data can be used to solve complex problems.
Here are a couple of example problems that can be solved by using statistics:

- Your company has created a new drug that may cure cancer. How would you conduct a test to confirm the drug's effectiveness?
- You and a friend are at a baseball game, and out of the blue, he offers you a bet that neither team will hit a home run in that game. Should you take the bet?
- The latest sales data have just come in, and your boss wants you to prepare a report for management on places where the company could improve its business. What should you look for? What should you not look for?

Statistical Terms

There are various statistical terms that one should be aware of while dealing with statistics.
Popolation: A collection of all probable observations of a specific characteristic of interest. It is
a group from which data are collected.
Example: All learners taking data science course.
Sample: A subset of population
Example: A group of 20 learners selected for a quiz
Variable: An item of interest that can acquire various numerical values

Categories Of Data

Data can be categorized into two sub-categories: 1. Qualitative Data 2. Quantitative Data
Qualitative Data: Qualitative data deals with characteristics and descriptors that can't be easily measured, but can be observed subjectively. Qualitative data is further divided into two types of data:
Nominal Data: Data with no inherent order or ranking such as gender or race.
Ordinal Data: Data with an ordered series of information is called ordinal data.
Quantitative Data: Quantitative data deals with numbers and things you can measure objectively. This is further divided into two:
Discrete Data: Also known as categorical data, it can hold a finite number of possible values.
Example: Number of students in a class.
Continuous Data: Data that can hold an infinite number of possible values. Example: Weight of a person.

Types of Statistics
There are two well-defined types of statistics:
1. Descriptive Statistics
2. Inferential Statistics

Descriptive Statistics
Descriptive statistics is a method used to describe and understand the features of a specific data set by giving short summaries about the sample and measures of the data.
Descriptive Statistics is mainly focused upon the main characteristics of data. It provides a graphical
summary of the data.
Suppose you want to gift all your classmate's t-shirts. To study the average shirt size of students in a classroom, in descriptive statistics you would record the shirt size of all students in the class
and then you would find out the maximum, minimum and average shirt size of the class.
Descriptive Statistics is broken down into two categories:
1. Measures of Central Tendency
2. Measures of Variability (spread)

Measures of Central Tendency
Measures of center are statistics that give us a sense of the "middle" of a numeric variable. Common
measures of center include: - mean - median - mode
0.2.3 Mean
Arithmetic average of a range of values or quantities, computed by dividing the total of all values
by the number of values.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Median Denotes value or quantity lying at the midpoint of a frequency distribution of observed values or quantities, such that there is an equal probability of falling above or below it. Simply put, it is the *middle* value in the list of numbers.

If count is odd, the median is the value at $\frac{(n+1)}{2}$,

else it is the average of $\frac{n}{2}$ and $\frac{(n+1)}{2}$

Measures of Spread
Measures of spread (dispersion) are statistics that describe how data varies. While measures of center give us an idea of the typical value, measures of spread give us a sense of how much the data
4
tends to diverge from the typical value. The measures of spread are: - Range - Standard deviation
- Variance - Interquartile range
Range Range is the difference between the maximum and minimum observations.

$Range = max(x) - min(x)$

Variance It’s the average distance of the data values from the *mean*. Variance can be calculated by using the below formula:

$$Variance = S^2 = \sum \frac{(X - \overline{X})^2}{n}$$

Here,

X: Individual data points n: Total number of data points x: Mean of data points

Interquartile Range(IQR) It is the measure of statistcal dispersion, being equal to the difference between upper and lower quartile.

$$IQR = Q3 - Q1$$

Quartiles tell us about the spread of a data set by breaking the data set into quarters, just like the median breaks it in half.

Standard Deviation It is the square root of variance. This will have the same units as the data and mean. It can be calculated by using the below formula:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2}$$

Skewness and Kurtosis Beyond measures of center and spread, descriptive statistics include measures that give you a sense of the shape of a distribution.
Skewness is a measure of the asymmetry of a data distribution. Skewness is asymmetry in a statistical distribution, in which the curve appears distorted or skewed either to the left or to the right. Skewness can be quantified to define the extent to which a distribution differs from a normal distribution.
Kurtosis
Skewness measures the skew while kurtosis measures the “peakedness” of a distribution.
We won’t go into the exact calculations behind skewness and kurtosis, but they are essentially just statistics that take the idea of variance a step further: while variance involves squaring deviations from the mean, skewness involves cubing deviations from the mean and kurtosis involves raising deviations from the mean to the 4th power.

Inferential Statistics

Inferential statistics generalize the larger dataset and applies probability theory to draw a conclusion. It allows you to infer population parameters based on sample statistics and to model relationships within data.

Correlation

A correlation is a statistical test of association between variables that is measured on a -1 to 1 scale. The closer the correlation value is to -1 or 1 the stronger the association, the closer to 0, the weaker the association. It measures how change in one variable is associated with change in another variable.
There are a few common types of tests to measure the level of correlation: **Pearson, Spearman,**

and Kendall.
Each have their own assumptions about the data that needs to be meet in order for the test to be able to accurately measure the level of correlation. Each type of correlation test is testing the following hypothesis.
Extent to which two or more variables fluctuate together. A **positive correlation** indicates the extent to which those variables increase or decrease in parallel; a **negative correlation** indicates the extent to which one variable increases as the other decreases.

$$r = \frac{1}{n-1} \sum \left(\frac{x-\bar{x}}{s_x}\right)\left(\frac{y-\bar{y}}{s_y}\right)$$

Pearson correlation assumptions
Pearson correlation test is a parametric test that makes assumption about the data. In order for the results of a Pearson correlation test to be valid, the data must meet these assumptions:
- The sample is independently and randomly drawn
- A linear relationship between the two variables is present

– When plotted, the lines form a line and is not curved
- There is homogeneity of variance

The variables being used in the correlation test should be continuous and measured either on a ratio or interval sale, each variable must have equal number of non-missing observations, and there should be no outliers present.

Spearman Rank correlation assumptions
The Spearman rank correlation is a non-parametric test that does not make any assumptions about the distribution of the data. The assumption for the Spearman rank correlation test is:
- There is a monotonic relationship between the variables being tested
- A monotonic relationship exists when one variable increases so does the other

For the Spearman rank correlation, the data can be used on ranked data, if the data is not normally distributed, and even if the there is not homogeneity of variance.
Kendall's Tau correlation assumptions
The Kendall's Tau correlation is a non-parametric test that does not make any assumptions about the distribution of the data. The only assumption is:
- There should be a monotonic relationship between the variables being tested

The data should be measured on either an ordinal, ratio, or interval scale.

Null Hypotheses, Alternative Hypotheses, and p-Values

Many of the statistical tests in this chapter use a time-tested paradigm of statistical inference. In the paradigm, we have one or two data samples. We also have two competing hypotheses, either of which could reasonably be true.

One hypothesis, called the *null hypothesis*, is that *nothing happened*: the mean was unchanged; the treatment had no effect; you got the expected answer; the model did not improve; and so forth.

The other hypothesis, called the *alternative hypothesis*, is that *something happened*: the mean rose; the treatment improved the patients' health; you got an unexpected answer; the model fit better; and so forth.

We want to determine which hypothesis is more likely in light of the data:

1. To begin, we assume that the null hypothesis is true.
2. We calculate a test statistic. It could be something simple, such as the mean of the sample, or it could be quite complex. The critical requirement is that we must know the statistic's distribution. We might know the distribution of the sample mean, for example, by invoking the Central Limit Theorem.
3. From the statistic and its distribution we can calculate a *p*-value, the probability of a test statistic value as extreme or more extreme than the one we observed, while assuming that the null hypothesis is true.
4. If the *p*-value is too small, we have strong evidence against the null hypothesis. This is called *rejecting* the null hypothesis.
5. If the *p*-value is not small, then we have no such evidence. This is called *failing to reject* the null hypothesis.

There is one necessary decision here: When is a *p*-value "too small"?

But the real answer is, "it depends." Your chosen significance level depends on your problem domain. The conventional limit of $p < 0.05$ works for many problems. In our work, the data are especially noisy and so we are often satisfied with $p < 0.10$. For someone working in high-risk areas, $p < 0.01$ or $p < 0.001$ might be necessary.

In the recipes, we mention which tests include a *p*-value so that you can compare the *p*-value against your chosen significance level of α. We worded the recipes to help you interpret the comparison. Here is the wording from Recipe 9.4, Testing Categorical Variables for Independence, a test for the independence of two factors:

This is a compact way of saying:

- The null hypothesis is that the variables are independent.
- The alternative hypothesis is that the variables are not independent.
- For $\alpha = 0.05$, if $p < 0.05$ then we reject the null hypothesis, giving strong evidence that the variables are not independent; if $p > 0.05$, we fail to reject the null hypothesis.
- You are free to choose your own α, of course, in which case your decision to reject or fail to reject might be different.

Remember, the recipe states the *informal interpretation* of the test results, not the rigorous mathematical interpretation. We use colloquial language in the hope that it will guide you toward a practical understanding and application of the test.

Confidence Intervals

Hypothesis testing is a well-understood mathematical procedure, but it can be frustrating. First, the semantics is tricky. The test does not reach a definite, useful conclusion. You might get strong evidence against the null hypothesis, but that's all you'll get. Second, it does not give you a number, only evidence.

If you want numbers then use confidence intervals, which bound the estimate of a population parameter at a given level of confidence. Recipes in this chapter can calculate confidence intervals for means, medians, and proportions of a population.

For example, calculates a 95% confidence interval for the population mean based on sample data. The interval is $97.16 < \mu < 103.98$, which means there is a 95% probability that the population's mean, μ, is between 97.16 and 103.98.

Summarizing Your Data

The summary function gives some useful statistics for vectors, matrices, factors, and data frames:

The Solution exhibits the summary of a vector. The 1st Qu. and 3rd Qu. are the first and third quartile, respectively. Having both the median and mean is useful because you can quickly detect skew. The preceding Solution, for example, shows a mean that is larger than the median; this indicates a possible skew to the right, as one would expect from a lognormal distribution.

Scenario 1: vec is a Numeric Vector

If vec is a numeric vector, summary(vec) will provide the following summary statistics:

- Min.: The minimum value
- 1st Qu.: The first quartile (25th percentile)
- Median: The median (50th percentile)
- Mean: The mean (average) value
- 3rd Qu.: The third quartile (75th percentile)
- Max.: The maximum value

```
vec <- c(1, 2, 3, 4, 5)

summary(vec)
```

output

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
    1     2     3            3      4        5
```

Scenario 2: vec is a Factor

If vec is a factor, summary(vec) will provide a frequency table of the levels.

Example:

```
vec <- factor(c("apple", "banana", "apple", "orange", "banana", "apple"))

summary(vec)
```

output

```
  apple   banana   orange

     3       2       1
```

Scenario 3: vec is a Character Vector

If vec is a character vector, summary(vec) will convert it to a factor and then provide a frequency table of the levels.

Example:

```
vec <- c("apple", "banana", "apple", "orange", "banana", "apple")

summary(vec)
```

output

```
  Length     Class           Mode

      6     character      character
```

Scenario 4: vec is a Logical Vector

If vec is a logical vector, summary(vec) will count the number of TRUE, FALSE, and NA values.

Example:

```
vec <- c(TRUE, FALSE, TRUE, NA, FALSE, TRUE)

summary(vec)
```

output

```
  Mode    TRUE    FALSE      NA

logical      3          2          1
```

Scenario 5: vec is a Complex Vector

If vec is a complex vector, summary(vec) will provide summary statistics for the real and imaginary parts separately.

Example:

```
vec <- c(1+2i, 3-4i, 5+6i)

summary(vec)
```

output

```
          Min.   1st Qu.   Median     Mean   3rd Qu.     Max.

Real       1.0      3.0      3.0      3.0      5.0      5.0

Imaginary -4.0      2.0      2.0      1.3      4.0      6.0
```

Scenario 6: vec is a Date or Time Object

If vec is a date or time object, summary(vec) will provide information specific to dates, such as the earliest and latest dates.

Example:

```
vec <- as.Date(c("2022-01-01", "2022-06-15", "2022-12-31"))

summary(vec)
```

output

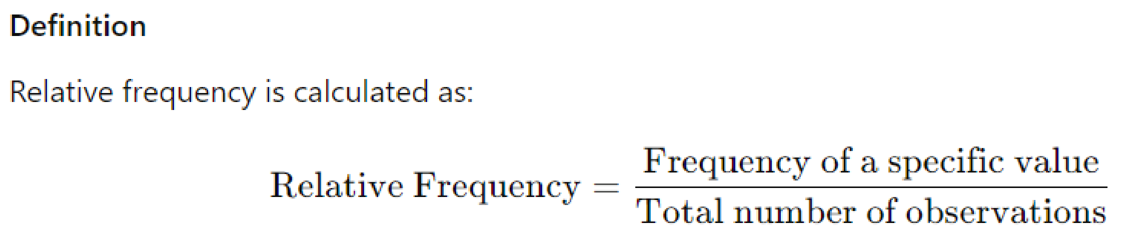
```
        Min.       1st Qu.       Median         Mean       3rd Qu.        Max.

"2022-01-01" "2022-04-08" "2022-06-15" "2022-06-15" "2022-09-23" "2022-12-31"
```

Calculating Relative Frequencies

Relative frequency is a measure used in statistics to describe the proportion of times a particular value occurs compared to the total number of observations. It provides a way to understand how common or rare a particular event is within a dataset. Relative frequencies are often used in exploratory data analysis and are particularly useful for comparing different categories or classes in categorical data.

Definition

Relative frequency is calculated as:



Definition

Relative frequency is calculated as:

$$\text{Relative Frequency} = \frac{\text{Frequency of a specific value}}{\text{Total number of observations}}$$

Where:

- **Frequency of a specific value is the count of how many times a particular value occurs in the dataset.**
- **Total number of observations is the total count of all values in the dataset.**

The result of this calculation is a proportion that can range from 0 to 1, or it can be expressed as a percentage by multiplying the proportion by 100.

Steps to Calculate Relative Frequencies

1. **Collect Data: Gather the dataset containing the values you wish to analyze.**
2. **Count Frequencies: Count how many times each distinct value occurs in the dataset. This can be done using a frequency table.**
3. **Calculate Total Observations: Determine the total number of observations in the dataset.**
4. **Compute Relative Frequencies: For each distinct value, divide the frequency of that value by the total number of observations.**
5. **(Optional) Convert to Percentage: Multiply the relative frequency by 100 to express it as a percentage.**

```
# Create a sample data frame with a factor variable
data_frame <- data.frame(
  Category = factor(c("A", "B", "A", "C", "B", "A", "C", "C", "B", "A"))
)
print(data_frame)


# Count the absolute frequency of each category
absolute_frequency_df <- table(data_frame$Category)
print(absolute_frequency_df)


# Calculate the relative frequency
relative_frequency_df <- absolute_frequency_df / nrow(data_frame)
print(relative_frequency_df)
```

output

```
   Category
1         A
2         B
3         A
4         C
```

```
5         B
6         A
7         C
8         C
9         B
10        A
Category
A B C
4 3 3
Category
  A   B   C
0.4 0.3 0.3
```

Example: Contingency Table with Two Factors

A **contingency table**, also known as a cross-tabulation or cross-tab, is a type of data table that displays the frequency distribution of two categorical variables. It allows researchers to observe the relationship between the two variables and analyze how the presence or absence of one variable affects the other. Contingency tables are a fundamental tool in statistics, particularly in categorical data analysis.

Definition

A contingency table consists of rows and columns that represent the different categories of the two factors being analyzed. The cells of the table contain counts (frequencies) that show how many observations fall into each combination of categories.

Structure of a Contingency Table

1. **Rows**: Represent the categories of one variable (Factor A).
2. **Columns**: Represent the categories of the other variable (Factor B).
3. **Cells**: Contain the frequency counts of observations that correspond to the specific combinations of the categories of Factor A and Factor B.
4. **Marginal Totals**: The sums of rows and columns, indicating the total frequencies for each category of both factors.

Step 1: Create a Data Frame

Let's create a small data frame with two categorical variables: Gender and Preference.

```
# Create a sample data frame
```

```
data <- data.frame(
  Gender = factor(c("Male", "Female", "Female", "Male", "Male", "Female")),
  Preference = factor(c("A", "A", "B", "B", "A", "B"))
)
print(data)
```

output

```
   Gender Preference
1    Male          A
2  Female          A
3  Female          B
4    Male          B
5    Male          A
6  Female          B
```

Step 2: Create a Contingency Table

Now, use the table() function to create a contingency table from these two factors.

```
# Create a contingency table for Gender and Preference
contingency_table <- table(data$Gender, data$Preference)
print(contingency_table)
```

```
          A  B
  Female  1  2
  Male    2  1
```

Chi-Squared Test

The **Chi-Square test** is a statistical method used to determine whether there is a significant association between two categorical variables. It is widely used in hypothesis testing to assess how closely observed frequencies match expected frequencies under the null hypothesis. This test is particularly useful in various fields, including social sciences, health research, and market analysis.

Types of Chi-Square Tests

1. **Chi-Square Test of Independence**: This test is used to determine whether two categorical variables are independent of each other. It compares the observed frequency distribution of the variables to what would be expected if they were independent.

2. **Chi-Square Goodness of Fit Test**: This test assesses whether the distribution of a single categorical variable matches a specified theoretical distribution. It evaluates how well the observed frequencies of the categories fit the expected frequencies.

Chi-Square Test of Independence

Hypotheses

- **Null Hypothesis (H_0)**: Assumes that there is no association between the two categorical variables (they are independent).
- **Alternative Hypothesis (H_a)**: Assumes that there is an association between the two categorical variables (they are not independent).

Test Statistic

The Chi-Square test statistic is calculated using the formula:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

Where:

- O_i = Observed frequency for category i
- E_i = Expected frequency for category i
- The summation is over all categories.

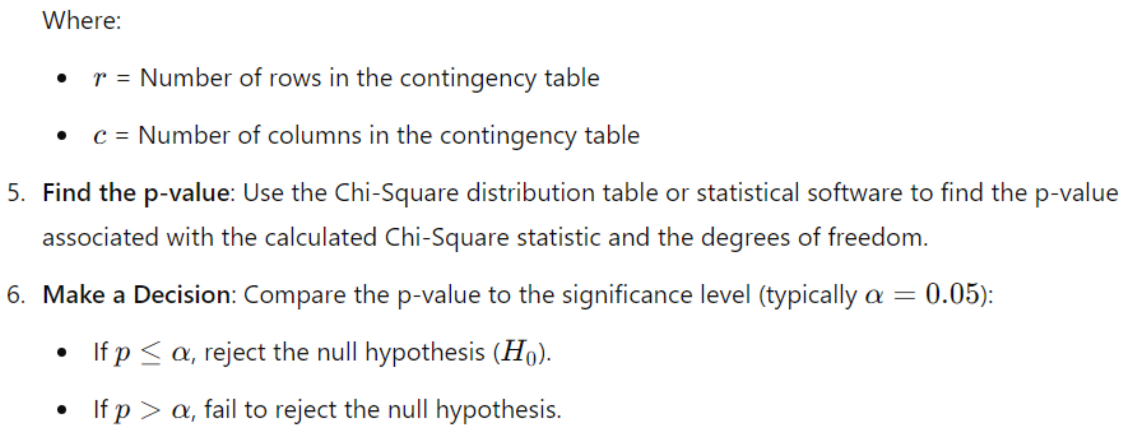
The expected frequency is calculated as:

$$E_i = \frac{(Row\,Total \times Column\,Total)}{Grand\,Total}$$

Steps to Conduct the Chi-Square Test of Independence

1. **Create a Contingency Table**: Organize the data into a contingency table to display the observed frequencies for each combination of the categorical variables.
2. **Calculate Expected Frequencies**: Use the formula for expected frequencies based on the row and column totals.
3. **Compute the Chi-Square Statistic**: Apply the formula to calculate the Chi-Square test statistic.
4. **Determine Degrees of Freedom**: The degrees of freedom for the test is calculated as:

$$df = (r - 1) \times (c - 1)$$

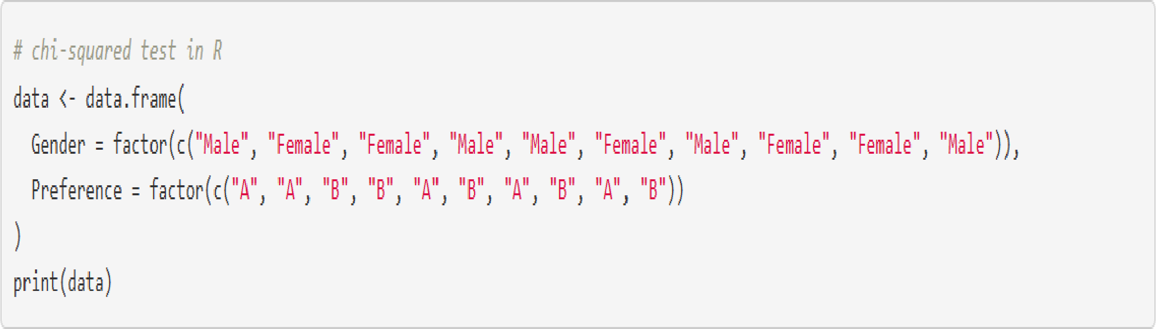


Where:

- r = Number of rows in the contingency table
- c = Number of columns in the contingency table

5. **Find the p-value**: Use the Chi-Square distribution table or statistical software to find the p-value associated with the calculated Chi-Square statistic and the degrees of freedom.
6. **Make a Decision**: Compare the p-value to the significance level (typically $\alpha = 0.05$):
   - If $p \leq \alpha$, reject the null hypothesis (H_0).
   - If $p > \alpha$, fail to reject the null hypothesis.

Chi square test



```r
# chi-squared test in R
data <- data.frame(
  Gender = factor(c("Male", "Female", "Female", "Male", "Male", "Female", "Male", "Female", "Female", "Male")),
  Preference = factor(c("A", "A", "B", "B", "A", "B", "A", "B", "A", "B"))
)
print(data)
```

```
##    Gender Preference
## 1    Male          A
## 2  Female          A
## 3  Female          B
## 4    Male          B
## 5    Male          A
## 6  Female          B
## 7    Male          A
## 8  Female          B
## 9  Female          A
## 10   Male          B
```

```
contingency_table <- table(data$Gender, data$Preference)
print(contingency_table)
```

```
## 
##          A B
##   Female 2 3
##   Male   3 2
```

```
chi_squared_test <- chisq.test(contingency_table)
```

```
## Warning in chisq.test(contingency_table): Chi-squared approximation may be
## incorrect
```

```
print(chi_squared_test)
```

```
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  contingency_table
## X-squared = 0, df = 1, p-value = 1
```

```
chi_squared_test <- chisq.test(contingency_table)
```

```
## Warning in chisq.test(contingency_table): Chi-squared approximation may be
## incorrect
```

```
print(chi_squared_test)
```

Quartiles

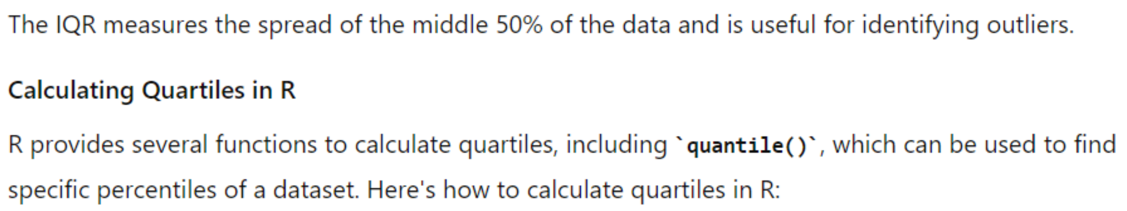
Quartiles are statistical measures that divide a dataset into four equal parts, providing insight into the distribution of the data. The three quartiles—first quartile (Q1), second quartile (Q2), and third quartile (Q3)—help summarize the spread and central tendency of the data. Quartiles are particularly useful in exploratory data analysis and in the context of boxplots, where they provide a visual representation of data distribution.

Definitions of Quartiles

1. **First Quartile (Q1)**: The value below which 25% of the data points fall. It represents the 25th percentile.
2. **Second Quartile (Q2)**: Also known as the median, it is the value that separates the dataset into two equal halves. It represents the 50th percentile.
3. **Third Quartile (Q3)**: The value below which 75% of the data points fall. It represents the 75th percentile.

The interquartile range (IQR) is defined as the difference between the third quartile and the first quartile:

$$\text{IQR} = Q3 - Q1$$

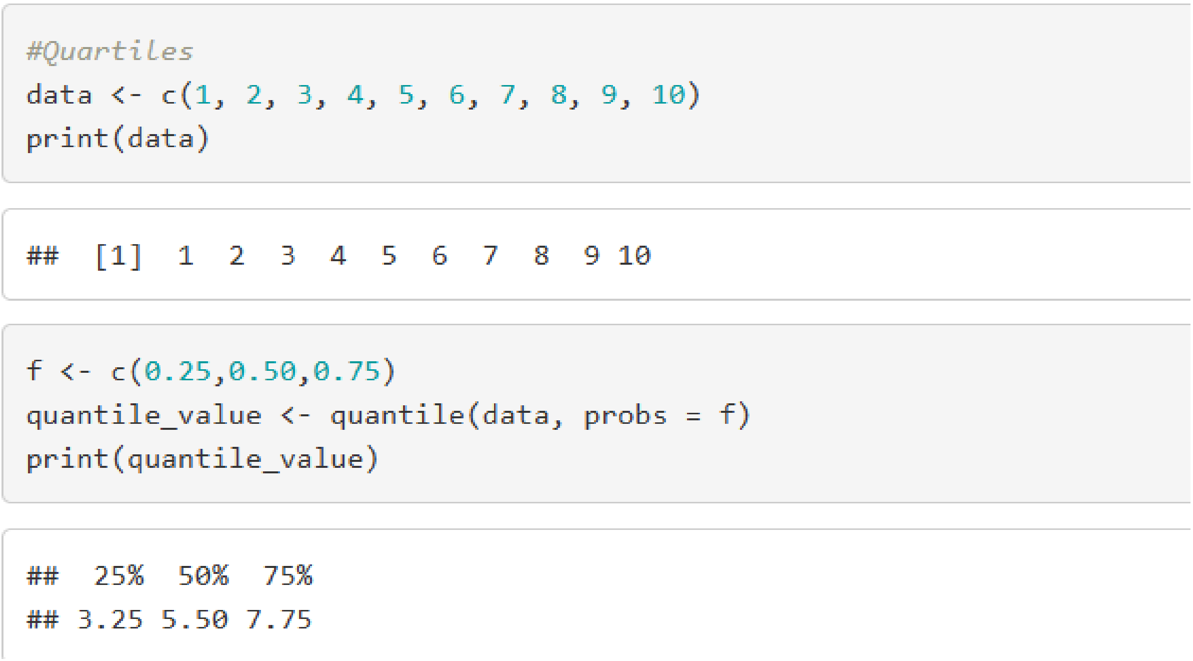


The IQR measures the spread of the middle 50% of the data and is useful for identifying outliers.

Calculating Quartiles in R

R provides several functions to calculate quartiles, including `` `quantile()` ``, which can be used to find specific percentiles of a dataset. Here's how to calculate quartiles in R:

The quantile() function allows you to specify which quartiles you want to calculate. By default, it computes the minimum, 1st quartile (Q1), median (Q2), 3rd quartile (Q3), and maximum values.



```r
#Quartiles
data <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
print(data)
```

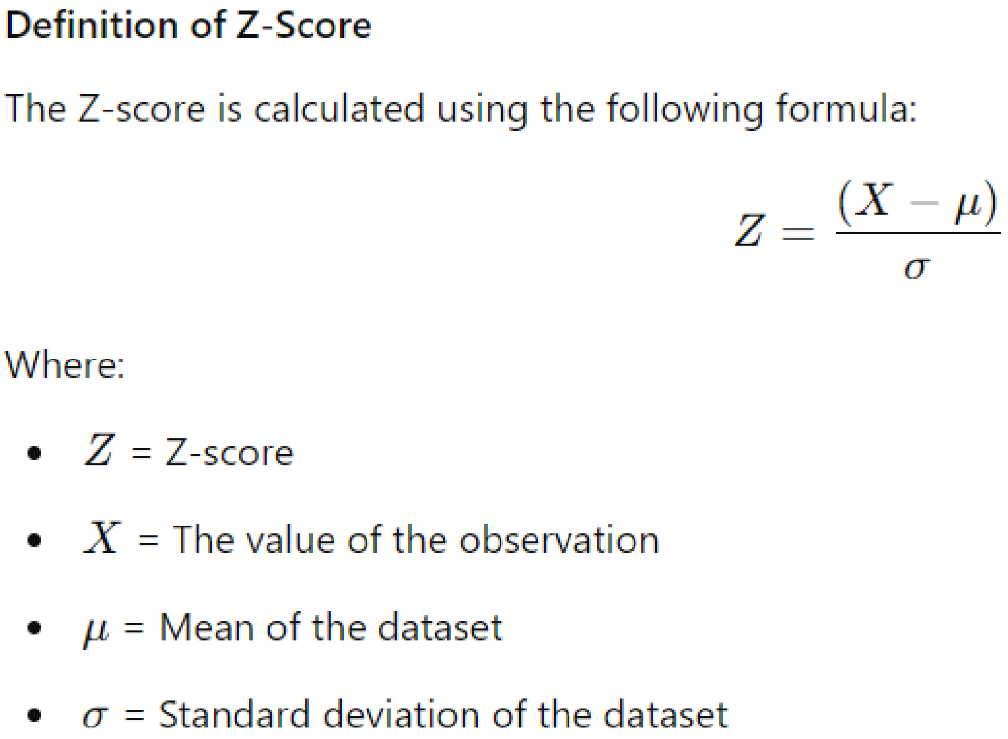
```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
f <- c(0.25,0.50,0.75)
quantile_value <- quantile(data, probs = f)
print(quantile_value)
```

```
##  25%  50%  75% 
## 3.25 5.50 7.75
```

Z- Score

The **Z-score**, also known as the standard score or normalized score, is a statistical measure that quantifies the number of standard deviations a data point is from the mean of a dataset. Z-scores are commonly used in statistical analysis to standardize scores from different distributions, allowing for comparisons across different datasets or variables. The Z-score helps to identify outliers and assess the relative standing of a data point within a distribution.



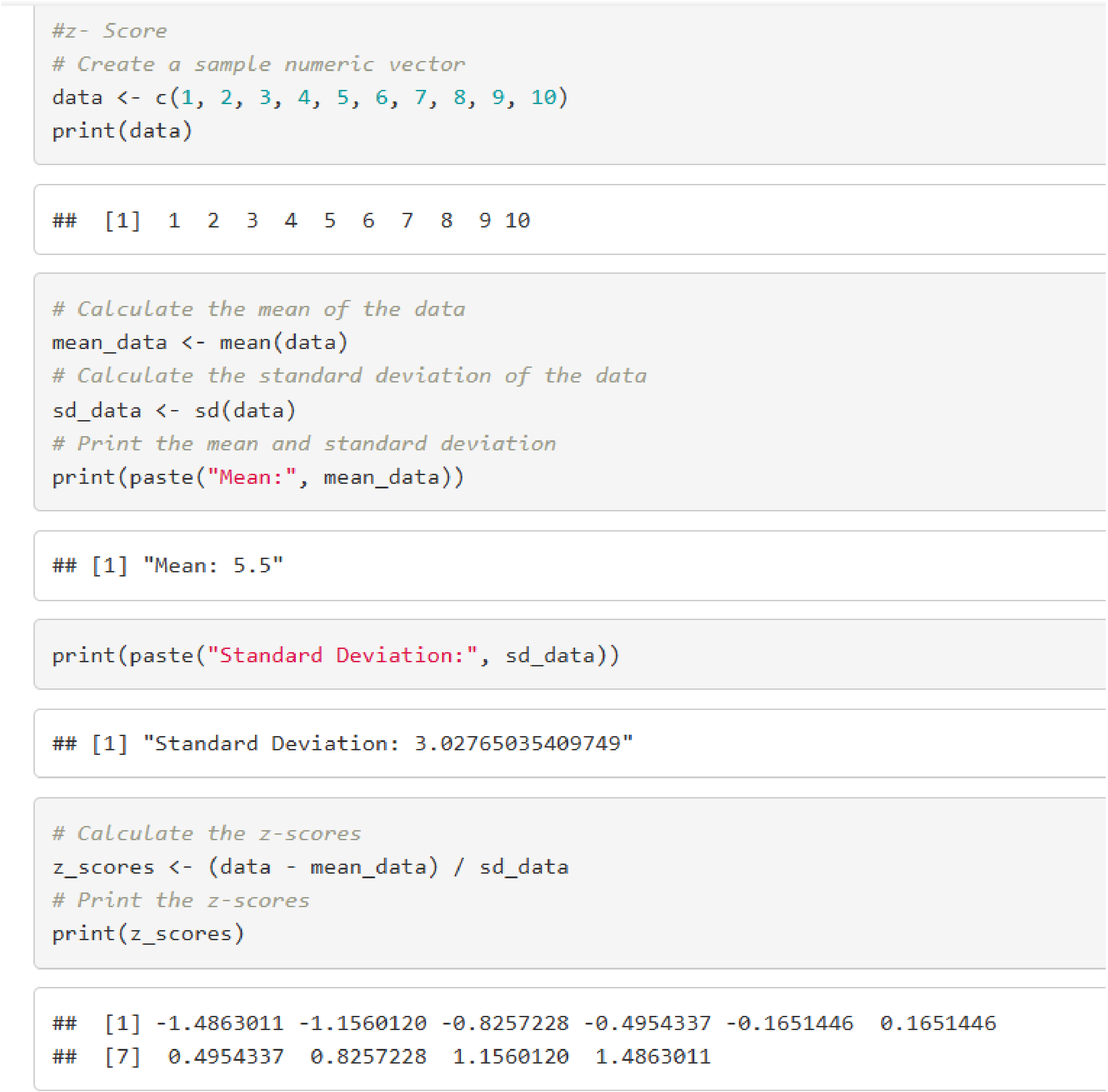
Definition of Z-Score

The Z-score is calculated using the following formula:

$$Z = \frac{(X - \mu)}{\sigma}$$

Where:

- Z = Z-score
- X = The value of the observation
- μ = Mean of the dataset
- σ = Standard deviation of the dataset



```r
#z- Score
# Create a sample numeric vector
data <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
print(data)
```

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

```r
# Calculate the mean of the data
mean_data <- mean(data)
# Calculate the standard deviation of the data
sd_data <- sd(data)
# Print the mean and standard deviation
print(paste("Mean:", mean_data))
```

```
## [1] "Mean: 5.5"
```

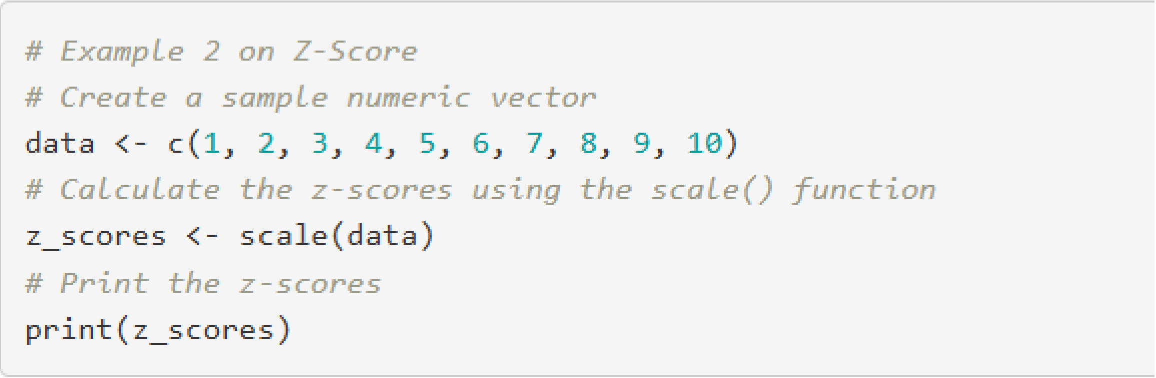
```r
print(paste("Standard Deviation:", sd_data))
```

```
## [1] "Standard Deviation: 3.02765035409749"
```

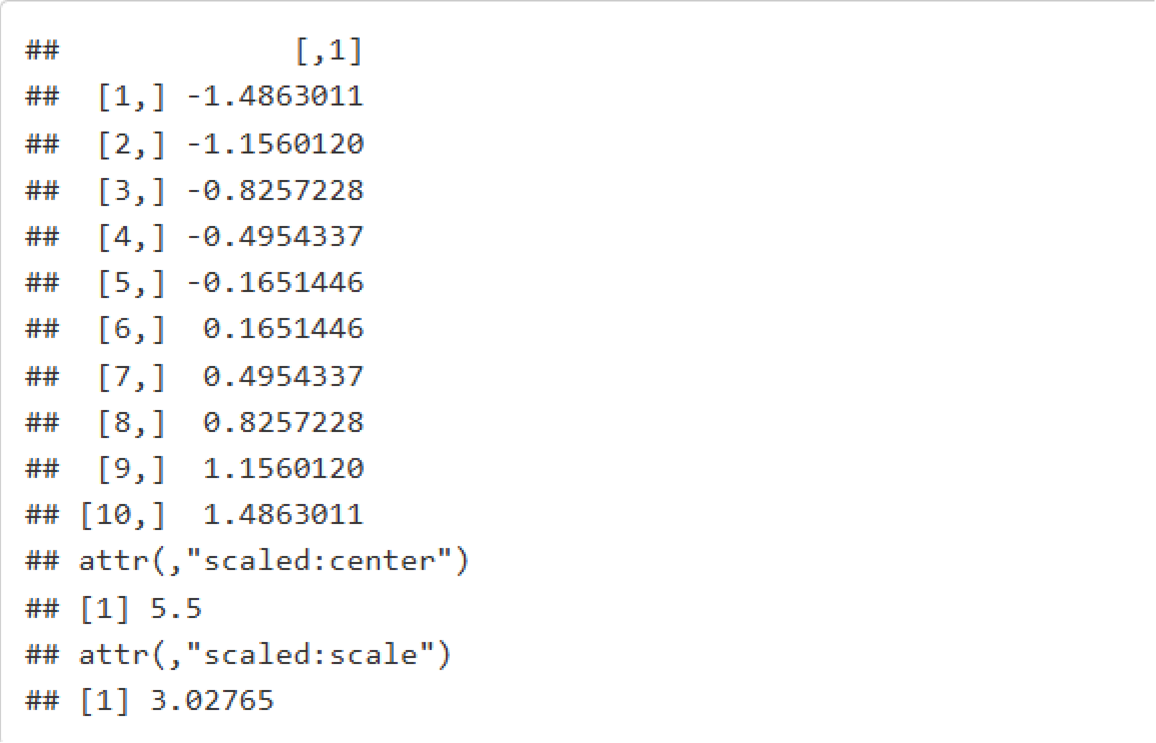
```r
# Calculate the z-scores
z_scores <- (data - mean_data) / sd_data
# Print the z-scores
print(z_scores)
```

```
##  [1] -1.4863011 -1.1560120 -0.8257228 -0.4954337 -0.1651446  0.1651446
##  [7]  0.4954337  0.8257228  1.1560120  1.4863011
```

Example 2:



```
# Example 2 on Z-Score
# Create a sample numeric vector
data <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# Calculate the z-scores using the scale() function
z_scores <- scale(data)
# Print the z-scores
print(z_scores)
```



```
##              [,1]
##  [1,] -1.4863011
##  [2,] -1.1560120
##  [3,] -0.8257228
##  [4,] -0.4954337
##  [5,] -0.1651446
##  [6,]  0.1651446
##  [7,]  0.4954337
##  [8,]  0.8257228
##  [9,]  1.1560120
## [10,]  1.4863011
## attr(,"scaled:center")
## [1] 5.5
## attr(,"scaled:scale")
## [1] 3.02765
```

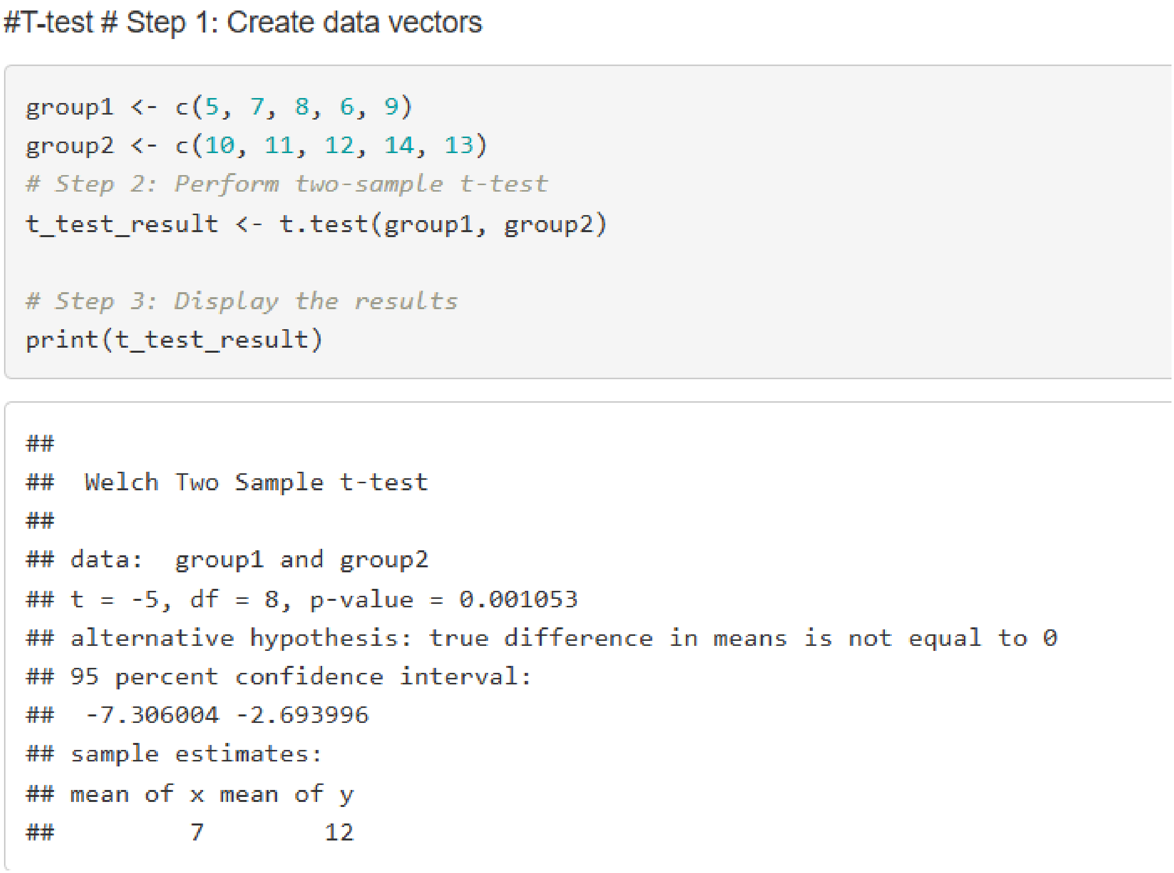
T-test

The **t-test** is a statistical hypothesis test used to determine if there is a significant difference between the means of two groups. It is commonly used when the sample sizes are small and the population standard deviations are unknown. The t-test helps to assess whether the means of two datasets are significantly different from each other.

There are three main types of t-tests:

1. **One-Sample T-Test**: Tests whether the mean of a single sample is significantly different from a known or hypothesized population mean.
2. **Independent Two-Sample T-Test**: Compares the means of two independent groups to determine if they are significantly different from each other.

3. **Paired T-Test**: Compares the means of two related groups (e.g., measurements taken on the same subjects before and after treatment).



#T-test # Step 1: Create data vectors

```
group1 <- c(5, 7, 8, 6, 9)
group2 <- c(10, 11, 12, 14, 13)
# Step 2: Perform two-sample t-test
t_test_result <- t.test(group1, group2)

# Step 3: Display the results
print(t_test_result)
```

```
##
##  Welch Two Sample t-test
##
## data:  group1 and group2
## t = -5, df = 8, p-value = 0.001053
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -7.306004 -2.693996
## sample estimates:
## mean of x mean of y
##         7        12
```

testing sample proportions

Testing sample proportions is a statistical method used to determine if the proportion of a certain outcome in a sample is significantly different from a known or hypothesized population proportion. This is particularly useful in various fields such as social sciences, healthcare, and market research, where researchers often want to compare sample proportions to assess trends, preferences, or changes over time.

Types of Tests for Proportions

1. **One-Sample Proportion Test**: Used to determine if the proportion of successes in a single sample differs from a specified population proportion.
2. **Two-Sample Proportion Test**: Compares the proportions of successes between two independent groups to see if they are significantly different from each other.

1. One-Sample Proportion Test

The one-sample proportion test evaluates whether the proportion of a success in a single sample significantly differs from a hypothesized population proportion.

Hypotheses

- **Null Hypothesis (H_0)**: The sample proportion is equal to the population proportion ($p = p_0$).
- **Alternative Hypothesis (H_a)**: The sample proportion is not equal to the population proportion ($p \neq p_0$).

Formula

The test statistic for the one-sample proportion test is calculated using the formula:

$$z = \frac{\hat{p} - p_0}{\sqrt{\frac{p_0(1-p_0)}{n}}}$$

Where:

- $\hat{p}$ = Sample proportion
- p_0 = Population proportion
- n = Sample size

testing normality

Normality testing is a statistical procedure used to determine whether a dataset follows a normal distribution (Gaussian distribution). Normal distribution is a foundational assumption for many statistical analyses and tests, including t-tests, ANova, and linear regression. Ensuring that the data is normally distributed is crucial because the validity of these tests relies on this assumption. When the assumption of normality is violated, the results of statistical tests may be unreliable or misleading.

Importance of Testing Normality

1. **Assumption Verification**: Many parametric statistical tests assume that the data follows a normal distribution. Verifying this assumption is essential for accurate analysis.
2. **Choice of Statistical Tests**: If the data is not normally distributed, researchers may need to choose non-parametric tests or transform the data to meet normality assumptions.
3. **Understanding Data Characteristics**: Assessing normality can provide insights into the underlying characteristics of the data, including potential outliers or skewness.

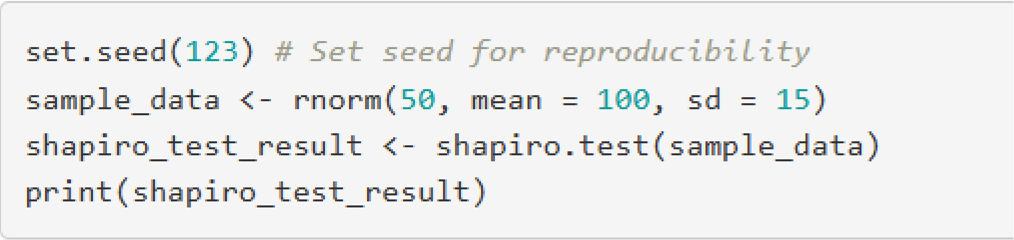
Methods for Testing Normality

There are several statistical tests and visual methods for assessing normality:

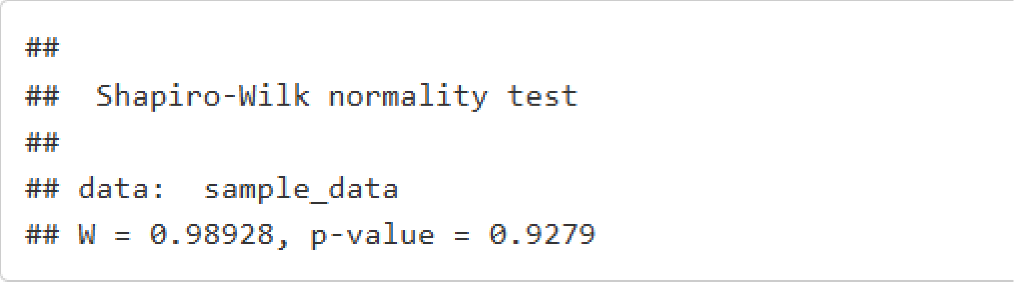
1. **Graphical Methods**:
   - **Histogram**: A histogram can provide a visual representation of the data distribution. A bell-shaped histogram indicates normality.

   - **Q-Q Plot (Quantile-Quantile Plot)**: A Q-Q plot compares the quantiles of the dataset against the quantiles of a normal distribution. If the points lie approximately along a straight line, the data is likely normally distributed.
   - **Boxplot**: A boxplot can highlight the symmetry and spread of the data, indicating potential deviations from normality.

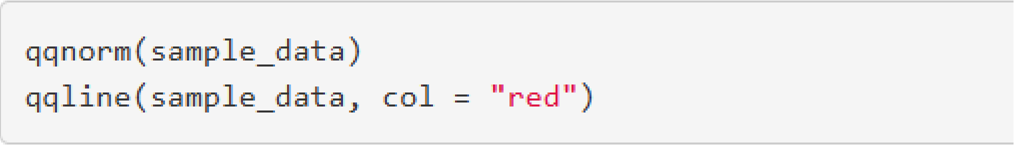
2. **Statistical Tests**:
   - **Shapiro-Wilk Test**: A widely used test for normality that tests the null hypothesis that the data is normally distributed. A significant p-value (typically < 0.05) indicates a departure from normality.
   - **Kolmogorov-Smirnov Test**: Compares the empirical distribution function of the sample with the cumulative distribution function of a normal distribution.
   - **Anderson-Darling Test**: A modification of the Kolmogorov-Smirnov test that gives more weight to the tails of the distribution, making it more sensitive to deviations from normality in the tails.
   - **D'Agostino's K-squared Test**: Tests for skewness and kurtosis to assess normality.



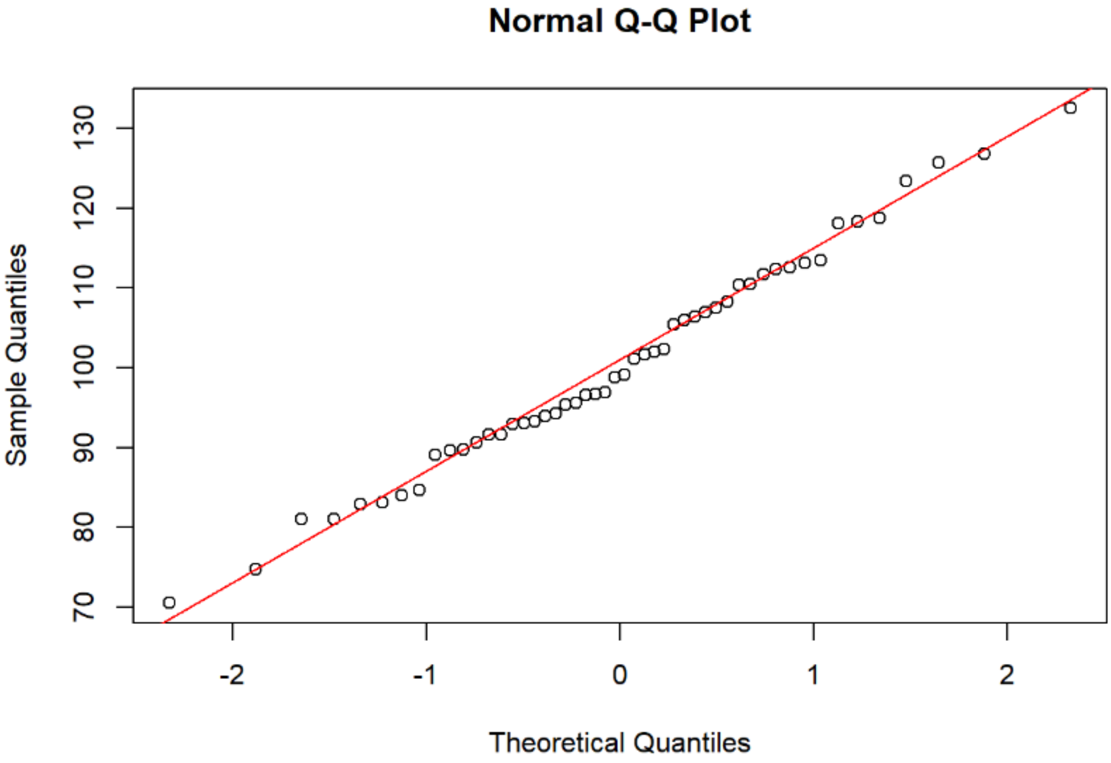
```
set.seed(123) # Set seed for reproducibility
sample_data <- rnorm(50, mean = 100, sd = 15)
shapiro_test_result <- shapiro.test(sample_data)
print(shapiro_test_result)
```



```
##
##  Shapiro-Wilk normality test
##
## data:  sample_data
## W = 0.98928, p-value = 0.9279
```



```
qqnorm(sample_data)
qqline(sample_data, col = "red")
```



comparing means of two samples

Comparing the means of two samples is a fundamental statistical procedure used to determine whether there is a significant difference between the means of two groups. This analysis is widely employed in various fields, including medicine, social sciences, marketing, and psychology, to assess the effectiveness of treatments, interventions, or to understand differences between populations.

Types of Comparisons

There are different scenarios when comparing means of two samples:

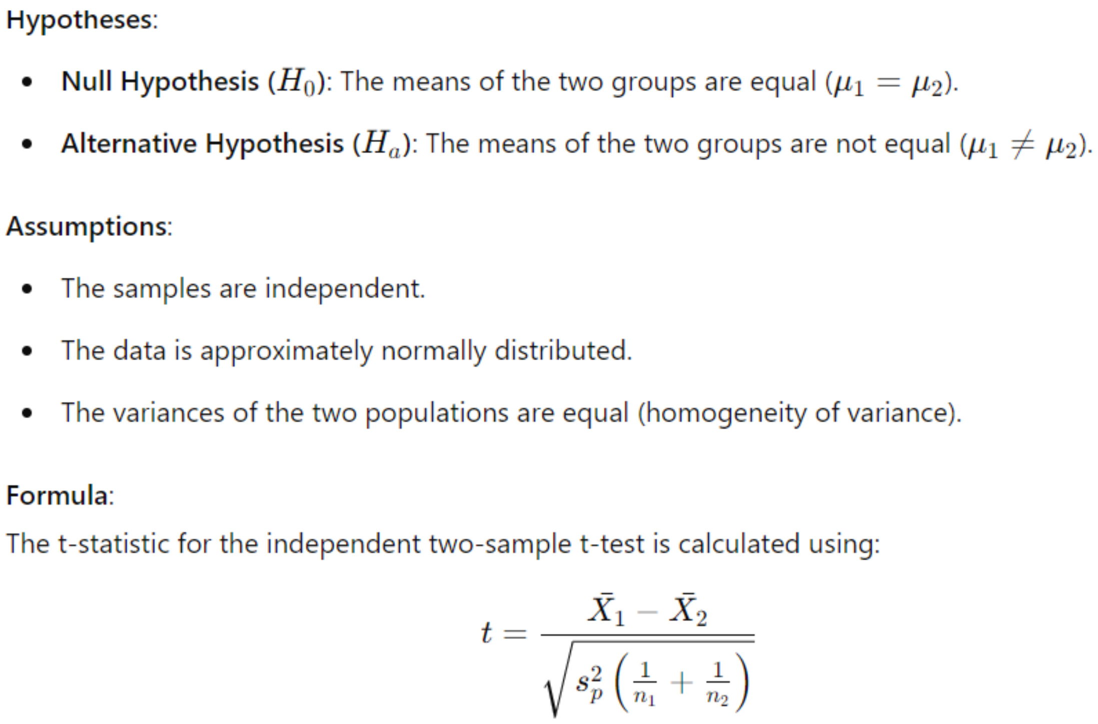
1. **Independent Samples**: When the two samples are drawn from different populations and are not related.
2. **Paired Samples**: When the two samples are related or matched in some way, such as before-and-after measurements on the same subjects.

Statistical Tests for Comparing Means

Several statistical tests can be used to compare means, depending on the data characteristics and the design of the study.

1. Independent Two-Sample T-Test

The independent two-sample t-test is used when comparing the means of two independent groups.



Hypotheses:

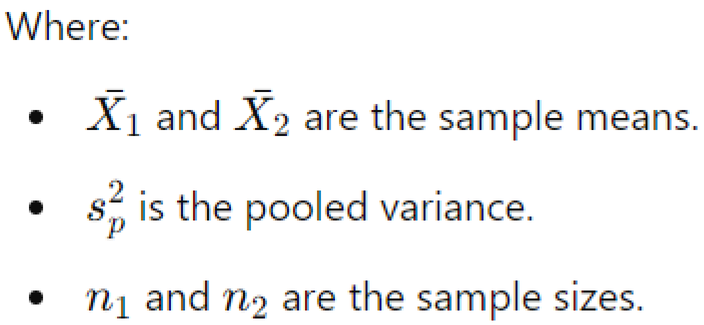
- **Null Hypothesis** (H_0): The means of the two groups are equal ($\mu_1 = \mu_2$).
- **Alternative Hypothesis** (H_a): The means of the two groups are not equal ($\mu_1 \neq \mu_2$).

Assumptions:

- The samples are independent.
- The data is approximately normally distributed.
- The variances of the two populations are equal (homogeneity of variance).

Formula:
The t-statistic for the independent two-sample t-test is calculated using:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{s_p^2 \left(\frac{1}{n_1} + \frac{1}{n_2} \right)}}$$



Where:

- $\bar{X}_1$ and $\bar{X}_2$ are the sample means.
- s_p^2 is the pooled variance.
- n_1 and n_2 are the sample sizes.

testing correlation for significance

Correlation measures the strength and direction of the relationship between two variables. It provides valuable insights into how changes in one variable may relate to changes in another. However, merely observing a correlation is not enough; it is crucial to test the significance of the correlation to determine whether the observed relationship is statistically significant or could have occurred by chance.

Importance of Testing Correlation for Significance

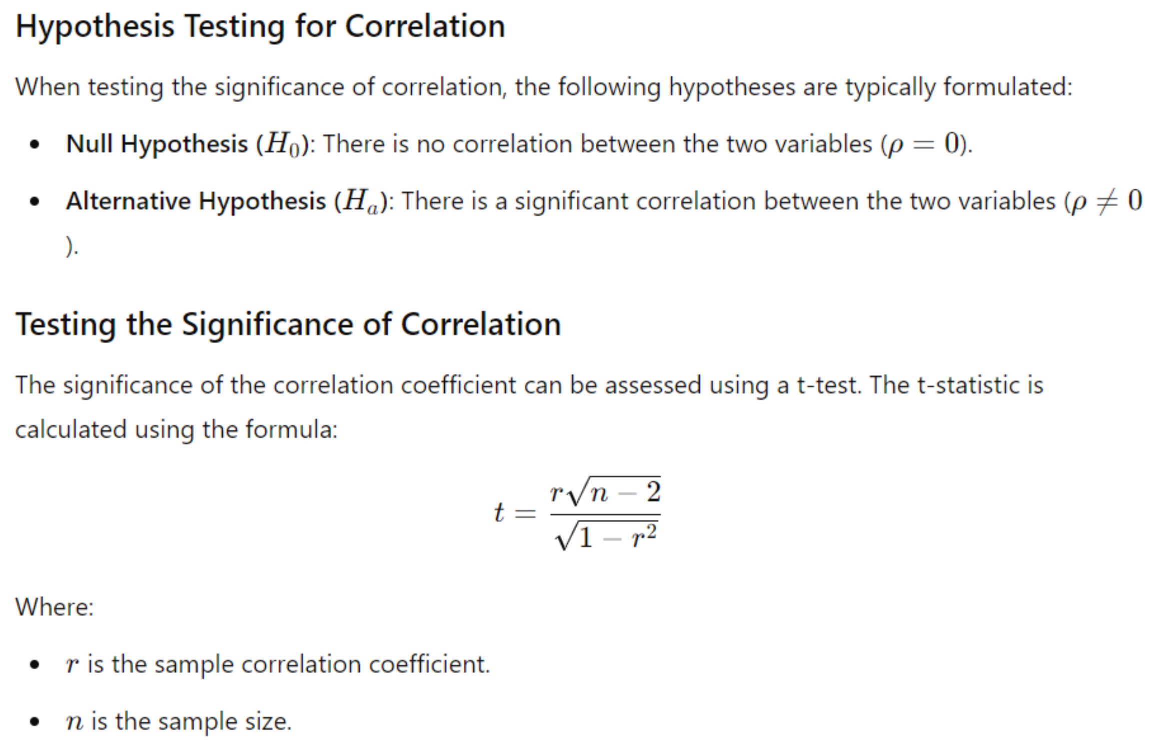
1. **Understanding Relationships**: Assessing the significance of correlation helps to establish whether a true relationship exists between variables, which is critical for making informed decisions in research and practice.
2. **Guiding Further Research**: Significant correlations may indicate areas for further investigation, helping researchers identify variables that warrant deeper exploration.

3. **Supporting Hypotheses**: Testing correlation significance can provide statistical support for hypotheses in various fields, including psychology, economics, and health sciences.

Types of Correlation Coefficients

The most commonly used correlation coefficients are:

1. **Pearson Correlation Coefficient (r)**: Measures the linear relationship between two continuous variables. It assumes that both variables are normally distributed and that the relationship is linear.
2. **Spearman Rank Correlation Coefficient (ρ)**: A non-parametric measure of correlation that assesses how well the relationship between two variables can be described using a monotonic function. It is used when the data do not meet the assumptions of Pearson's correlation.
3. **Kendall's Tau (τ)**: Another non-parametric measure of correlation that evaluates the strength of association between two variables by considering the ranks of data.



Hypothesis Testing for Correlation

When testing the significance of correlation, the following hypotheses are typically formulated:

- **Null Hypothesis** (H_0): There is no correlation between the two variables ($\rho = 0$).
- **Alternative Hypothesis** (H_a): There is a significant correlation between the two variables ($\rho \neq 0$).

Testing the Significance of Correlation

The significance of the correlation coefficient can be assessed using a t-test. The t-statistic is calculated using the formula:

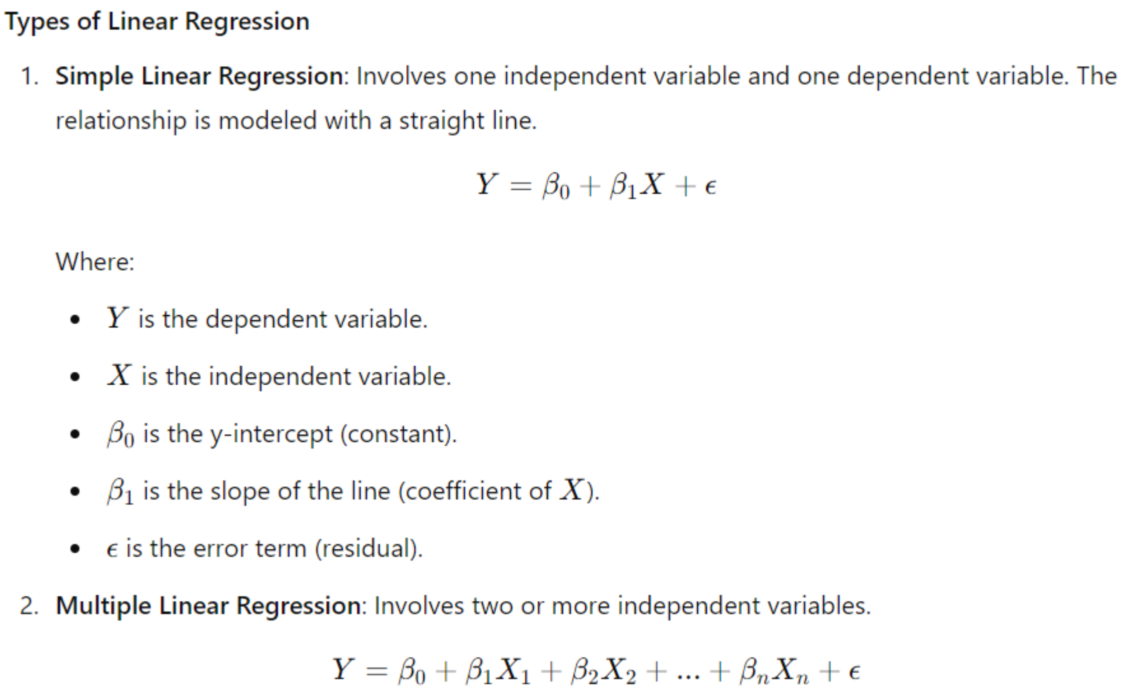
$$t = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}}$$

Where:

- r is the sample correlation coefficient.
- n is the sample size.

The degrees of freedom (df) for the test is n−2n - 2n−2. A critical value from the t-distribution can be compared to the calculated t-statistic to determine significance.

Linear regression is a statistical method used to model the relationship between a dependent variable (response variable) and one or more independent variables (predictors). It aims to find the best-fitting linear equation that describes how the independent variables influence the dependent variable. Linear regression is widely used in various fields, including economics, biology, engineering, and social sciences, for prediction, inference, and understanding relationships.



Types of Linear Regression

1. **Simple Linear Regression**: Involves one independent variable and one dependent variable. The relationship is modeled with a straight line.

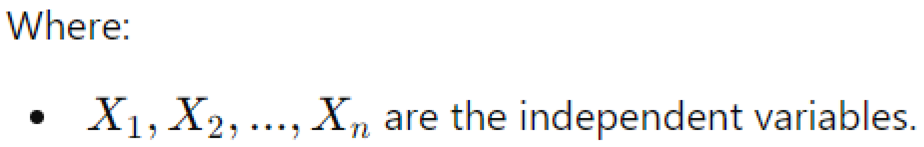
   $$Y = \beta_0 + \beta_1 X + \epsilon$$

   Where:

   - Y is the dependent variable.
   - X is the independent variable.
   - β_0 is the y-intercept (constant).
   - β_1 is the slope of the line (coefficient of X).
   - ϵ is the error term (residual).

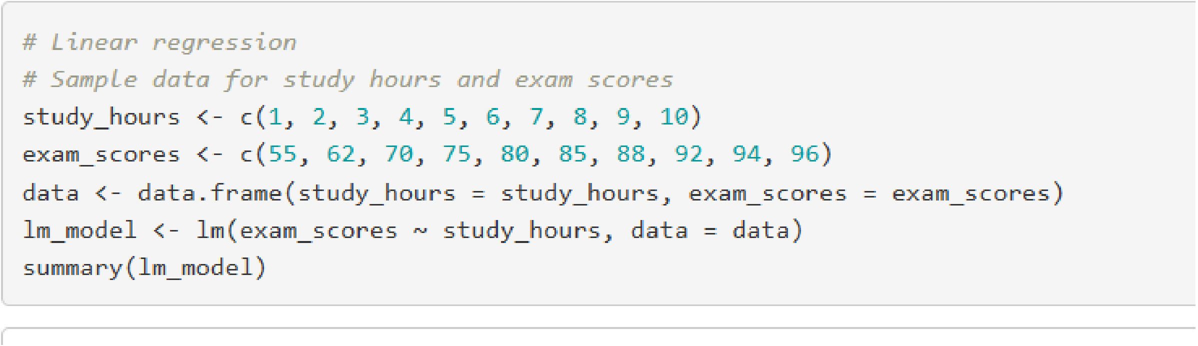
2. **Multiple Linear Regression**: Involves two or more independent variables.

   $$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n + \epsilon$$

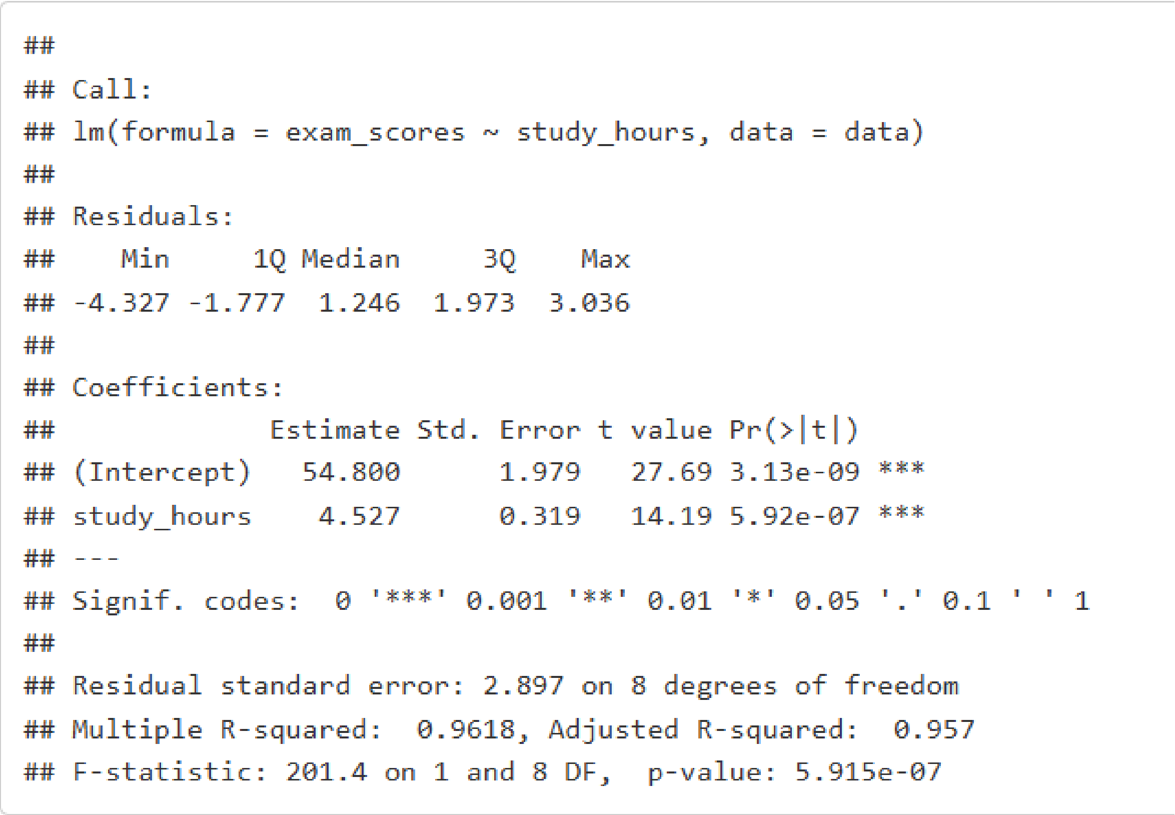


Where:

- $X_1, X_2, ..., X_n$ are the independent variables.



```
# Linear regression
# Sample data for study hours and exam scores
study_hours <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
exam_scores <- c(55, 62, 70, 75, 80, 85, 88, 92, 94, 96)
data <- data.frame(study_hours = study_hours, exam_scores = exam_scores)
lm_model <- lm(exam_scores ~ study_hours, data = data)
summary(lm_model)
```



```
##
## Call:
## lm(formula = exam_scores ~ study_hours, data = data)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -4.327 -1.777  1.246  1.973  3.036
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   54.800      1.979   27.69 3.13e-09 ***
## study_hours    4.527      0.319   14.19 5.92e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.897 on 8 degrees of freedom
## Multiple R-squared:  0.9618, Adjusted R-squared:  0.957
## F-statistic: 201.4 on 1 and 8 DF,  p-value: 5.915e-07
```

Logistic Regression in R Programming

Logistic regression is a statistical method used for modeling the relationship between a binary dependent variable and one or more independent variables. It is a type of regression analysis used when the dependent variable is categorical, specifically when it has two possible outcomes (e.g., success/failure, yes/no, 1/0). Logistic regression estimates the probability that a given observation falls into one of the categories based on the values of the independent variables.

Importance of Logistic Regression

1. **Binary Outcome Prediction**: Logistic regression is widely used in scenarios where the outcome is binary, making it suitable for various fields, such as medicine (disease presence/absence), finance (default/no default), and social sciences (vote/no vote).
2. **Interpretability**: The coefficients obtained from logistic regression can be interpreted in terms of odds ratios, providing insights into how changes in independent variables affect the probability of the outcome.
3. **Probabilistic Interpretation**: Logistic regression estimates probabilities, making it possible to assess the likelihood of different outcomes based on predictor variables.
4. **Handling Non-linear Relationships**: While logistic regression is a linear model, it can effectively handle non-linear relationships through transformations of the independent variables.

Logistic Regression Model

The logistic regression model uses the logistic function (also known as the sigmoid function) to model the probability of the binary outcome. The equation for the logistic regression model is:

$$P(Y=1|X) = \frac{1}{1+e^{-(\beta_0+\beta_1 X_1+\beta_2 X_2+\ldots+\beta_n X_n)}}$$

Where:

- $P(Y=1|X)$ is the probability of the outcome being 1 given the independent variables X.
- β_0 is the intercept (constant term).
- $\beta_1, \beta_2, \ldots, \beta_n$ are the coefficients for the independent variables $X_1, X_2, \ldots, X_n$.
- e is the base of the natural logarithm.

Estimating Coefficients

The coefficients of the logistic regression model are estimated using the method of **maximum likelihood estimation (MLE)**. This approach finds the values of the coefficients that maximize the likelihood of the observed data under the model.

Assumptions of Logistic Regression

1. **Binary Dependent Variable**: The outcome variable must be binary.
2. **Independence of Observations**: The observations must be independent of each other.
3. **No Multicollinearity**: The independent variables should not be highly correlated with each other.
4. **Linearity of Logit**: The log odds of the dependent variable should be linearly related to the independent variables.

Fitting a Logistic Regression Model in R

R provides a straightforward way to perform logistic regression using the `` `glm()` `` function, specifying the family as `` `binomial` ``. Below is an example of how to fit a logistic regression model, interpret the results, and evaluate model performance.

```
study_hours <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
pass_fail <- c(0, 0, 0, 0, 1, 1, 1, 1, 1, 1)
logit_model <- glm(pass_fail ~ study_hours, data = data, family = binomial)
```

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
summary(logit_model)
```

```
##
## Call:
## glm(formula = pass_fail ~ study_hours, family = binomial, data = data)
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   -200.37  265802.23  -0.001    0.999
## study_hours     44.52   58511.58   0.001    0.999
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1.3460e+01  on 9  degrees of freedom
## Residual deviance: 8.6042e-10  on 8  degrees of freedom
## AIC: 4
##
## Number of Fisher Scoring iterations: 25
```

clustering with R.

Clustering is a type of unsupervised machine learning technique used to group similar data points into clusters. It aims to identify patterns or structures in the data without prior labels, making it valuable for exploratory data analysis, pattern recognition, and feature engineering. Clustering is widely used in various fields, including marketing, biology, social sciences, and image processing.

Importance of Clustering

1. **Data Exploration**: Clustering helps in exploring the inherent structure of data, providing insights into groupings that might not be immediately apparent.

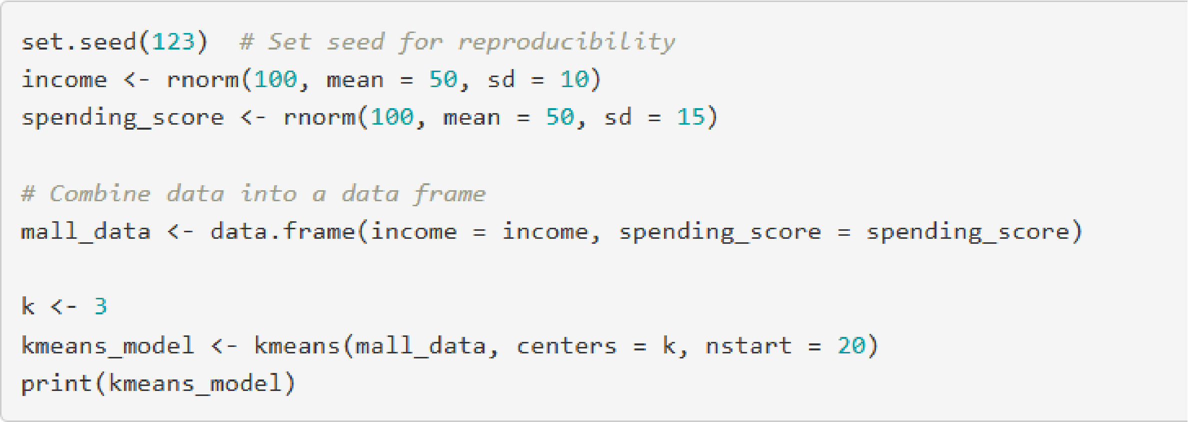
2. **Segmentation**: It allows for segmentation of data into distinct groups, which can be useful for targeted marketing, customer segmentation, or identifying distinct biological species.

3. **Anomaly Detection**: Clustering can assist in detecting outliers or anomalies in the dataset by identifying data points that do not belong to any cluster.

4. **Dimensionality Reduction**: Clustering can help reduce the dimensionality of the data by summarizing it into clusters, making it easier to visualize and analyze.

There are various clustering algorithms, each with its own approach and characteristics. Some of the most commonly used algorithms include:

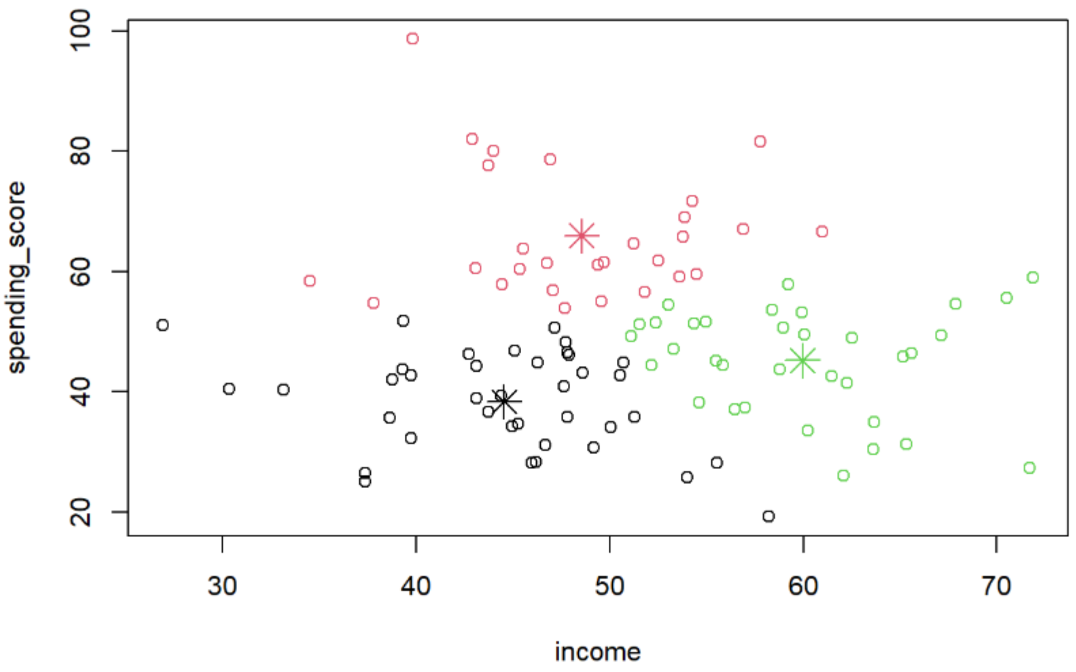
1. **K-Means Clustering**:
    - K-Means is one of the most popular clustering algorithms.
    - It partitions data into kkk clusters, where each data point belongs to the cluster with the nearest mean.
    - The algorithm iteratively updates the cluster centroids and reassigns data points until convergence.
    - K-Means is sensitive to the initial placement of centroids and may converge to local minima.



```r
set.seed(123)  # Set seed for reproducibility
income <- rnorm(100, mean = 50, sd = 10)
spending_score <- rnorm(100, mean = 50, sd = 15)

# Combine data into a data frame
mall_data <- data.frame(income = income, spending_score = spending_score)

k <- 3
kmeans_model <- kmeans(mall_data, centers = k, nstart = 20)
print(kmeans_model)
```

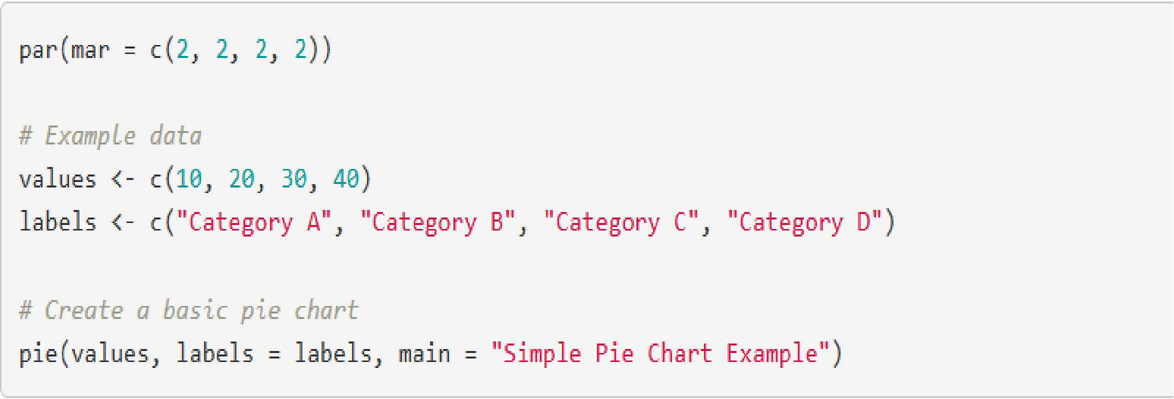
```
## K-means clustering with 3 clusters of sizes 38, 28, 34
##
## Cluster means:
##     income spending_score
## 1 44.53374       38.32162
## 2 48.55266       65.89596
## 3 59.96027       45.21681
##
## Clustering vector:
##   [1] 1 2 3 1 1 3 3 1 1 2 3 2 1 3 2 3 3 1 3 1 1 1 1 1 2 1 3 3 1 3 2 2 3 3 1 2 1
##  [38] 2 2 1 2 1 1 3 3 1 1 2 2 1 2 2 2 3 1 3 2 3 2 3 2 1 1 2 1 3 2 1 3 3 1 1 3 2
##  [75] 1 3 1 2 2 1 1 2 1 3 1 3 2 3 2 3 3 3 3 1 3 2 3 3 1 1
##
## Within cluster sum of squares by cluster:
## [1] 4041.416 4063.051 3712.325
##  (between_SS / total_SS =  59.4 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

```
plot(mall_data, col = kmeans_model$cluster)
points(kmeans_model$centers, col = 1:k, pch = 8, cex = 2)
```

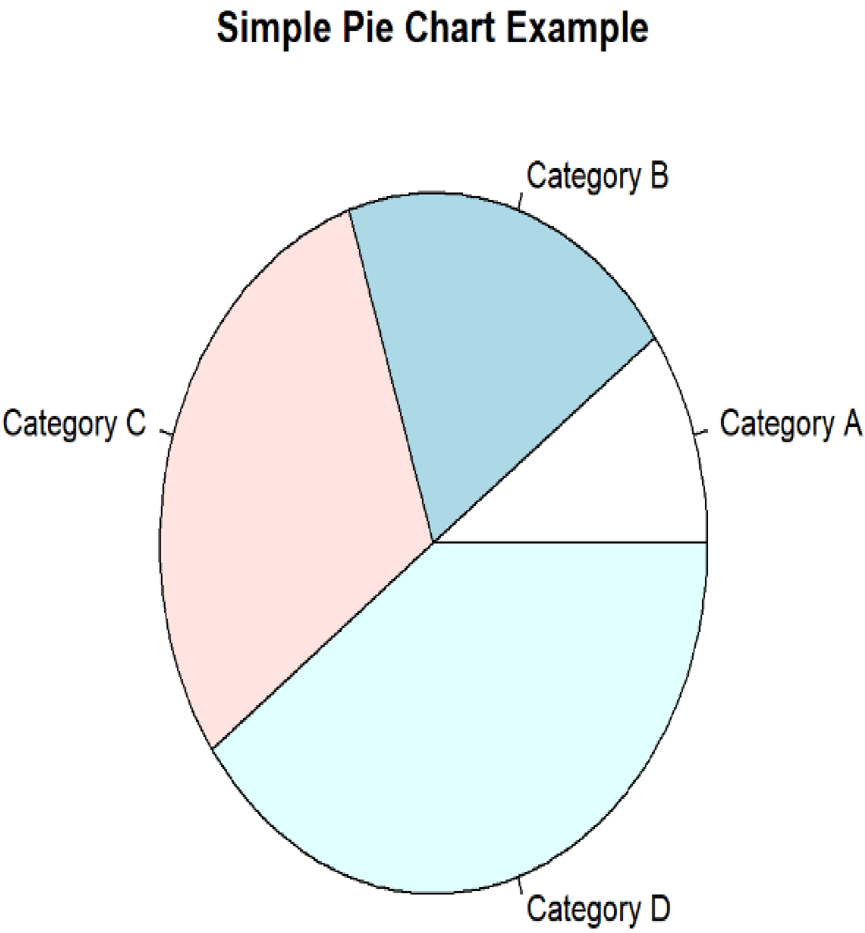


UNIT-V

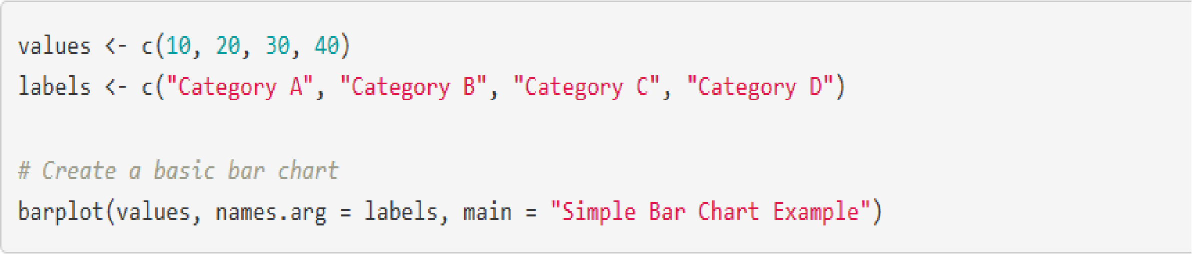
Pie chart



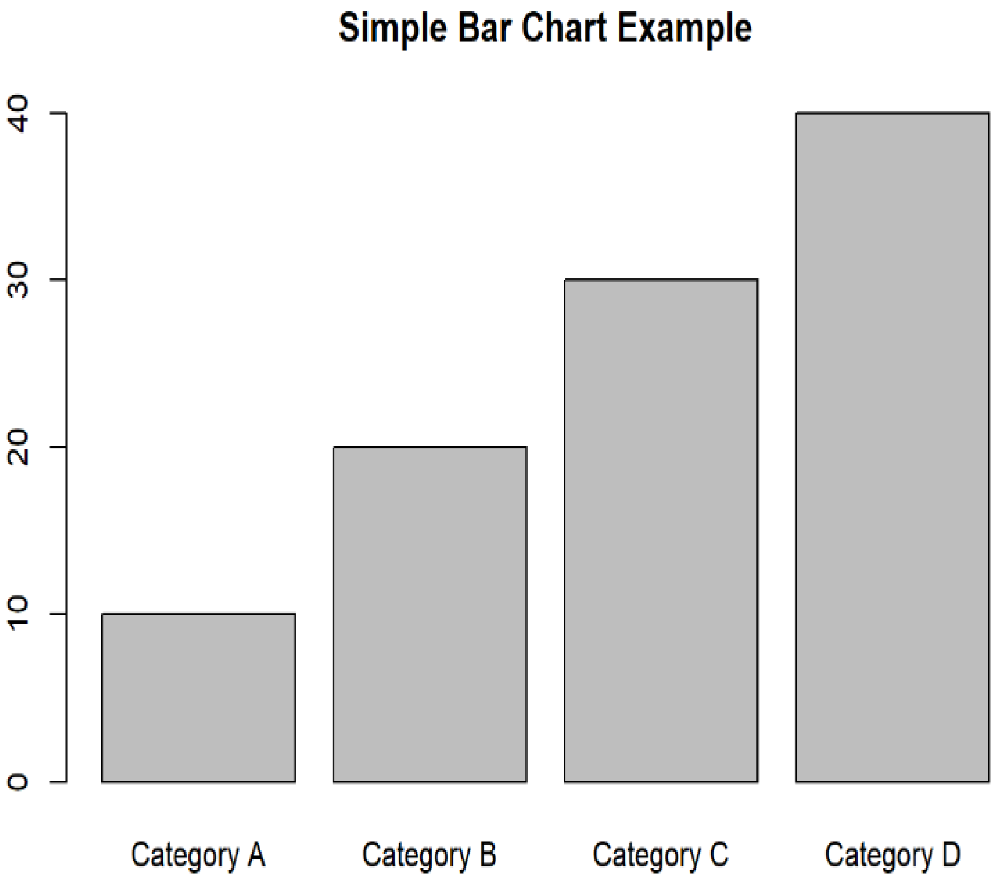
```r
par(mar = c(2, 2, 2, 2))

# Example data
values <- c(10, 20, 30, 40)
labels <- c("Category A", "Category B", "Category C", "Category D")

# Create a basic pie chart
pie(values, labels = labels, main = "Simple Pie Chart Example")
```

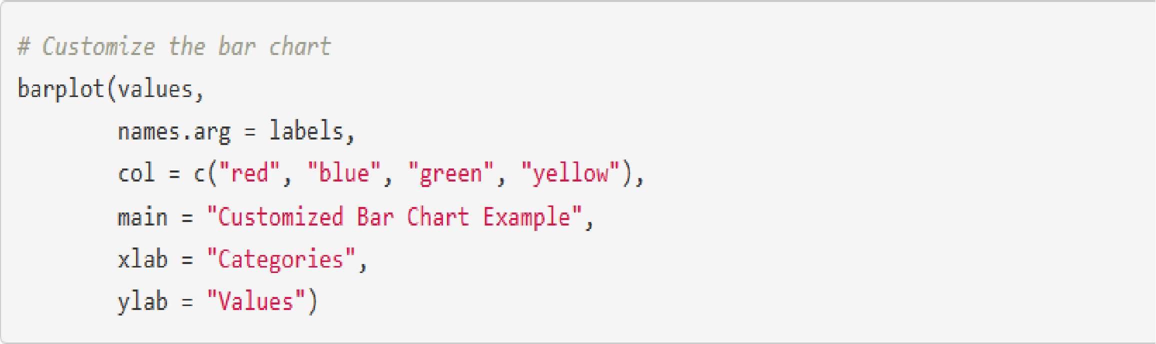


Bar Chart

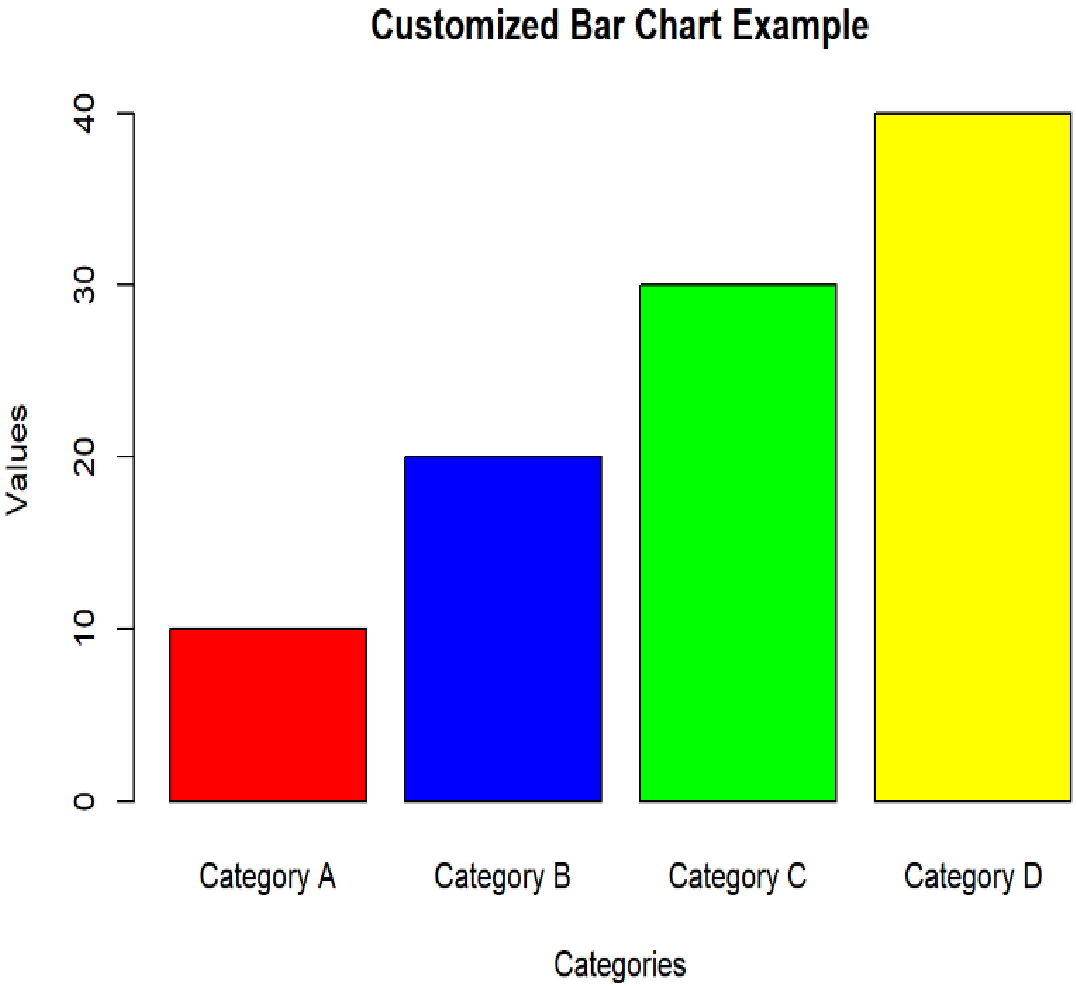


```
values <- c(10, 20, 30, 40)
labels <- c("Category A", "Category B", "Category C", "Category D")

# Create a basic bar chart
barplot(values, names.arg = labels, main = "Simple Bar Chart Example")
```

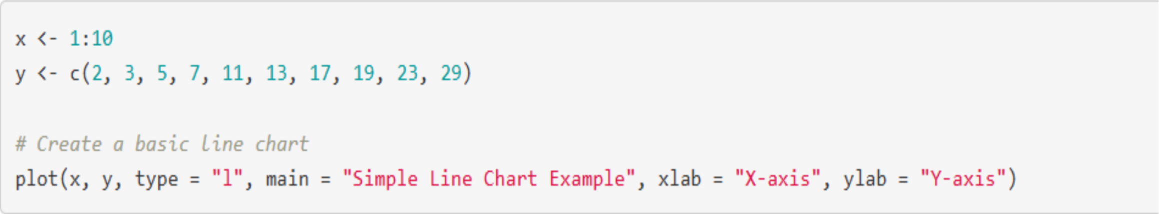




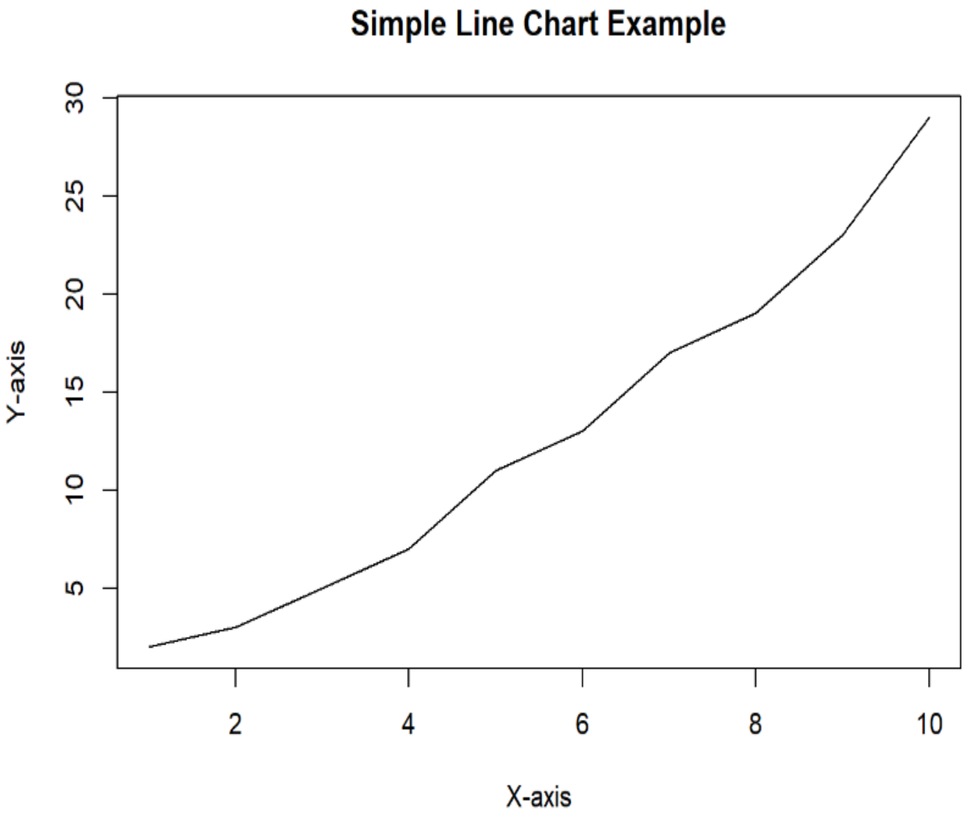
```r
# Customize the bar chart
barplot(values,
        names.arg = labels,
        col = c("red", "blue", "green", "yellow"),
        main = "Customized Bar Chart Example",
        xlab = "Categories",
        ylab = "Values")
```

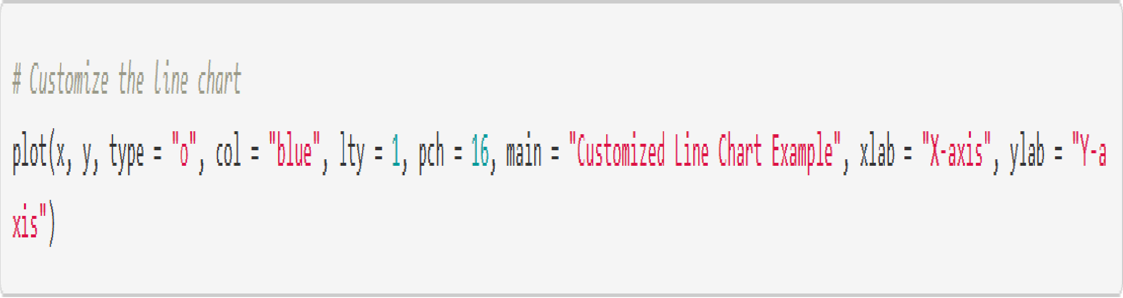


Line chart

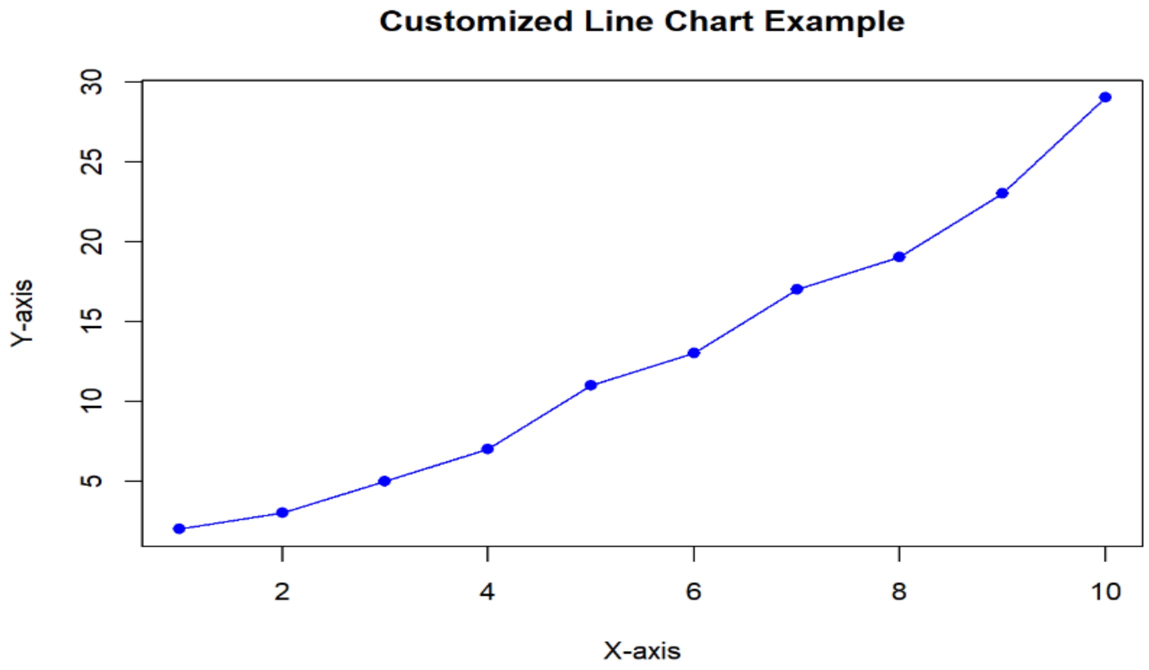


```
x <- 1:10
y <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)

# Create a basic line chart
plot(x, y, type = "l", main = "Simple Line Chart Example", xlab = "X-axis", ylab = "Y-axis")
```

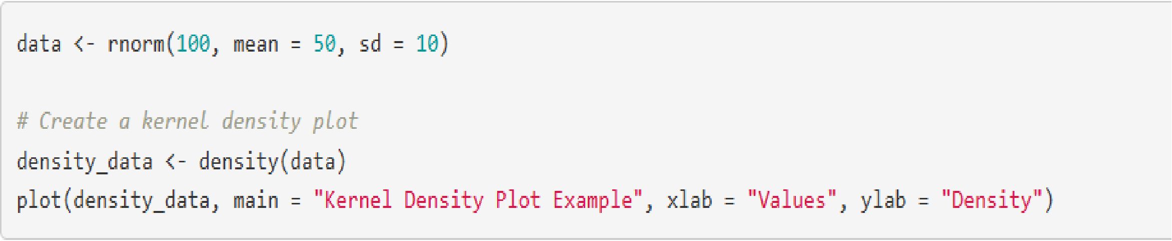




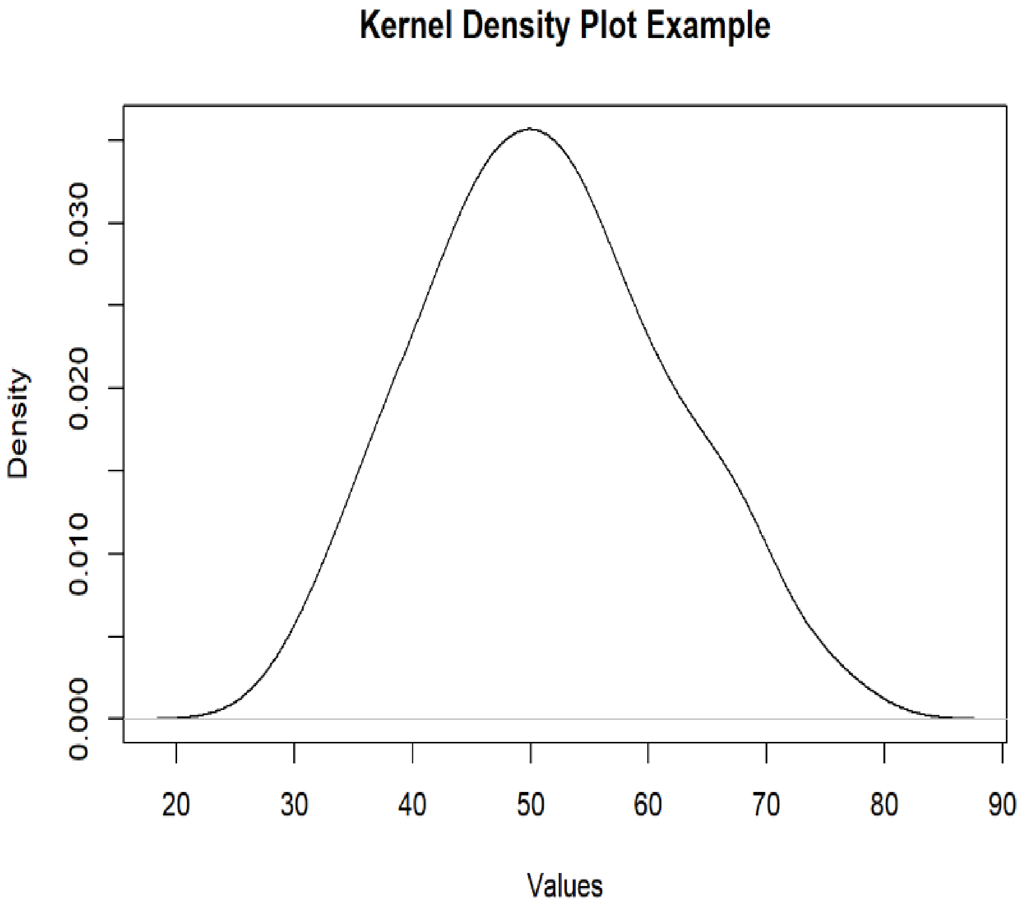
```
# Customize the line chart
plot(x, y, type = "o", col = "blue", lty = 1, pch = 16, main = "Customized Line Chart Example", xlab = "X-axis", ylab = "Y-a
xis")
```

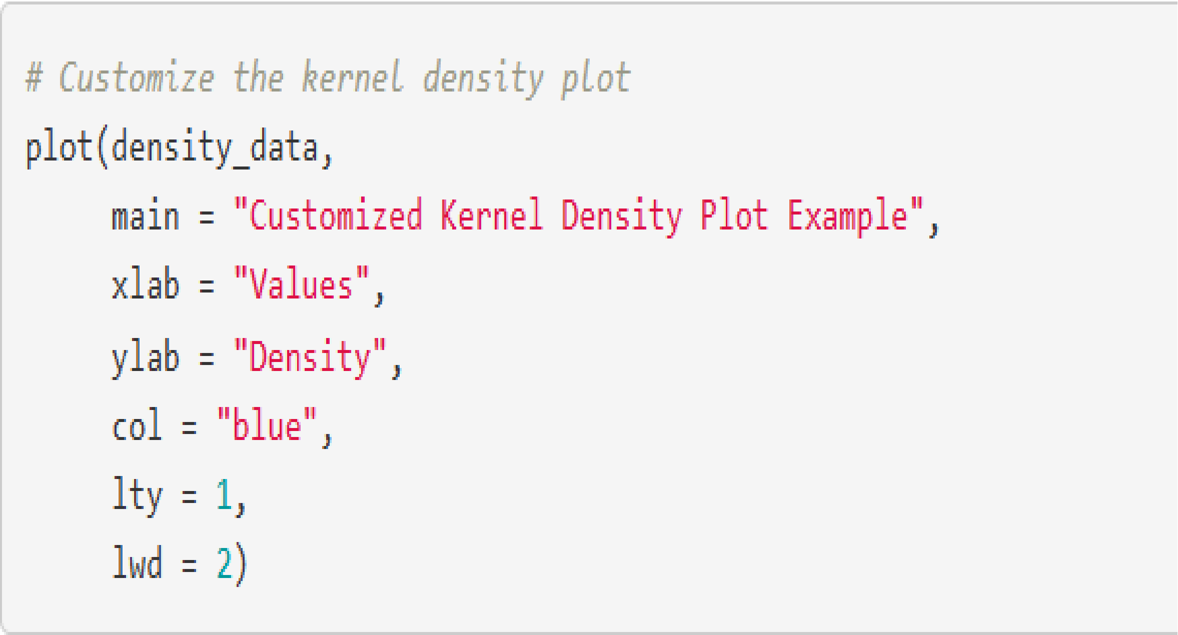


Kernal density plot

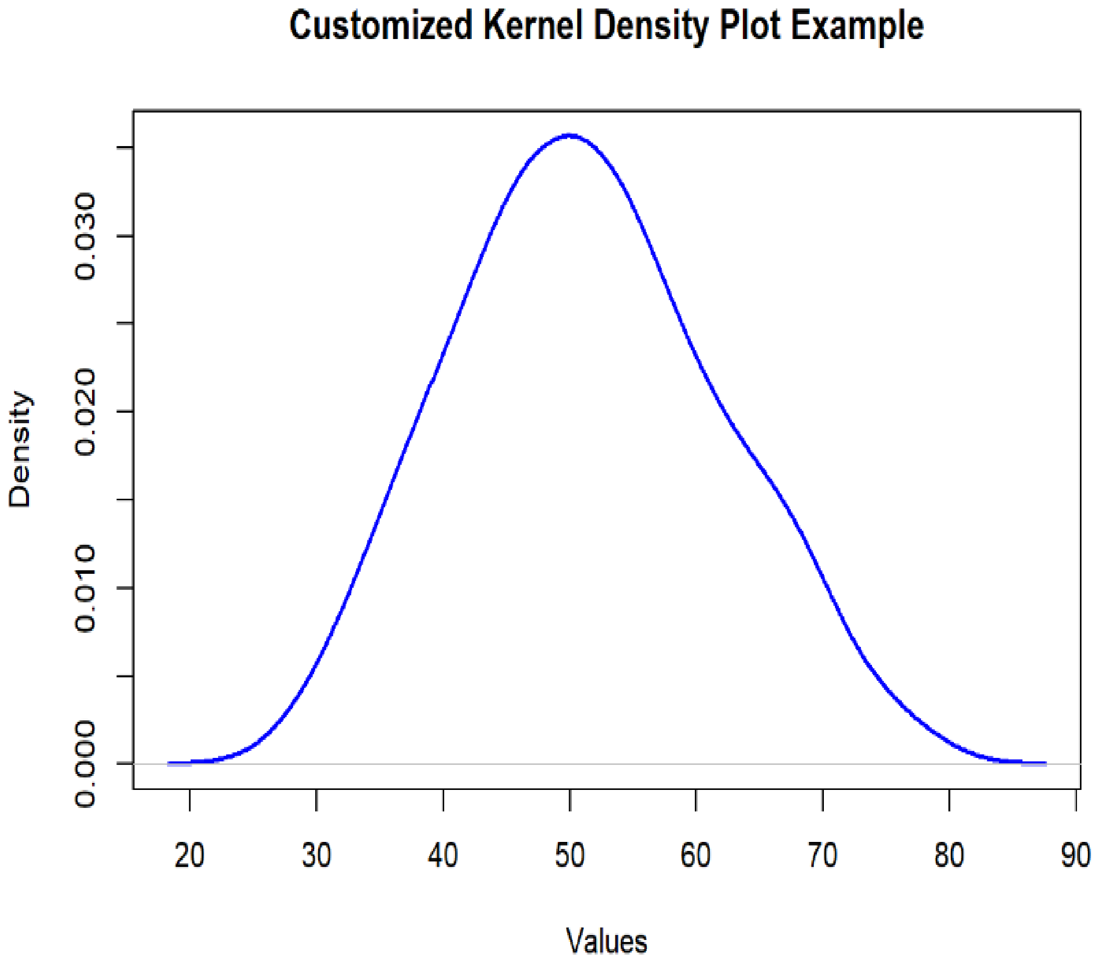


```
data <- rnorm(100, mean = 50, sd = 10)

# Create a kernel density plot
density_data <- density(data)
plot(density_data, main = "Kernel Density Plot Example", xlab = "Values", ylab = "Density")
```

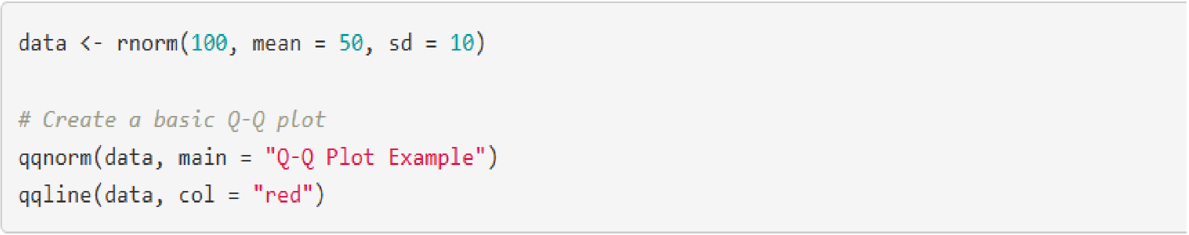




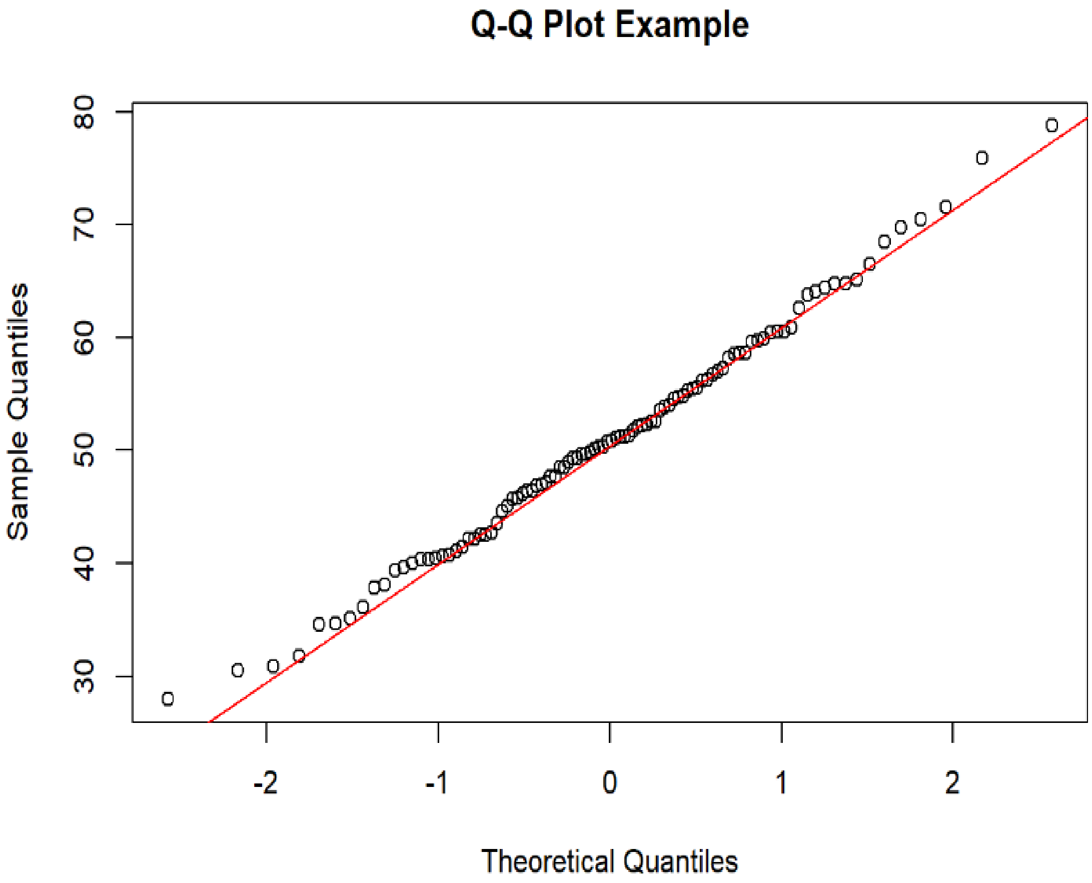
```r
# Customize the kernel density plot
plot(density_data,
     main = "Customized Kernel Density Plot Example",
     xlab = "Values",
     ylab = "Density",
     col = "blue",
     lty = 1,
     lwd = 2)
```

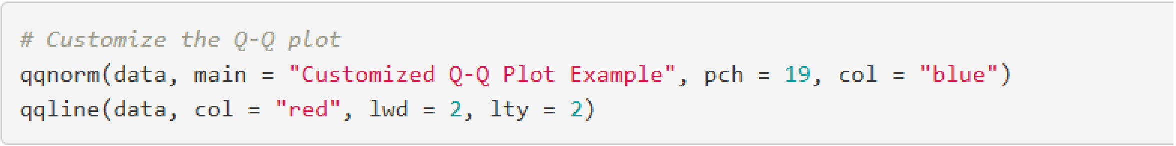


Q Q plot

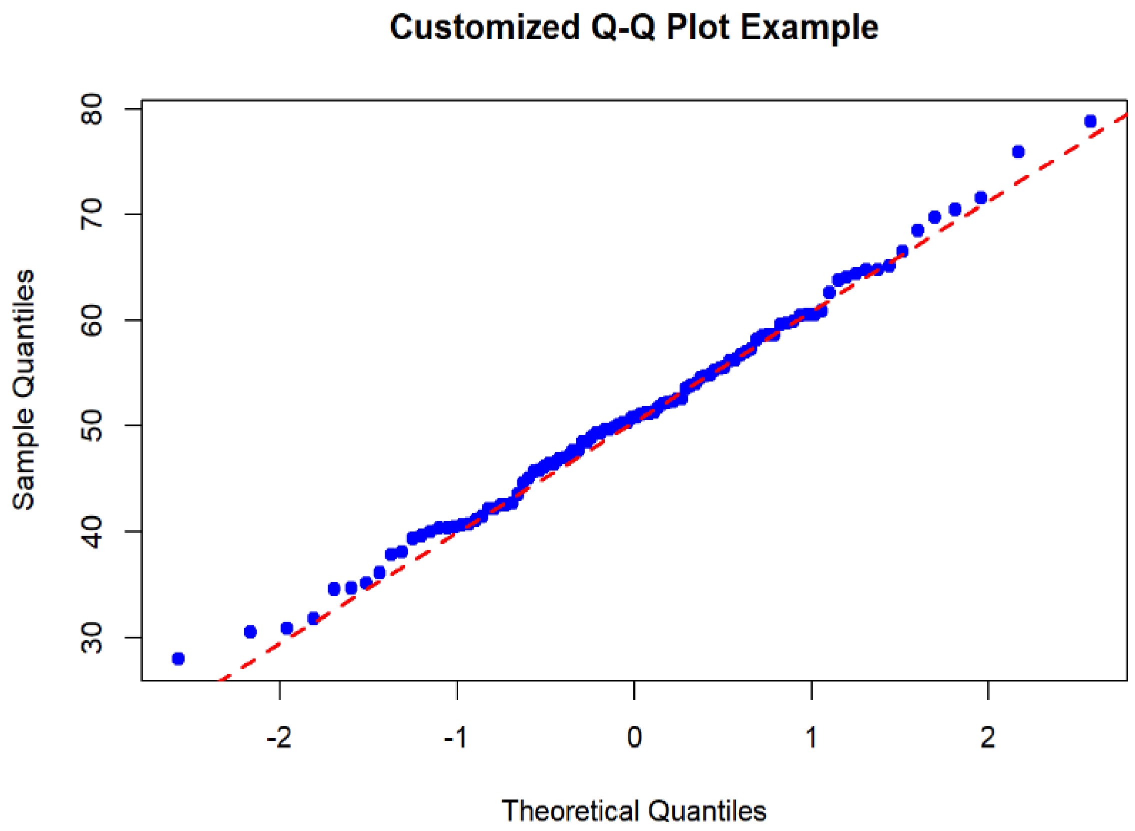


```
data <- rnorm(100, mean = 50, sd = 10)

# Create a basic Q-Q plot
qqnorm(data, main = "Q-Q Plot Example")
qqline(data, col = "red")
```

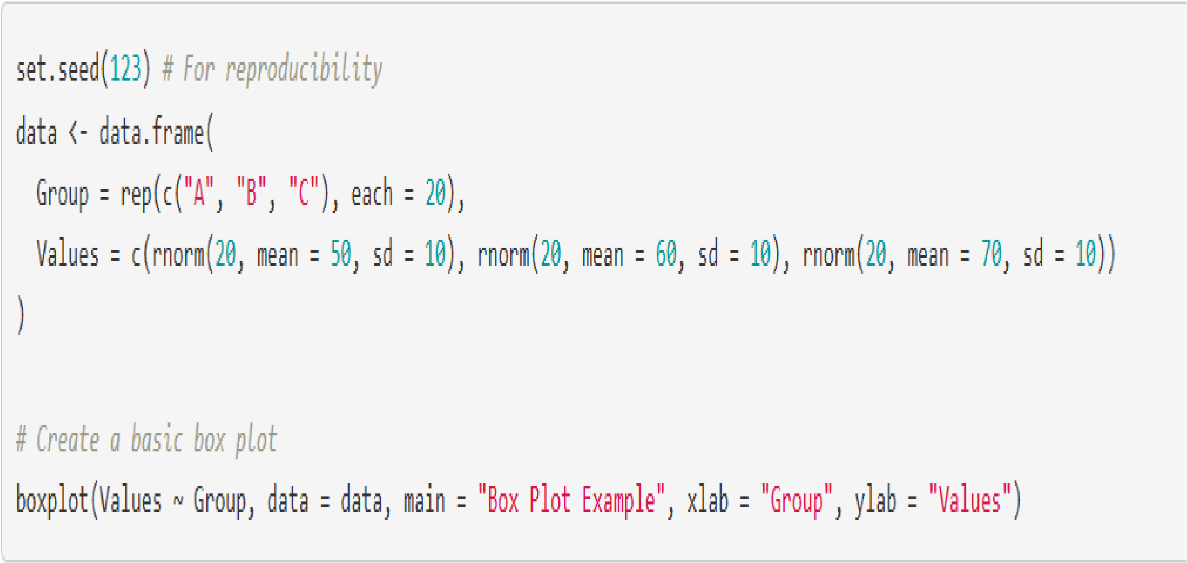




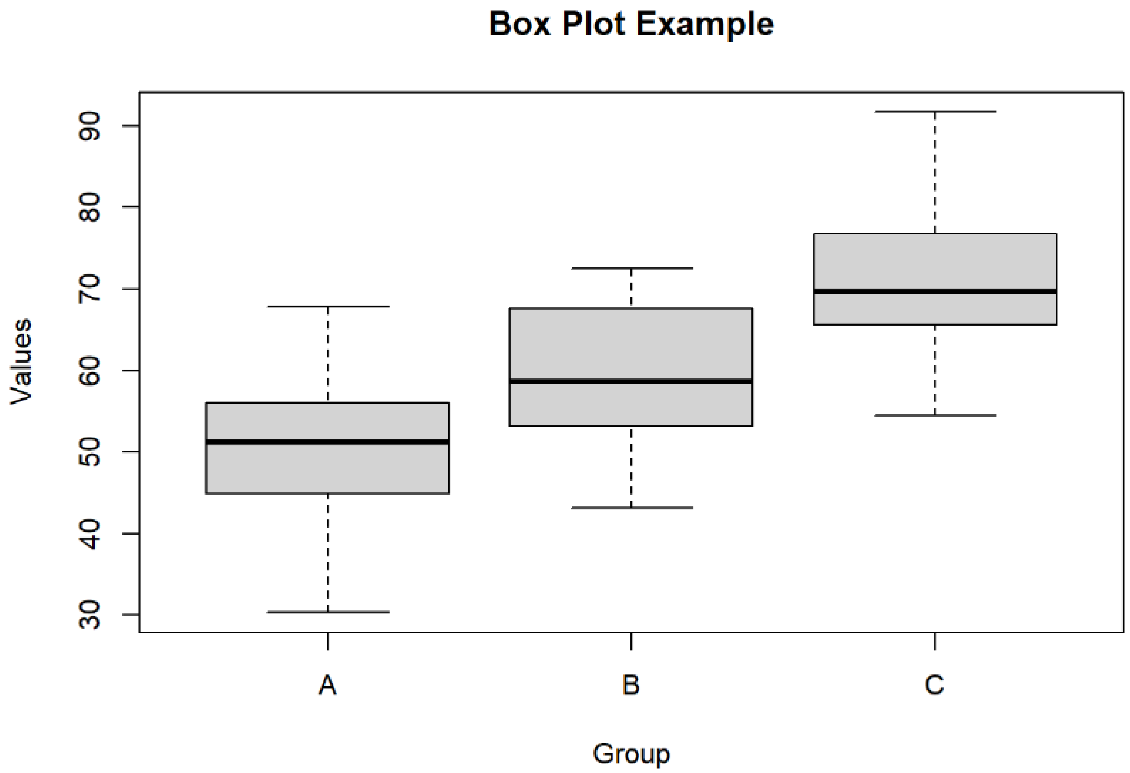
```
# Customize the Q-Q plot
qqnorm(data, main = "Customized Q-Q Plot Example", pch = 19, col = "blue")
qqline(data, col = "red", lwd = 2, lty = 2)
```

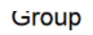


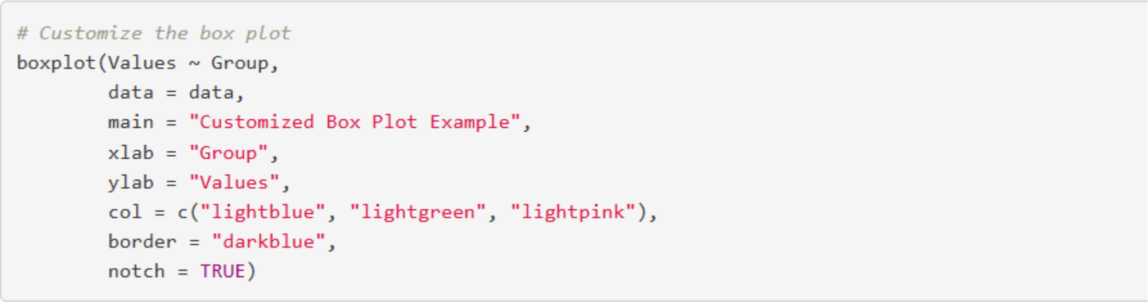
Box Plot



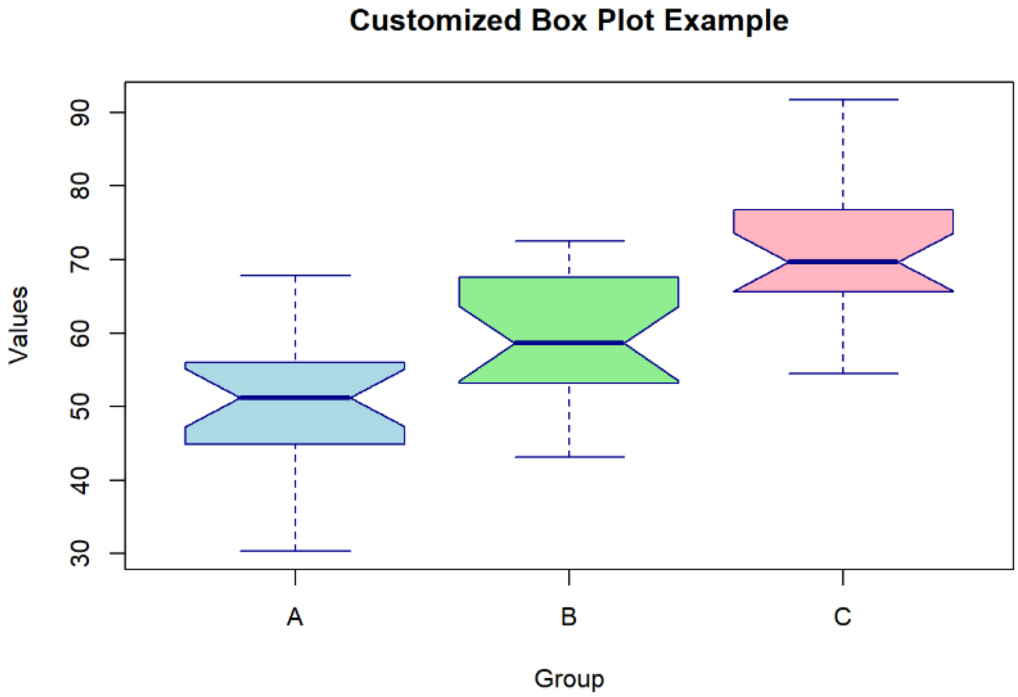
```
set.seed(123) # For reproducibility
data <- data.frame(
  Group = rep(c("A", "B", "C"), each = 20),
  Values = c(rnorm(20, mean = 50, sd = 10), rnorm(20, mean = 60, sd = 10), rnorm(20, mean = 70, sd = 10))
)

# Create a basic box plot
boxplot(Values ~ Group, data = data, main = "Box Plot Example", xlab = "Group", ylab = "Values")
```


Box Plot Example
Values
90
80
70
60
50
40
30
A
B
C
Group

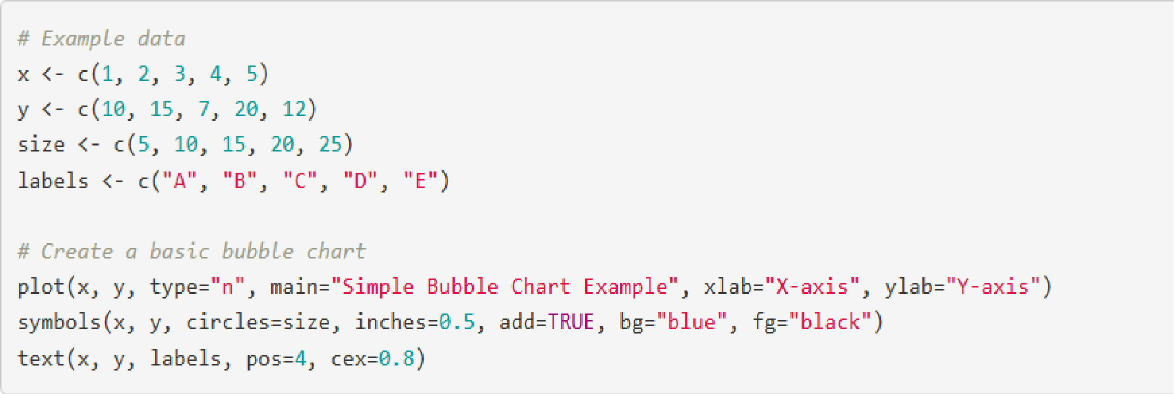




```
# Customize the box plot
boxplot(Values ~ Group,
        data = data,
        main = "Customized Box Plot Example",
        xlab = "Group",
        ylab = "Values",
        col = c("lightblue", "lightgreen", "lightpink"),
        border = "darkblue",
        notch = TRUE)
```



Bubble chart



```r
# Example data
x <- c(1, 2, 3, 4, 5)
y <- c(10, 15, 7, 20, 12)
size <- c(5, 10, 15, 20, 25)
labels <- c("A", "B", "C", "D", "E")

# Create a basic bubble chart
plot(x, y, type="n", main="Simple Bubble Chart Example", xlab="X-axis", ylab="Y-axis")
symbols(x, y, circles=size, inches=0.5, add=TRUE, bg="blue", fg="black")
text(x, y, labels, pos=4, cex=0.8)
```

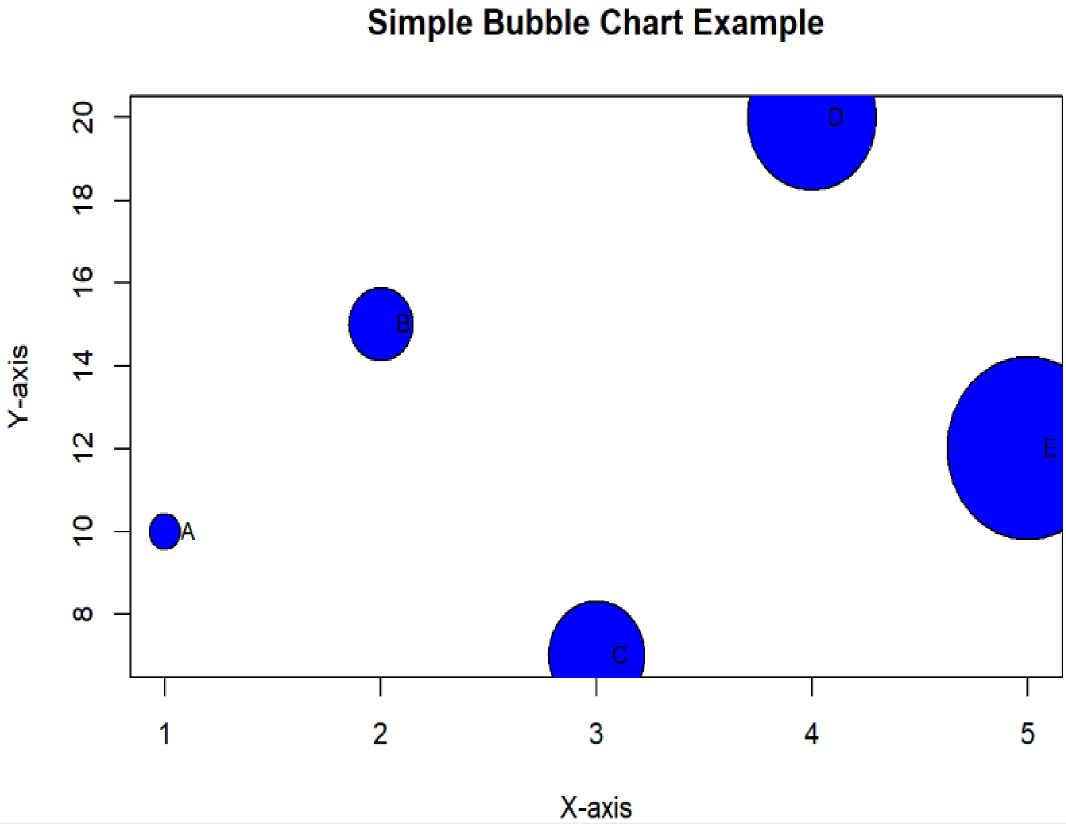
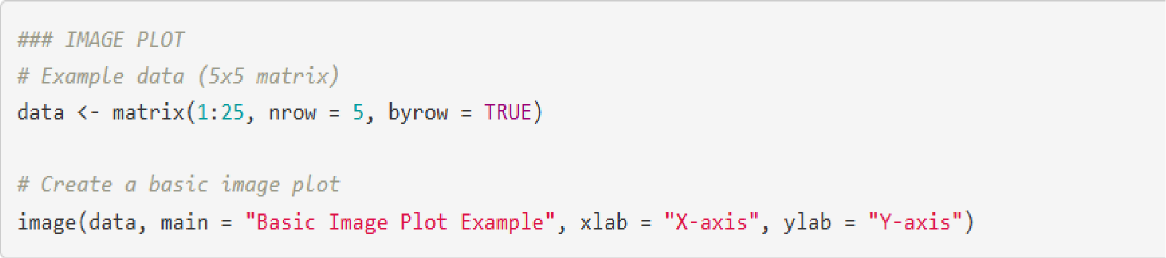
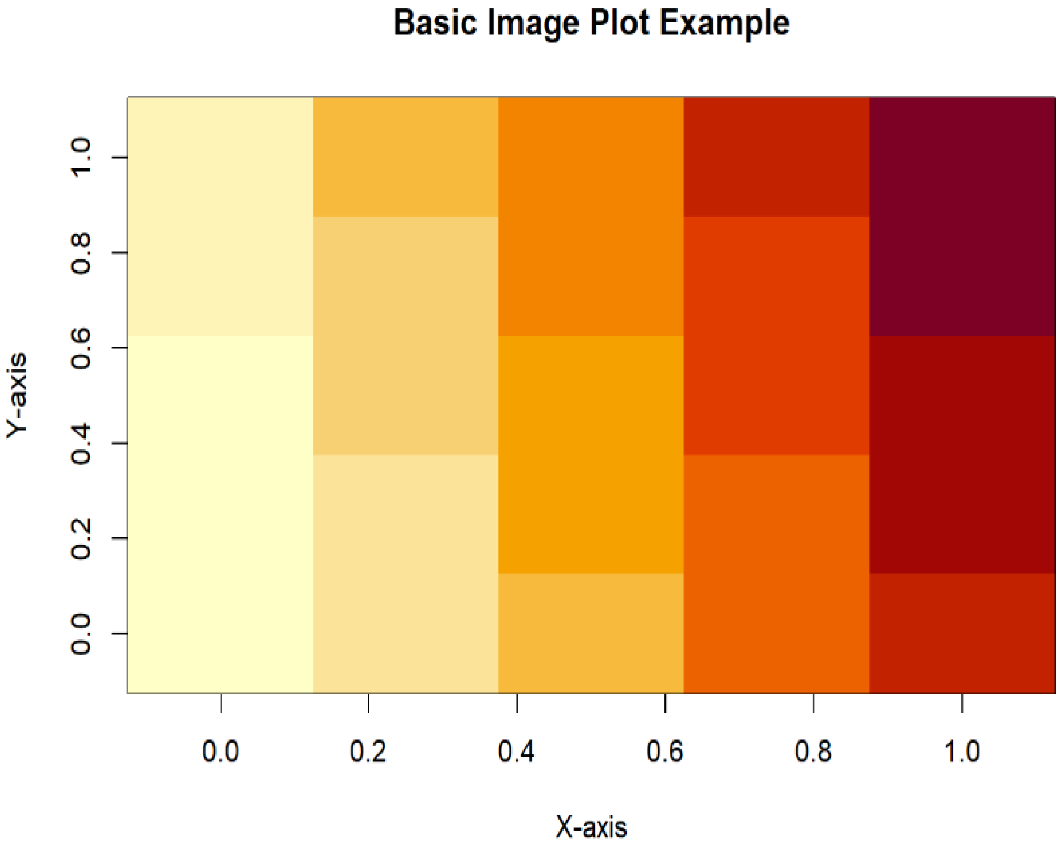
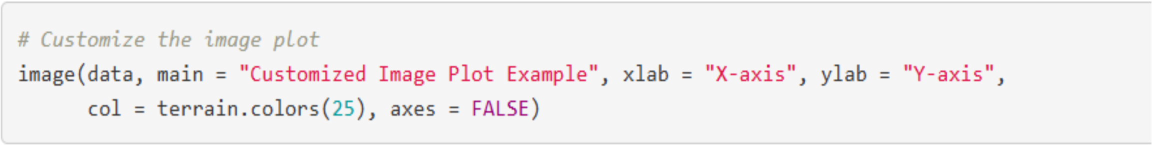


Image Plot

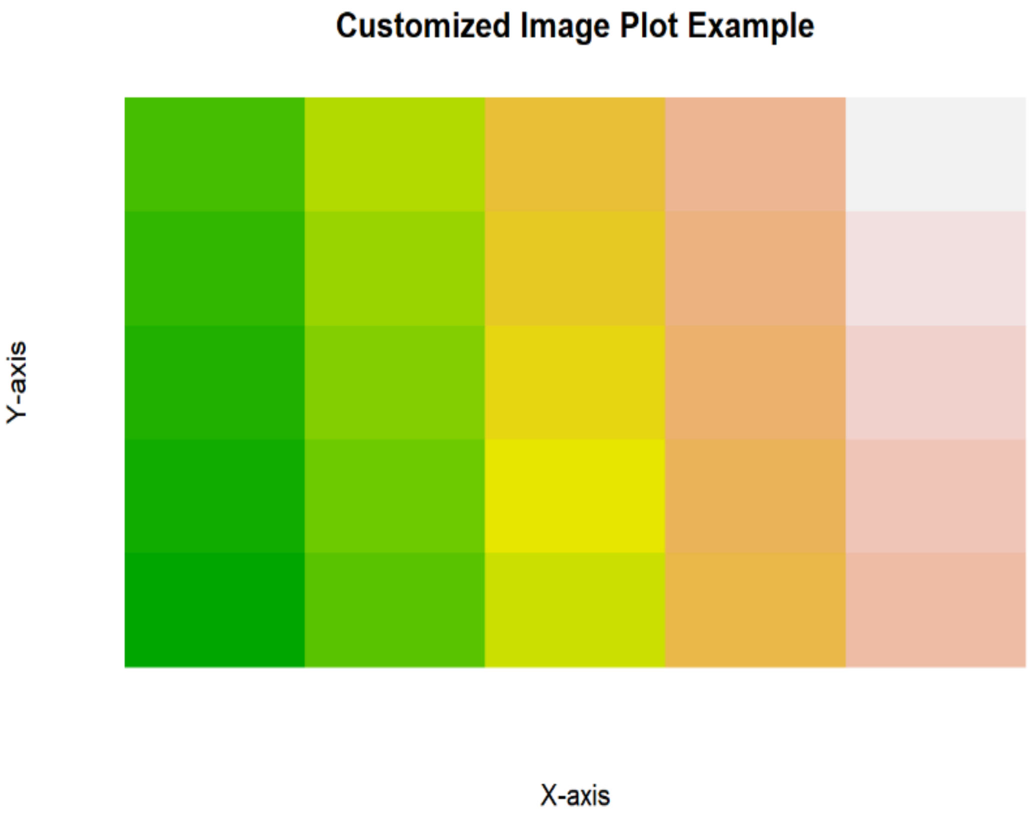


```r
### IMAGE PLOT
# Example data (5x5 matrix)
data <- matrix(1:25, nrow = 5, byrow = TRUE)

# Create a basic image plot
image(data, main = "Basic Image Plot Example", xlab = "X-axis", ylab = "Y-axis")
```

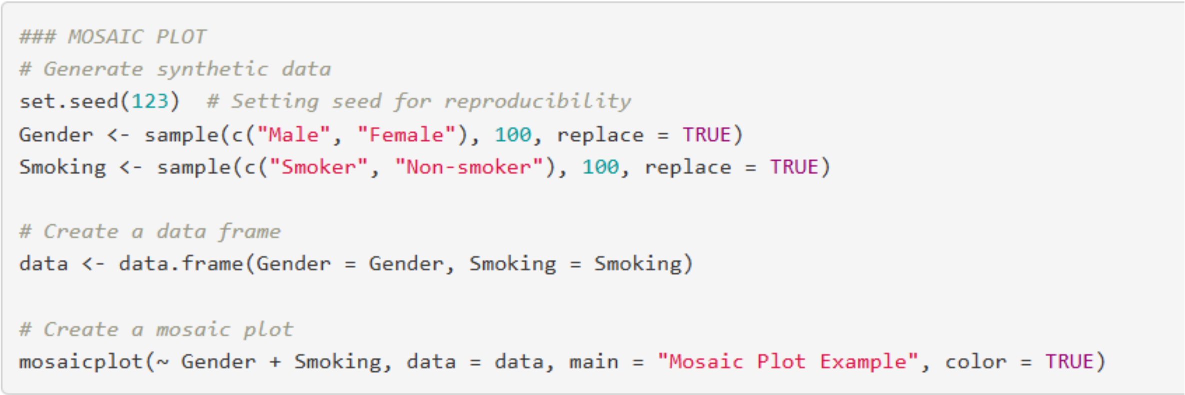




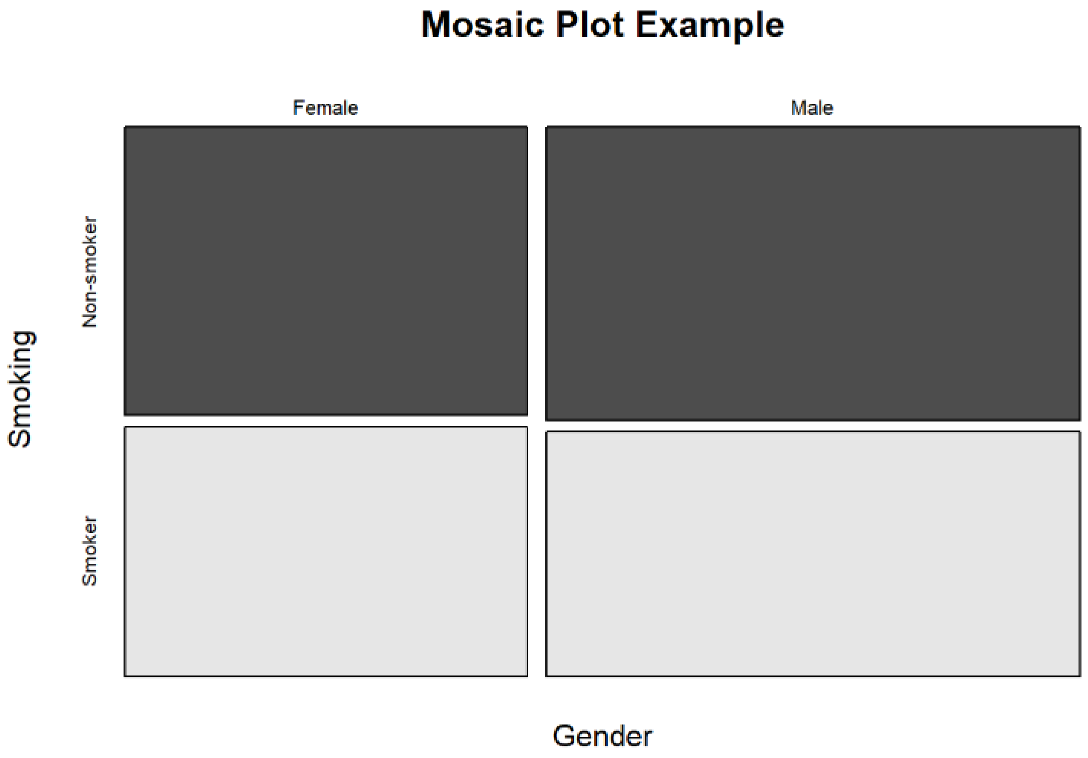
```r
# Customize the image plot
image(data, main = "Customized Image Plot Example", xlab = "X-axis", ylab = "Y-axis",
      col = terrain.colors(25), axes = FALSE)
```



Mosaic Plot



```r
### MOSAIC PLOT
# Generate synthetic data
set.seed(123)  # Setting seed for reproducibility
Gender <- sample(c("Male", "Female"), 100, replace = TRUE)
Smoking <- sample(c("Smoker", "Non-smoker"), 100, replace = TRUE)

# Create a data frame
data <- data.frame(Gender = Gender, Smoking = Smoking)

# Create a mosaic plot
mosaicplot(~ Gender + Smoking, data = data, main = "Mosaic Plot Example", color = TRUE)
```


Mosaic Plot Example
Female
Male
Non-smoker
Smoker
Smoking
Gender