VIDYASAGAR COLLEGE OF ARTS AND SCIENCE, UDUMALPET

DEPARTMENT OF DATA SCIENCE

II B.Sc [DATA SCIENCE]

R Programming

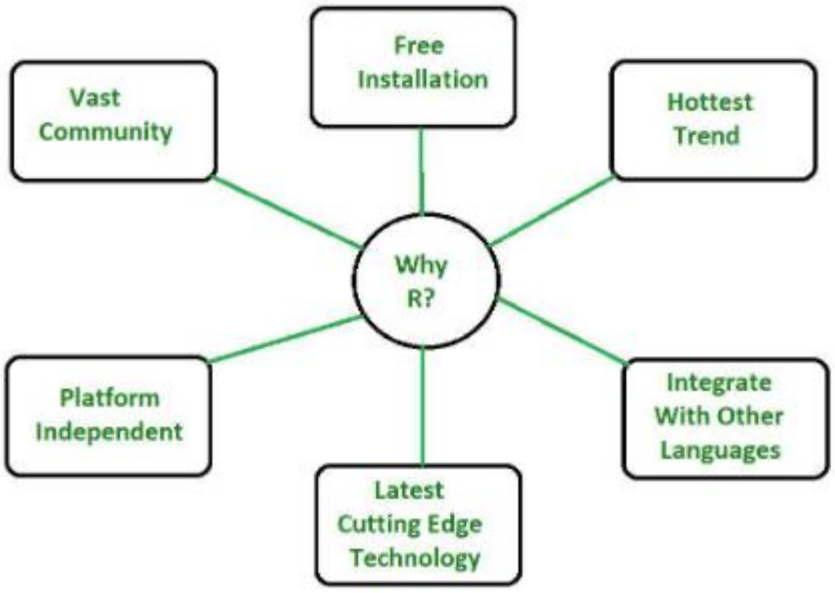
Ms.M.BALAMONICA M.Sc
ASSISTANT PROFESSOR

UNIT I Introducing to R – R Data Structures – Help Functions in R – Vectors – Scalars – Declarations – Recycling – Common Vector Operations – Using all and any – Vectorized operations – Filtering – Victoriesed if-then else – Vector Element names

Introducing to R:

The R Language stands out as a powerful tool in the modern era of statistical computing and data analysis. Widely embraced by statisticians, data scientists, and researchers, the R Language offers an extensive suite of packages and libraries tailored for data manipulation, statistical modeling, and visualization. In this article, we explore the features, benefits, and applications of the R Programming Language, shedding light on why it has become an indispensable asset for data-driven professionals across various industries.

R programming language is an implementation of the S programming language. It also combines with lexical scoping semantics inspired by Scheme. Moreover, the project was conceived in 1992, with an initial version released in 1995 and a stable beta version in 2000.



R Programming Language

What is R Programming Language?

R programming is a leading tool for machine learning, statistics, and data analysis, allowing for the easy creation of objects, functions, and packages. Designed by Ross Ihaka and Robert Gentleman at the University of Auckland and developed by the R Development Core Team, R Language is platform-independent and open-source, making it accessible for use across all operating systems without licensing costs. Beyond its capabilities as a statistical package, R integrates with other languages like C and C++, facilitating interaction with various data sources and statistical tools. With a growing community of users and high demand in the Data Science job market, R is one of the most sought-after programming languages today. Originating as an implementation of the S programming language with influences from Scheme, R has evolved since its conception in 1992, with its first stable beta version released in 2000.

Why Use R Language?

The **R Language** is a powerful tool widely used for data analysis, statistical computing, and machine learning. Here are several reasons why professionals across various fields prefer R:

1. Comprehensive Statistical Analysis:

- R language is specifically designed for statistical analysis and provides a vast array of statistical techniques and tests, making it ideal for data-driven research.

2. Extensive Packages and Libraries:

- The R Language boasts a rich ecosystem of packages and libraries that extend its capabilities, allowing users to perform advanced data manipulation, visualization, and machine learning tasks with ease.

3. Strong Data Visualization Capabilities:

- R language excels in data visualization, offering powerful tools like ggplot2 and plotly, which enable the creation of detailed and aesthetically pleasing graphs and plots.

4. Open Source and Free:

- As an open-source language, R is free to use, which makes it accessible to everyone, from individual researchers to large organizations, without the need for costly licenses.

5. Platform Independence:

- The R Language is platform-independent, meaning it can run on various operating systems, including Windows, macOS, and Linux, providing flexibility in development environments.

6. Integration with Other Languages:

- R can easily integrate with other programming languages such as C, C++, Python, and Java, allowing for seamless interaction with different data sources and statistical packages.

7. Growing Community and Support:

- R language has a large and active community of users and developers who contribute to its continuous improvement and provide extensive support through forums, mailing lists, and online resources.

8. High Demand in Data Science:

- R is one of the most requested programming languages in the Data Science job market, making it a valuable skill for professionals looking to advance their careers in this field.

Features of R Programming Language

The **R Language** is renowned for its extensive features that make it a powerful tool for data analysis, statistical computing, and visualization. Here are some of the key features of R:

1. Comprehensive Statistical Analysis:

- R langauge provides a wide array of statistical techniques, including linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, and clustering.

2. Advanced Data Visualization:

- With packages like ggplot2, plotly, and lattice, R excels at creating complex and aesthetically pleasing data visualizations, including plots, graphs, and charts.

3. Extensive Packages and Libraries:

- The Comprehensive R Archive Network (CRAN) hosts thousands of packages that extend R's capabilities in areas such as machine learning, data manipulation, bioinformatics, and more.

4. Open Source and Free:

- R is free to download and use, making it accessible to everyone. Its open-source nature encourages community contributions and continuous improvement.

5. Platform Independence:
- R is platform-independent, running on various operating systems, including Windows, macOS, and Linux, which ensures flexibility and ease of use across different environments.

6. Integration with Other Languages:
- R language can integrate with other programming languages such as C, C++, Python, Java, and SQL, allowing for seamless interaction with various data sources and computational processes.

7. Powerful Data Handling and Storage:
- R efficiently handles and stores data, supporting various data types and structures, including vectors, matrices, data frames, and lists.

8. Robust Community and Support:
- R has a vibrant and active community that provides extensive support through forums, mailing lists, and online resources, contributing to its rich ecosystem of packages and documentation.

9. Interactive Development Environment (IDE):
- RStudio, the most popular IDE for R, offers a user-friendly interface with features like syntax highlighting, code completion, and integrated tools for plotting, history, and debugging.

10. Reproducible Research:

R supports reproducible research practices with tools like R Markdown and Knitr, enabling users to create dynamic reports, presentations, and documents that combine code, text, and visualizations.

Advantages of R language
- R is the most comprehensive statistical analysis package. As new technology and concepts often appear first in R.
- As R programming language is an open source. Thus, you can run R anywhere and at any time.
- R programming language is suitable for GNU/Linux and Windows operating systems.
- R programming is cross-platform and runs on any operating system.
- In R, everyone is welcome to provide new packages, bug fixes, and code enhancements.

Disadvantages of R language
- In the R programming language, the standard of some packages is less than perfect.
- Although, R commands give little pressure on memory management. So R programming language may consume all available memory.
- In R basically, nobody to complain if something doesn't work.
- R programming language is much slower than other programming languages such as Python and MATLAB.

Applications of R language
- We use R for Data Science. It gives us a broad variety of libraries related to statistics. It also provides the environment for statistical computing and design.
- R is used by many quantitative analysts as its programming tool. Thus, it helps in data importing and cleaning.
- R is the most prevalent language. So many data analysts and research programmers use it. Hence, it is used as a fundamental tool for finance.
- Tech giants like Google, Facebook, Bing, Twitter, Accenture, Wipro, and many more using R nowadays.

Data Structures in R Programming

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Data structures in R programming are tools for holding multiple values.

R's base data structures are often organized by their dimensionality (1D, 2D, or nD) and whether they're homogeneous (all elements must be of the identical type) or heterogeneous (the elements are often of various types). This gives rise to the six data types which are most frequently utilized in data analysis.

The most essential data structures used in R include:

- Vectors
- Lists
- Dataframes
- Matrices
- Arrays
- Factors
- Tibbles

Vectors

A vector is an ordered collection of basic data types of a given length. The only key thing here is all the elements of a vector must be of the identical data type e.g homogeneous data structures. Vectors are one-dimensional data structures.

Example:

R

```
# R program to illustrate Vector

# Vectors(ordered collection of same data type)
X = c(1, 3, 5, 7, 8)



# Printing those elements in console
print(X)
```

Output:

```
[1] 1 3 5 7 8
```

Lists

A list is a generic object consisting of an ordered collection of objects. Lists are heterogeneous data structures. These are also one-dimensional data structures. A list can be a list of vectors, list of matrices, a list of characters and a list of functions and so on.

Example:

R

```
# R program to illustrate a List

# The first attributes is a numeric vector
# containing the employee IDs which is
# created using the 'c' command here
empId = c(1, 2, 3, 4)

# The second attribute is the employee name
# which is created using this line of code here
# which is the character vector
empName = c("Debi", "Sandeep", "Subham", "Shiba")

# The third attribute is the number of employees
# which is a single numeric variable.
numberOfEmp = 4

# We can combine all these three different
# data types into a list
# containing the details of employees
# which can be done using a list command
empList = list(empId, empName, numberOfEmp)

print(empList)
```
Output:
```
[[1]]
[1]                    1                    2                    3                    4

[[2]]
[1]      "Debi"                          "Sandeep"      "Subham"          "Shiba"

[[3]]
[1] 4
```
Dataframes

Dataframes are generic data objects of R which are used to store the tabular data. Dataframes are the foremost popular data objects in R programming because we are comfortable in seeing the data within the tabular form. They are two-dimensional, heterogeneous data structures. These are lists of vectors of equal lengths.

Data frames have the following constraints placed upon them:

- A data-frame must have column names and every row should have a unique name.
- Each column must have the identical number of items.
- Each item in a single column must be of the same data type.
- Different columns may have different data types.

To create a data frame we use the data.frame() function.

Example:

R
```
# R program to illustrate dataframe

# A vector which is a character vector

Name = c("Amiya", "Raj", "Asish")



# A vector which is a character vector
Language = c("R", "Python", "Java")

# A vector which is a numeric vector
Age = c(22, 25, 45)

# To create dataframe use data.frame command
# and then pass each of the vectors
# we have created as arguments
# to the function data.frame()
df = data.frame(Name, Language, Age)

print(df)
```

Output:
```
   Name  Language     Age
1 Amiya      R         22
2 Raj     Python       25
3 Asish    Java        45
```

Matrices

A matrix is a rectangular arrangement of numbers in rows and columns. In a matrix, as we know rows are the ones that run horizontally and columns are the ones that run vertically. Matrices are two-dimensional, homogeneous data structures.
Now, let’s see how to create a matrix in R. To create a matrix in R you need to use the function called matrix. The arguments to this matrix() are the set of elements in the vector. You have to pass how many numbers of rows and how many numbers of columns you want to have in your matrix and this is the important point you have to remember that by default, matrices are in column-wise order.

Example:

R

```
# R program to illustrate a matrix

A = matrix(
   # Taking sequence of elements
   c(1, 2, 3, 4, 5, 6, 7, 8, 9),

   # No of rows and columns
   nrow = 3, ncol = 3,

   # By default matrices are
   # in column-wise order
   # So this parameter decides
   # how to arrange the matrix
   byrow = TRUE
)

print(A)
```

Output:

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

Arrays

Arrays are the R data objects which store the data in more than two dimensions. Arrays are n-dimensional data structures. For example, if we create an array of dimensions (2, 3, 3) then it creates 3 rectangular matrices each with 2 rows and 3 columns. They are homogeneous data structures.

Now, let’s see how to create arrays in R. To create an array in R you need to use the function called array(). The arguments to this array() are the set of elements in vectors and you have to pass a vector containing the dimensions of the array.

Example:

Python3

```
# R program to illustrate an array

A = array(
   # Taking sequence of elements
```

```
    c(1, 2, 3, 4, 5, 6, 7, 8),

    # Creating two rectangular matrices
    # each with two rows and two columns
    dim = c(2, 2, 2)
)

print(A)
```
Output:
```
,                                    ,                                    1

    [,1][,2]
[1,]       1          3
[2,]       2          4

, ,        2

    [,1][,2]
[1 ]  5    7
[2,]  6   8
```

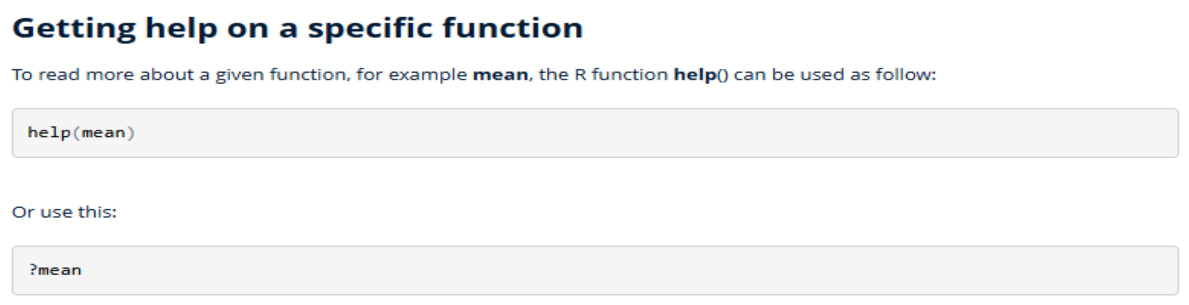
Factors
Factors are the data objects which are used to categorize the data and store it as levels. They are useful for storing categorical data. They can store both strings and integers. They are useful to categorize unique values in columns like “TRUE” or “FALSE”, or “MALE” or “FEMALE”, etc.. They are useful in data analysis for statistical modeling.
Now, let’s see how to create factors in R. To create a factor in R you need to use the function called factor(). The argument to this factor() is the vector.
Example:
R
```
# R program to illustrate factors

# Creating factor using factor()
fac = factor(c("Male", "Female", "Male",
           "Male", "Female", "Male", "Female"))

print(fac)
```
Output:
```
[1]    Male          Female     Male          Male          Female     Male          Female
Levels: Female Male
```
Tibbles
Tibbles are an enhanced version of data frames in R, part of the tidyverse. They offer improved printing, stricter column types, consistent subsetting behavior, and allow variables to be referred to as objects. Tibbles provide a modern, user-friendly approach to tabular data in R.

Now, let’s see how we can create a tibble in R. To create tibbles in R we can use the tibble function from the tibble package, which is part of the tidyverse.
Example:
R
```
# Load the tibble package
library(tibble)
```

```
# Create a tibble with three columns: name, age, and city
my_data <- tibble(
  name = c("Sandeep", "Amit", "Aman"),
  age = c(25, 30, 35),
  city = c("Pune", "Jaipur", "Delhi")
)
# Print the tibble
print(my_data)
```

Output:

```
  Name age          city
  <chr><dbl><chr>
1Sandeep  25         Pune
2Amit      30         Jaipur
3 Aman     35        Delhi
```



Getting help on a specific function

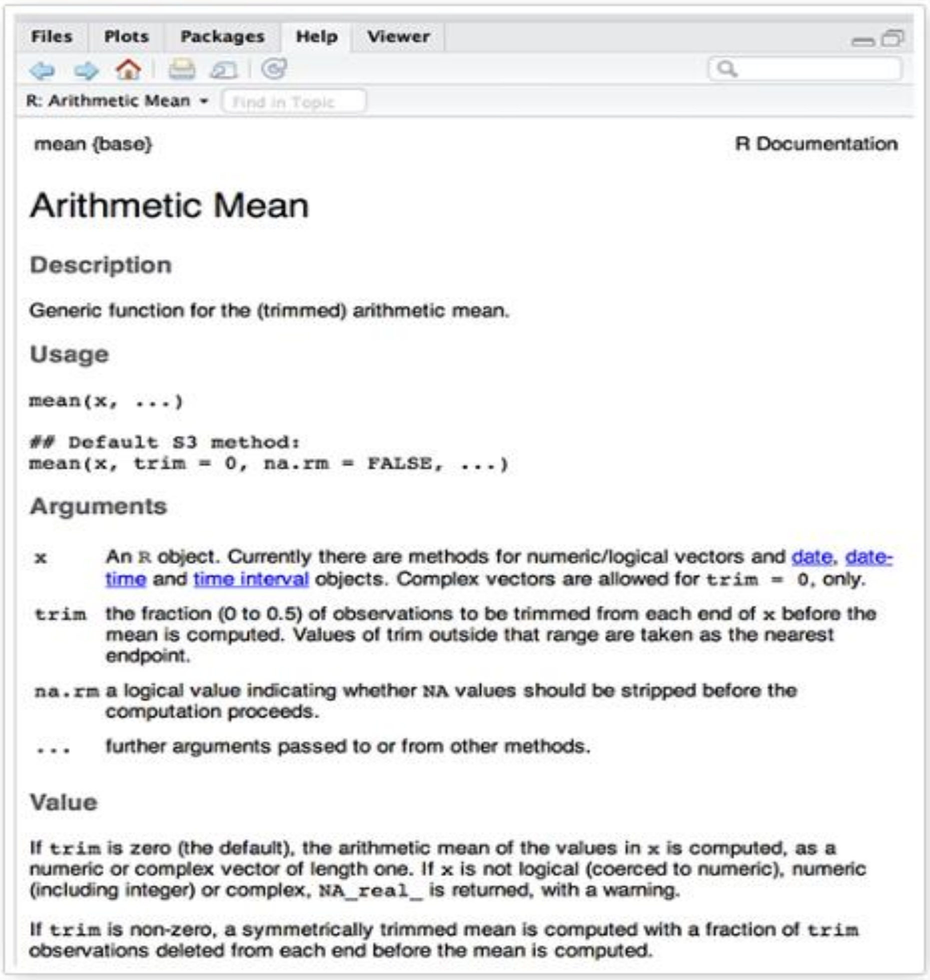
To read more about a given function, for example **mean**, the R function **help()** can be used as follow:

```
help(mean)
```

Or use this:

```
?mean
```

The output look like this:



mean {base}　　R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An `R` object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

If you want to see some examples of how to use the function, type this: **example**(function_name).

```
example(mean)
```

Note that, typical R help files contain the following sections:

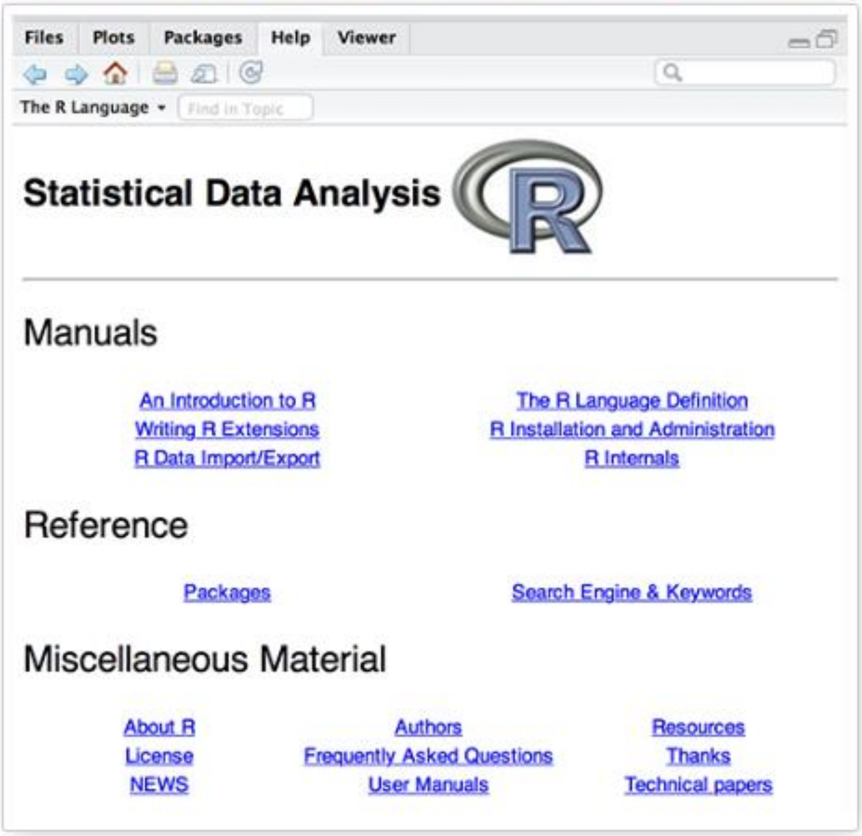
- **Title**
- **Description**: a short description of what the function does.
- **Usage**: the syntax of the function.
- **Arguments**: the description of the arguments taken by the function.
- **Value**: the value returned by the function
- **Examples**: provide examples on how to use the function

General help

If you want to read the general documentation about R, use the function **help.start**():

```
help.start()
```

The output is a web page, on most R installations, which can be browsed by clicking the hyperlinks.



Statistical Data Analysis

Manuals

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

Reference

Packages
Search Engine & Keywords

Miscellaneous Material

About R
Authors
Resources
License
Frequently Asked Questions
Thanks
NEWS
User Manuals
Technical papers

Others

- **apropos**(): returns a list of object, containing the pattern you searched, by partial matching. This is useful when you don't remember exactly the name of the function:

```
# Returns the list of object containing "med"
apropos("med")
```

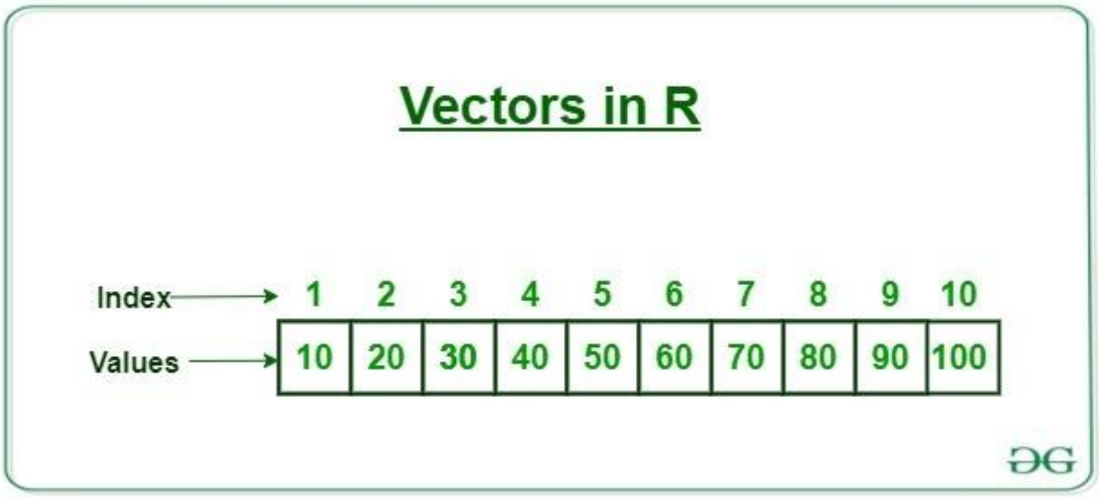
```
[1] ".__C__namedList" "elNamed"         "elNamed<-"       "median"          "median.default"
[6] "medpolish"       "runmed"
```

- **healp.search**() (alternatively **??**): Search for documentation matching a given character in different ways. It returns a list of function containing your searched term with a short description of the function.

```
help.search("mean")
# Or use this
??mean
```

Vectors

R Vectors are the same as the arrays in R language which are used to hold multiple data values of the same type. One major key point is that in R Programming Language the indexing of the vector will start from '1' and not from '0'. We can create numeric vectors and character vectors as well.



R – Vector

Creating a vector

A vector is a basic data structure that represents a one-dimensional array. to create a array we use the "c" function which the most common method use in R Programming Language.

- R

```
# R program to create Vectors



# we can use the c function

# to combine the values as a vector.

# By default the type will be double

X<- c(61, 4, 21, 67, 89, 2)

cat('using c function', X, '\n')



# seq() function for creating

# a sequence of continuous values.
# length.out defines the length of vector.

Y<- seq(1, 10, length.out = 5)

cat('using seq() function', Y, '\n')

# use':' to create a vector

# of continuous values.

Z<- 2:7

cat('using colon', Z)
```

Output:

```
using c function 61 4 21 67 89 2
using seq() function 1 3.25 5.5 7.75 10
using colon 2 3 4 5 6 7
```

Types of R vectors

Vectors are of different types which are used in R. Following are some of the types of vectors:

Numeric vectors

Numeric vectors are those which contain numeric values such as integer, float, etc.

- R

```
# R program to create numeric Vectors

# creation of vectors using c() function.

v1<- c(4, 5, 6, 7)

# display type of vector

typeof(v1)

# by using 'L' we can specify that we want integer values.

v2<- c(1L, 4L, 2L, 5L)

# display type of vector
```

```
typeof(v2)
```

Output:

```
[1] "double"
[1] "integer"
```

Character vectors

Character vectors in R contain alphanumeric values and special characters.

- R

```
# R program to create Character Vectors

# by default numeric values

# are converted into characters

v1<- c('geeks', '2', 'hello', 57)

# Displaying type of vector

typeof(v1)
```

Output:

```
[1] "character"
```

Logical vectors

Logical vectors in R contain Boolean values such as TRUE, FALSE and NA for Null values.

- R

```
# R program to create Logical Vectors

# Creating logical vector

# using c() function

v1<- c(TRUE, FALSE, TRUE, NA)
```

```
# Displaying type of vector

typeof(v1)
```

Output:

```
[1] "logical"
```

Length of R vector

In R, the length of a vector is determined by the number of elements it contains. we can use the length() function to retrieve the length of a vector.

```
# Create a numeric vector

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

# Find the length of the vector

length(x)

# Create a character vector

y <- c("apple", "banana", "cherry")

# Find the length of the vector

length(y)

# Create a logical vector

z <- c(TRUE, FALSE, TRUE, TRUE)

# Find the length of the vector

length(z)
```

Output:

```
> length(x)
[1] 5

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

```
> length(z)
[1] 4
```

Accessing R vector elements
Accessing elements in a vector is the process of performing operation on an individual element of a vector. There are many ways through which we can access the elements of the vector. The most common is using the '[]', symbol.
Note: *Vectors in R are 1 based indexing unlike the normal C, python, etc format.*

```
# R program to access elements of a Vector



# accessing elements with an index number.

X<- c(2, 5, 18, 1, 12)

cat('Using Subscript operator', X[2], '\n')



# by passing a range of values

# inside the vector index.

Y<- c(4, 8, 2, 1, 17)

cat('Using combine() function', Y[c(4, 1)], '\n')
```

Output:

```
Using Subscript operator 5
Using combine() function 1 4
```

Modifying a R vector
Modification of a Vector is the process of applying some operation on an individual element of a vector to change its value in the vector. There are different ways through which we can modify a vector:

- R

Ms.M.BALAMONICA M.Sc
ASSISTANT PROFESSOR

```
# R program to modify elements of a Vector

# Creating a vector

X<- c(2, 7, 9, 7, 8, 2)

# modify a specific element

X[3] <- 1

X[2] <-9

cat('subscript operator', X, '\n')

# Modify using different logics.

X[1:5]<- 0

cat('Logical indexing', X, '\n')

# Modify by specifying

# the position or elements.

X<- X[c(3, 2, 1)]

cat('combine() function', X)
```

Output:

```
subscript operator 2 9 1 7 8 2
Logical indexing 0 0 0 0 0 2
combine() function 0 0 0
```

Deleting a R vector

Deletion of a Vector is the process of deleting all of the elements of the vector. This can be done by assigning it to a NULL value.

Output:

```
Output vector NULL
```

```
# R program to delete a Vector

# Creating a Vector

M<- c(8, 10, 2, 5)

# set NULL to the vector

M<- NULL

cat('Output vector', M)
```

Sorting elements of a R Vector
sort() function is used with the help of which we can sort the values in ascending or descending order.

- R

```
# R program to sort elements of a Vector

# Creation of Vector

X<- c(8, 2, 7, 1, 11, 2)

# Sort in ascending order

A<- sort(X)

cat('ascending order', A, '\n')

# sort in descending order

# by setting decreasing as TRUE

B<- sort(X, decreasing = TRUE)

cat('descending order', B)
```

Output:

```
ascending order 1  2  2  7  8 11
descending order 11  8  7  2  2  1
```

Scalars

The simplest object type in R is a **scalar**. A scalar object is just a single value like a number or a name. In the previous chapter we defined several scalar objects. Here are examples of numeric scalars:

```
# Examples of numeric scalars
a <- 100
b <- 3 / 100
c <- (a + b) / b
```

Scalars don't have to be numeric, they can also be **characters** (also known as strings). In R, you denote characters using quotation marks. Here are examples of character scalars:

```
# Examples of character scalars
d <- "ship"

e <- "cannon"
f <- "Do any modern armies still use cannons?"
```

As you can imagine, R treats numeric and character scalars differently. For example, while you can do basic arithmetic operations on numeric scalars – they won't work on character scalars. If you try to perform numeric operations (like addition) on character scalars, you'll get an error like this one:

```
a <- "1"
b <- "2"
a + b
Error in a + b: non-numeric argument to binary operator
```

If you see an error like this one, it means that you're trying to apply numeric operations to character objects. That's just sick and wrong.

##Vectors

Now let's move onto vectors. A vector object is just a combination of several scalars stored as a single object. For example, the numbers from one to ten could be a vector of length 10, and the characters in the English alphabet could be a vector of length 26. Like scalars, vectors can be either numeric or character (but not both!).
There are many ways to create vectors in R. Here are the methods we will cover in this chapter:

Functions to create vectors.

Function	Example	Result
c(a, b, ...)	c(1, 5, 9)	1, 5, 9
a:b	1:5	1, 2, 3, 4, 5
seq(from, to, by, length.out)	seq(from = 0, to = 6, by = 2)	0, 2, 4, 6
rep(x, times, each, length.out)	rep(c(7, 8), times = 2, each = 2)	7, 7, 8, 8, 7, 7, 8, 8

The simplest way to create a vector is with the c() function. The c here stands for concatenate, which means "bring them together". The c() function takes several scalars as arguments, and returns a vector containing those objects. When using c(), place a comma in between the objects (scalars or vectors) you want to combine:
Let's use the c() function to create a vector called a containing the integers from 1 to 5.

```
# Create an object a with the integers from 1 to 5
a <- c(1, 2, 3, 4, 5)

# Print the result
a


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

As you can see, R has stored all 5 numbers in the object a. Thanks R!
You can also create longer vectors by combining vectors you have already defined. Let's create a vector of the numbers from 1 to 10 by first generating a vector a from 1 to 5, and a vector b from 6 to 10 then combine them into a single vector x:

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

You can also create character vectors by using the c() function to combine character scalars into character vectors:

Figure This is not a pipe. It is a character vector.

```
char.vec <- c("Ceci", "nest", "pas", "une", "pipe")
char.vec
## [1] "Ceci" "nest" "pas"  "une"  "pipe"
```

While the c() function is the most straightforward way to create a vector, it's also one of the most tedious. For example, let's say you wanted to create a vector of all integers from 1 to 100. You definitely don't want to have to type all the numbers into a c() operator. Thankfully, R has many simple built-in functions for generating numeric vectors. Let's start with three of them: a:b, seq(), and rep():

a:b

The a:b function takes two numeric scalars a and b as arguments, and returns a vector of numbers from the starting point a to the ending point b in steps of 1.
Here are some examples of the a:b function in action. As you'll see, you can go backwards or forwards, or make sequences between non-integers:

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
10:1
##  [1] 10  9  8  7  6  5  4  3  2  1
2.5:8.5


## [1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5
```

###seq()

The seq() function is a more flexible version of a:b. Like a:b, seq() allows you to create a sequence from a starting number to an ending number. However, seq() has additional arguments that allow you to specify either the size of the steps between numbers, or the total length of the sequence.

The seq() function has two new arguments: by and length.out. If you use the by argument, the

Argument	Definition
from	The start of the sequence
to	The end of the sequence
by	The step-size of the sequence
length.out	The desired length of the final sequence (only use if you don’t specify by)

sequence will be in steps of the input to the by argument:

```
# Create the numbers from 1 to 10 in steps of 1
seq(from = 1, to = 10, by = 1)
##  [1]  1  2  3  4  5  6  7  8  9 10

# Integers from 0 to 100 in steps of 10
seq(from = 0, to = 100, by = 10)
##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

If you use the length.out argument, the sequence will have a length equal to length.out.

```
# Create 10 numbers from 1 to 5
seq(from = 1, to = 5, length.out = 10)
##  [1] 1.0 1.4 1.9 2.3 2.8 3.2 3.7 4.1 4.6 5.0

# 3 numbers from 0 to 100
seq(from = 0, to = 100, length.out = 3)


## [1]   0  50 100
```

###rep()

Argument	Definition
x	A scalar or vector of values to repeat
times	The number of times to repeat x
each	The number of times to repeat each value within x
length.out	The desired length of the final sequence

The rep() function allows you to repeat a scalar (or vector) a specified number of times, or to a desired length. Let’s do some reps.

```
rep(x = 3, times = 10)
##  [1] 3 3 3 3 3 3 3 3 3 3
rep(x = c(1, 2), each = 3)
## [1] 1 1 1 2 2 2
rep(x = 1:3, length.out = 10)
```

```
## [1] 1 2 3 1 2 3 1 2 3 1
```
As you can see, you can can include an a:b call within a rep()!
You can even combine the times and each arguments within a single rep() function. For example, here's how to create the sequence {1, 1, 2, 2, 3, 3, 1, 1, 2, 2, 3, 3} with one call to rep():
```
rep(x = 1:3, each = 2, times = 2)
## [1] 1 1 2 2 3 3 1 1 2 2 3 3
```
Warning! Vectors contain either numbers or characters, not both

A vector can only contain one type of scalar: either numeric or character. If you try to create a vector with numeric and character scalars, then R will convert *all* of the numeric scalars to characters. In the next code chunk, I'll create a new vector called my.vec that contains a mixture of numeric and character scalars.
```
my.vec <- c("a", 1, "b", 2, "c", 3)
my.vec
## [1] "a" "1" "b" "2" "c" "3"
```

As you can see from the output, my.vec is stored as a character vector where all the numbers are converted to characters.

Declarations

In R, declarations typically refer to the process of creating or assigning values to variables, functions, or objects. However, R doesn't require explicit "declarations" as in some other languages (e.g., Java or C++). You can directly assign a value to a variable, and R will infer its type. That said, there are some important concepts in R related to how variables and functions are "declared" and used.

1. Variable Declaration and Assignment

In R, variables are created when you assign a value to them. There's no need to declare the type of a variable, as R is dynamically typed. You can use either the <- or = assignment operators.

r

Copy code
```
# Using <- (preferred)
x <- 10

# Using =
y = 20
```

In this case, x and y are variables, and R automatically assigns their type based on the value assigned to them (e.g., numeric, character, etc.).

2. Vectors (Creating Lists)

You can declare and assign vectors using the c() function, which concatenates elements into a vector.

r
Copy code

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

3. Functions (Creating Functions)

Functions in R are defined using the function() keyword. You can "declare" a function like this:

r
Copy code
```
my_function <- function(a, b) {
  return(a + b)
}
```

In this example, my_function is declared, and it takes two arguments a and b, returning their sum.

4. Data Frames (Creating Data Frames)

You can create a data frame by using the data.frame() function. This is a common structure for storing data in R.

r

Copy code
```
my_data <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35)
)
```

5. Constants (Global Constants)

While R doesn’t have constants in the strict sense (like const in C++), you can create a variable that acts like a constant by simply not modifying its value after it’s set.

r
Copy code
```
PI <- 3.14159  # Using the naming convention for constants
```

6. Lists (Creating Lists)

Lists in R are used to store heterogeneous elements. You declare a list like this:

r

Copy code
```
my_list <- list(name="John", age=30, height=5.9)
```

7. Packages

To use external libraries or packages in R, you need to declare them using the library() or require() function after installing them.

r
Copy code
```
install.packages("ggplot2")  # Install the package (if not already installed)
library(ggplot2)  # Declare the package for use
```

8. Global and Local Variables

R has global and local scopes. Variables that are created inside a function are local to that function. Variables that are created outside any function are global.

```
r
Copy code
x <- 5  # Global variable

my_function <- function() {
  y <- 10  # Local variable
  return(x + y)
}
```

9. Global Assignment

You can assign values globally within a function using the <<- operator. This changes the value of a variable outside the function.

```
r
Copy code
my_function <- function() {
  x <<- 100  # Modify global x
}

my_function()
print(x)  # Prints 100
```

Summary of Common R Declarations:

- **Variable**: x <- 10
- **Function**: my_func <- function() { }
- **Vector**: my_vector <- c(1, 2, 3)
- **Data Frame**: my_df <- data.frame(a = 1:5, b = letters[1:5])
- **List**: my_list <- list(name = "Alice", age = 25)
- **Package**: library(ggplot2)

RECYCLING

In R, **recycling** refers to a behavior that occurs when vectors of different lengths are combined or operated on element-wise. R automatically **recycles** the shorter vector to match the length of the longer vector in these situations. This allows operations like addition, subtraction, and other element-wise operations between vectors of different lengths without explicitly needing to make the vectors the same length.

However, recycling happens under certain rules, and it’s important to understand the behavior to avoid unintended results.

How Recycling Works

When performing operations between two vectors of different lengths, R will recycle the elements of the shorter vector until it matches the length of the longer vector. The shorter vector will be reused from the beginning once it reaches the end.

For example:

r
Copy code
```
# Vectors of different lengths
x <- c(1, 2, 3)
y <- c(10, 20)

# Adding vectors
z <- x + y
print(z)
```

Output:

r
Copy code
```
[1] 11 22 13
```

- In this case, x has 3 elements and y has 2 elements.
- R "recycles" y so that it becomes c(10, 20, 10), and the operation proceeds element-wise:
    - 1 + 10 = 11
    - 2 + 20 = 22
    - 3 + 10 = 13

Rules of Recycling

- **Recycling only happens when the length of the shorter vector divides the length of the longer vector**. If it doesn't, R will give a warning.
- **Length mismatch warning**: If the length of the longer vector isn't a multiple of the shorter vector's length, R will give a warning to indicate this behavior.

Example with Warning:
r
Copy code
```
x <- c(1, 2, 3, 4)
y <- c(10, 20)

z <- x + y  # This will give a warning
```

Output:

r
Copy code
```
[1] 11 22 13 24
Warning message:
In x + y : longer object length is not a multiple of shorter object length
```

Here, R recycles y to c(10, 20, 10, 20) to match the length of x, but since 4 (length of x) is not a multiple of 2 (length of y), it gives a warning.

Practical Example of Recycling

Recycling is frequently used in operations like plotting, element-wise transformations, and arithmetic operations. Here's an example with a more practical use case.

```r
Copy code
# Create a vector of numbers from 1 to 4
data <- c(1, 2, 3, 4)

# Create a vector of colors (2 colors)
colors <- c("red", "blue")

# Assign colors to data points in a plot (recycling happens here)
plot(data, col=colors)
```

In this case, R recycles the colors vector so that it repeats the sequence "red", "blue" for each data point in data.

Key Points:

- **Recycling works when the length of the longer vector is a multiple of the shorter vector's length**.
- **Be cautious** about potential bugs when the vectors are of mismatched lengths that don't fit the recycling rule.
- **It's good practice to manually check lengths** or ensure that you aren't inadvertently relying on unintended recycling, especially when performing complex operations.

Controlling Recycling

If you don't want recycling to occur or if you want to avoid warnings, you can always explicitly make the vectors the same length, for example, by using rep() (repeat) or length() functions:

```r
Copy code
# Make sure both vectors are the same length
x <- c(1, 2, 3, 4)



y <- rep(c(10, 20), length.out = length(x))  # Recycle y manually

z <- x + y  # Now no warning
print(z)
```

Output:

```r
Copy code
[1] 11 22 13 24
```

In this example, rep(c(10, 20), length.out = length(x)) ensures that y is the same length as x, and no warning is issued.

Common Vector Operations

Vectors are the most basic data types in R. Even a single object created is also stored in the form of a vector. Vectors are nothing but arrays as defined in other languages. Vectors contain a sequence of homogeneous types of data. If mixed values are given then it auto converts the data according to the precedence. There are various operations that can be performed on vectors in R.

Creating a vector

Vectors can be created in many ways as shown in the following example. The most usual is the use of ‘c’ function to combine different elements together.

```
# Use of 'c' function

# to combine the values as a vector.

# by default the type will be double

X <- c(1, 4, 5, 2, 6, 7)

print('using c function')

print(X)



# using the seq() function to generate

# a sequence of continuous values

# with different step-size and length.



# length.out defines the length of vector.

Y <- seq(1, 10, length.out = 5)
```

```
print('using seq() function')

print(Y)



# using ':' operator to create

# a vector of continuous values.

Z <- 5:10

print('using colon')

print(Y)
```

Output:

```
using c function 1 4 5 2 6 7

using seq function 1.00  3.25  5.50  7.75 10.00

using colon 5  6  7  8  9 10
```

Accessing vector elements
Vector elements can be accessed in many ways. The most basic is using the '[]', subscript operator. Following are the ways of accessing Vector elements:

Note: *vectors in R are 1 based indexed, unlike the normal C, python, etc format where indexing starts from 0*
Python3

```
# Accessing elements using the position number.

X <- c(2, 5, 8, 1, 2)

print('using Subscript operator')

print(X[2])

# Accessing specific values by passing
# a vector inside another vector.

Y <- c(4, 5, 2, 1, 7)

print('using c function')

print(Y[c(4, 1)])

# Logical indexing

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

print('Logical indexing')

print(Z[Z>3])
```

Output:

```
using Subscript operator 5

using c function 1 4

Logical indexing 5 4 4
```

Modifying a vector
Vectors can be modified using different indexing variations which are mentioned in the below code:

```
# Creating a vector

X <- c(2, 5, 1, 7, 8, 2)



# modify a specific element

X[3] <- 11

print('Using subscript operator')
print(X)

# Modify using different logics.

X[X>9] <- 0

print('Logical indexing')

print(X)

# Modify by specifying the position or elements.

X <- X[c(5, 2, 1)]

print('using c function')

print(X)
```

Output:

```
Using subscript operator 2  5 11  7  8  2
Logical indexing 2 5 0 7 8 2
using c function 8 5 2
```

Deleting a vector
Vectors can be deleted by reassigning them as NULL. To delete a vector we use the NULL operator.

- Python3

```
# Creating a vector

X <- c(5, 2, 1, 6)

# Deleting a vector

X <- NULL
```

```
print('Deleted vector')

print(X)
```

```
Deleted vector NULL
```

Arithmetic operations
We can perform arithmetic operations between 2 vectors. These operations are performed element-wise and hence the length of both the vectors should be the same.

```
# Creating Vectors

X <- c(5, 2, 5, 1, 51, 2)

Y <- c(7, 9, 1, 5, 2, 1)

# Addition

Z <- X + Y

print('Addition')

print(Z)

# Subtraction

S <- X - Y

print('Subtraction')

print(S)

# Multiplication

M <- X * Y

print('Multiplication')

print(M)
```

```
# Division

D <- X / Y

print('Division')

print(D)
```

Output:

```
Addition 12 11  6  6 53  3

Subtraction -2 -7  4 -4 49  1

Multiplication 35  18   5   5 102   2

Division 0.7142857  0.2222222  5.0000000  0.2000000 25.5000000  2.0000000
```

Sorting of Vectors
For sorting we use the **sort()** function which sorts the vector in ascending order by default.

```
# Creating a Vector

X <- c(5, 2, 5, 1, 51, 2)



# Sort in ascending order

A <- sort(X)

print('sorting done in ascending order')

print(A)

# sort in descending order.
```

```
B <- sort(X, decreasing = TRUE)

print('sorting done in descending order')

print(B)
```

Output:

```
sorting done in ascending order 1  2  2  5  5 51
sorting done in descending order 51  5  5  2  2  1
```

Using all and any

In R, the functions all() and any() are used to evaluate logical conditions over vectors, matrices, or other data structures. These functions are very useful for checking whether all or any elements in a logical object satisfy a given condition.

1. all() Function

The all() function returns TRUE if **all** the elements in a logical vector (or the result of a logical operation) are TRUE. If **any element** is FALSE, it returns FALSE.

Syntax:
r
Copy code
all(x, na.rm = FALSE)

- x: A logical vector (or an object that can be coerced to logical).
- na.rm: If TRUE, NA values are ignored. If FALSE (the default), NA will cause the result to be NA.

Example:
r
Copy code

```
# Logical vector
x <- c(TRUE, TRUE, TRUE)

# Check if all values are TRUE

all(x)
```

# Output: TRUE

# Another example with one FALSE
x <- c(TRUE, FALSE, TRUE)
all(x)
# Output: FALSE

# With NA (using na.rm = TRUE)
x <- c(TRUE, NA, TRUE)
all(x, na.rm = TRUE)
# Output: TRUE (ignores NA)

2. any() Function

The any() function returns TRUE if **any** of the elements in the vector are TRUE. It returns FALSE only if **all** elements are FALSE. If the vector contains NA, it will return NA unless you specify na.rm = TRUE.

Syntax:

r
Copy code
any(x, na.rm = FALSE)

- x: A logical vector (or an object that can be coerced to logical).
- na.rm: If TRUE, NA values are ignored. If FALSE (the default), NA will cause the result to be NA.

Example:

r
Copy code
# Logical vector
x <- c(TRUE, FALSE, FALSE)

# Check if any value is TRUE
any(x)
# Output: TRUE

# All values are FALSE
x <- c(FALSE, FALSE, FALSE)
any(x)
# Output: FALSE

# With NA (using na.rm = TRUE)
x <- c(FALSE, NA, FALSE)
any(x, na.rm = TRUE)
# Output: FALSE (ignores NA)

3. Use Cases and Examples

Here are a few practical examples of how all() and any() might be used:

Example 1: Checking conditions in a vector

You can use all() and any() to check conditions over vectors or arrays.

```
r

Copy code
# Check if all values are positive
numbers <- c(2, 3, 4, 5)
all(numbers > 0)  # Check if all numbers are greater than 0
# Output: TRUE

# Check if any value is negative
numbers <- c(2, -3, 4, 5)
any(numbers < 0)  # Check if any number is less than 0
# Output: TRUE
```

Example 2: Filtering or Subsetting Data

You might use these functions to filter or subset data based on certain conditions.

```
r
Copy code
# Given a data frame of students' scores
scores <- data.frame(name = c("Alice", "Bob", "Charlie"),
                     score = c(85, 45, 90))

# Check if all students passed (let's say passing score is 50)
all(scores$score >= 50)  # Returns TRUE if all students passed
# Output: FALSE (since Bob's score is 45)

# Check if any student failed
any(scores$score < 50)  # Returns TRUE if any student failed
# Output: TRUE (since Bob's score is less than 50)
```

Example 3: Handling NA Values

In real-world data, NA values are common. By using na.rm = TRUE, you can ignore missing values in your checks.

```
r
Copy code
# Vector with NA values
x <- c(TRUE, FALSE, NA, TRUE)

# Check if all values are TRUE (ignoring NA values)
all(x, na.rm = TRUE)  # Output: FALSE

# Check if any value is TRUE (ignoring NA values)
any(x, na.rm = TRUE)  # Output: TRUE
```

4. Combination of all() and any() with Logical Expressions

You can also combine all() and any() with logical expressions or conditions directly.

Example 1: Combining all() with logical operators

r
Copy code

```
# Checking if all values in a vector are greater than 0
x <- c(2, 3, 4, 5)

all(x > 0)  # TRUE, because all values are positive

# Checking if all values are even
x <- c(2, 4, 6, 8)


all(x %% 2 == 0)  # TRUE, because all numbers are even
```

Example 2: Using any() with logical operators

r
Copy code

```
# Checking if any value in a vector is less than 0
x <- c(1, -3, 5, 6)


any(x < 0)  # TRUE, because there is a negative number (-3)

# Checking if any value in a vector is NA
x <- c(1, 2, NA, 4)
any(is.na(x))  # TRUE, because there is an NA value
```

5. Summary

- **all(x)**: Returns TRUE if **all elements** of x are TRUE, otherwise FALSE.
- **any(x)**: Returns TRUE if **any element** of x is TRUE, otherwise FALSE.
- Both functions are useful for **evaluating logical conditions** over vectors, matrices, or arrays.
- You can handle NA values using the na.rm = TRUE argument to ignore NA values in the evaluation.

Vectorized operations – Filtering – Victoriesed if-then else – Vector Element names

Vectorized Operations in R

One of the core features of R is its ability to perform **vectorized operations**, which allows you to apply operations to entire vectors (or other data structures like matrices or data frames) without the need for explicit loops. This makes the code more concise and often much faster than traditional looping constructs.

Let's break down some key concepts: **vectorized operations**, **filtering**, **if-then-else (vectorized conditional logic)**, and **vector element names**.

1. Vectorized Operations

In R, many basic operations are **vectorized**. This means that R automatically applies the operation element-wise to the vectors, matrices, or other data structures.

Example of Vectorized Operations:

```
r
Copy code
# Vector addition (vectorized)
x <- c(1, 2, 3)
y <- c(4, 5, 6)
z <- x + y
print(z)
```

Output:

```
r

Copy code

[1] 5 7 9
```

Here, x + y automatically adds corresponding elements of x and y, which is much more efficient than using a loop.

Other Vectorized Operations:

```
r
Copy code
# Element-wise multiplication
z <- x * y
print(z)

# Element-wise division
z <- x / y
print(z)

# Element-wise comparison
z <- x > y  # Checks if elements in x are greater than those in y
print(z)
```

---

2. Filtering Vectors

Filtering refers to **selecting elements** from a vector that meet a certain condition. In R, this can be done using **logical indexing**. Logical indexing is vectorized, meaning you can apply a condition across the entire vector to filter it.

Example: Filtering Values Greater than a Threshold

```
r
Copy code
x <- c(1, 2, 3, 4, 5, 6)
filtered_x <- x[x > 3]
print(filtered_x)
```

Output:

r
Copy code
```
[1] 4 5 6
```

- x > 3 generates a logical vector c(FALSE, FALSE, FALSE, TRUE, TRUE, TRUE), and then x[x > 3] selects the corresponding elements from x where the condition is TRUE.

Example: Filtering with Multiple Conditions

You can combine multiple conditions using logical operators (& for AND, | for OR).

r
Copy code
```
# Filter values greater than 2 and less than 5
filtered_x <- x[x > 2 & x < 5]
print(filtered_x)
```
Output:
r
Copy code
```
[1] 3 4
```

---

3. Vectorized If-Then-Else (Conditional Logic)

In R, you can use the ifelse() function to apply a vectorized version of an "if-then-else" logic. This allows you to assign values based on a condition in a vectorized manner without using for loops.

Syntax:
r
Copy code
```
ifelse(test, yes, no)
```

- test: The condition to check (a logical vector).
- yes: The value to return if the condition is TRUE.
- no: The value to return if the condition is FALSE.

Example: Applying ifelse() to a Vector
r
Copy code
```
x <- c(1, 2, 3, 4, 5)

# Replace values less than 3 with 0, otherwise keep the original value
result <- ifelse(x < 3, 0, x)
print(result)
```

Output:

r
Copy code

```
[1] 0 0 3 4 5
```

In this case, values of x that are less than 3 are replaced with 0, and the rest of the values remain unchanged.

Example: More Complex Conditions

You can also use more complex conditions in ifelse().

r
Copy code
```
x <- c(1, 2, 3, 4, 5)

# Use different values for different conditions
result <- ifelse(x < 2, "Low", ifelse(x < 4, "Medium", "High"))
print(result)
```

Output:

r
Copy code
```
[1] "Low"    "Low"    "Medium" "Medium" "High"
```

In this example, we apply nested ifelse() to assign different categories ("Low", "Medium", "High") based on the values in x.

4. Vector Element Names

In R, vectors can have **named elements**. This allows you to access or modify elements by name rather than by position. You can assign names to elements of a vector using the names() function.

Assigning Names to a Vector:
r
Copy code
```
# Create a vector
x <- c(10, 20, 30)

# Assign names to the elements
names(x) <- c("A", "B", "C")
print(x)
```

Output:

r
Copy code
```
  A   B   C
 10  20  30
```

Now you can access or modify elements by their names:

r
Copy code
```
# Access by name
```

```
print(x["A"])

# Modify by name
x["B"] <- 25
print(x)
```

Output:

r
Copy code
```
  A   B   C
 10  25  30
```

Vector Element Names in Conditional Operations

You can also use element names in vectorized conditional operations:

r
Copy code
```
# Create a vector with named elements
x <- c(A = 10, B = 20, C = 30)



# Conditional filtering based on names
x[x > 15]
```

Output:

r
Copy code
```
B C

20 30
```

Example: Adding Names Dynamically

You can also dynamically assign names to vector elements. For instance, if you have a vector of numbers and a corresponding vector of names, you can combine them:

r
Copy code
```
# Numeric vector and names
numbers <- c(100, 200, 300)
labels <- c("X", "Y", "Z")

# Assign names from another vector
names(numbers) <- labels
print(numbers)
```

Output:

r

Copy code
```
  X   Y   Z
100 200 300
```

---

5. Summary of Key Concepts

- **Vectorized Operations**: R performs operations on entire vectors without the need for explicit loops. Operations are applied element-wise across the vector.
- **Filtering**: Use logical conditions to filter vectors. This is done using logical indexing, which is a vectorized operation.
- **If-Then-Else**: Use ifelse() for vectorized conditional logic, where you can specify what happens for TRUE and FALSE conditions.
- **Vector Element Names**: You can assign names to the elements of a vector, which allows you to access and manipulate the vector by name.

UNIT II Matrices Creating matrices – Matrix Operations – Applying Functions to Matrix Rows and Columns – Adding and deleting rows and columns - Vector/Matrix Distinction – Avoiding Dimension Reduction – Higher Dimensional arrays – lists – Creating lists – General list operations – Accessing list components and values – applying functions to lists – recursive lists.

Matrices

Matrices in R are fundamental data structures, particularly useful in statistical and mathematical computations. They are essential because they provide an efficient way to store and manipulate tabular data where all elements are of the same type (usually numeric). Here's why matrices are important in R:

1. Compact Representation of Data

A matrix is a two-dimensional array-like structure, where data is stored in rows and columns. It is well-suited for situations where all data is homogeneous.

Example:

R
```
matrix_example <- matrix(1:6, nrow = 2, ncol = 3)
print(matrix_example)
```

Output:

```
     [,1] [,2] [,3]

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

---

2. Simplifies Mathematical Operations

Matrices are optimized for matrix algebra, such as addition, subtraction, multiplication, and solving systems of linear equations.

Example:

```
# Matrix addition
A <- matrix(1:4, nrow = 2)
B <- matrix(5:8, nrow = 2)
C <- A + B
print(C)
```

Output:

```
     [,1] [,2]
[1,]    6    8
[2,]    8   10
```

3. Efficient Data Manipulation

Matrix operations are vectorized in R, meaning they are faster and more efficient than loops for large datasets.

Example:

```
R
# Element-wise multiplication
A * B
```

4. Widely Used in Statistical Analysis

Matrices are the basis for many statistical techniques, such as:

- Linear regression
- Principal Component Analysis (PCA)
- Eigenvalue decomposition
- Correlation and covariance calculations

---

5. Integrates with Other R Functions

Matrices are supported by many R functions, such as solve() for solving linear systems, t() for transpose, and eigen() for eigenvalues.

Example:

```
R
# Transpose of a matrix
t(A)
```

---

6. Visualization and Modeling

Matrices are often used to store and manipulate data for visualization (e.g., heatmaps) and models that require structured input data.

In summary, matrices in R are a versatile and essential tool for numerical and statistical programming, enabling efficient data handling and mathematical computations.

Matrices : Creating matrices

Creating matrices in R programming involves using the matrix() function, which allows you to define a matrix by specifying its elements, dimensions, and layout (row-wise or column-wise). Below are the basic steps and examples to create matrices in R:

---

1. Creating a Simple Matrix

R

```
# Create a 3x3 matrix with numbers 1 to 9
matrix_example <- matrix(1:9, nrow = 3, ncol = 3)
print(matrix_example)
```

Output:

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

---

2. Changing the Filling Order

By default, R fills a matrix column-wise. You can change it to row-wise using the byrow argument.

R

```
# Create a 3x3 matrix filled row-wise
matrix_rowwise <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
print(matrix_rowwise)
```

Output:

```
     [,1] [,2] [,3]

[1,
```

Matrix operations in R are straightforward and intuitive. R provides built-in functions to perform a wide variety of matrix operations, including basic arithmetic, transformations, and advanced computations like decompositions. Here's an overview of key matrix operations in R:

---

1. Basic Arithmetic Operations

Matrix operations in R are element-wise by default.

Addition and Subtraction

```
R
A <- matrix(1:4, nrow = 2)
B <- matrix(5:8, nrow = 2)

# Addition
C_add <- A + B
print(C_add)

# Subtraction
C_sub <- A - B
print(C_sub)
```

Output:

```
csharp
Addition:
     [,1] [,2]
[1,]    6   10
[2,]    8   12

Subtraction:
     [,1] [,2]
[1,]   -4   -4
[2,]   -4   -4
```

Multiplication (Element-Wise)

```
R
C_mult <- A * B
print(C_mult)
```

Output:

```
css
Copy code
     [,1] [,2]

[1,]    5   21
[2,]   12   32
```

Division (Element-Wise)

```
R
C_div <- A / B
print(C_div)
```

Output:

```
css
Copy code
     [,1]      [,2]
```

```
[1,] 0.2  0.4285714
[2,] 0.4  0.5000000
```

---

2. Matrix Multiplication

Matrix multiplication (dot product) is performed using the %*% operator.

```
R
Copy codeA <- matrix(c(1, 2, 3, 4), nrow = 2)
B <- matrix(c(2, 0, 1, 2), nrow = 2)

# Matrix multiplication
C_dot <- A %*% B
print(C_dot)
```

Output:

```
     [,1] [,2]
[1,]    4    4
[2,]   10    8
```

---

3. Transpose of a Matrix

The t() function transposes a matrix.

```
R
transpose <- t(A)
print(transpose)
```

Output:

```
css
Copy code
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

---

4. Determinant

The det() function calculates the determinant of a square matrix.

```
R
determinant <- det(A)
print(determinant)
```

---

5. Inverse of a Matrix

The solve() function computes the inverse of a matrix.

```
inverse <- solve(A)
print(inverse)
```

---

6. Eigenvalues and Eigenvectors

The eigen() function computes the eigenvalues and eigenvectors.

```
R
eigen_result <- eigen(A)
print(eigen_result)
```

---

7. Diagonal Operations

The diag() function extracts or creates diagonal matrices.

- **Extract diagonal:**

  ```
  R
  diag(A)
  ```

- **Create diagonal matrix:**

  ```
  R
  diag(c(1, 2, 3))
  ```

---

8. Identity Matrix

The diag() function can also create an identity matrix.

```
R
I <- diag(3)  # 3x3 identity matrix
print(I)
```

---

9. Solving Linear Systems

Solve a system of linear equations AX=BAX = BAX=B using solve().

```
R
A <- matrix(c(2, 1, 1, 3), nrow = 2)
B <- c(1, 2)

# Solve for X
X <- solve(A, B)


print(X)
```

---

10. Singular Value Decomposition (SVD)

The svd() function computes the singular value decomposition.

R
```
svd_result <- svd(A)
print(svd_result)
```

---

11. Element Access and Manipulation

- Access specific elements: A[1, 2]
- Access entire row: A[1,]
- Access entire column: A[, 2]

These operations allow efficient numerical computation and data manipulation, making matrices one of the most powerful tools in R!

Applying Functions to Matrix Rows and Columns

Applying functions to rows or columns of a matrix in R is a common task. R provides several tools for this, including the apply() function, which is versatile and efficient. Here's a guide to using it and other relevant functions:

---

1. Using **apply()**

The apply() function allows you to apply a function to the rows or columns of a matrix.

Syntax:
R
```
apply(X, MARGIN, FUN, ...)
```

- X: The matrix.
- MARGIN: 1 for rows, 2 for columns.
- FUN: The function to apply (e.g., sum, mean).

Example: Row and Column Sums
R
```
# Create a matrix
matrix_example <- matrix(1:12, nrow = 3, ncol = 4)

# Sum of rows
row_sums <- apply(matrix_example, 1, sum)
print(row_sums)
```

```
# Sum of columns



column_sums <- apply(matrix_example, 2, sum)
print(column_sums)
```

Output:

```
csharp
Copy code
Row sums:
[1] 22 26 30

Column sums:
[1] 12 15 18 21
```

Example: Applying a Custom Function

You can define your own function to apply.

```
R
# Mean of rows
row_means <- apply(matrix_example, 1, mean)
print(row_means)

# Custom function: Range of each column
column_ranges <- apply(matrix_example, 2, function(x) max(x) - min(x))
print(column_ranges)
```

---

2. Using **rowSums()**, **colSums()**, **rowMeans()**, **colMeans()**

For common operations like sums and means, R has optimized functions that are faster than apply().

```
R
# Row and column sums
row_sums <- rowSums(matrix_example)
column_sums <- colSums(matrix_example)

# Row and column means
row_means <- rowMeans(matrix_example)
column_means <- colMeans(matrix_example)

print(row_sums)
print(column_sums)
print(row_means)
print(column_means)
```

---

3. Using **lapply()** or **sapply()**

When dealing with individual rows or columns as lists or vectors, lapply() and sapply() can be useful.

***Example: Applying Functions with** sapply()*

```
R
# Transpose matrix to treat rows as columns
row_means <- sapply(1:nrow(matrix_example), function(i) mean(matrix_example[i, ]))
print(row_means)
```

---

4. Using the **dplyr** Package

For more advanced row/column operations, the dplyr package is handy.

Example: Summing Rows

```
R
library(dplyr)

# Convert matrix to data frame
df <- as.data.frame(matrix_example)

# Row sums using dplyr
df <- df %>% mutate(row_sum = rowSums(across()))
print(df)
```

---

5. Working with Specific Columns or Rows

You can use indexing to apply functions to specific rows or columns.

Example: Apply to Specific Columns

```
R
# Square values in the first column
matrix_example[, 1] <- matrix_example[, 1]^2
print(matrix_example)
```

By using these tools, you can efficiently manipulate and analyze rows and columns of matrices in R!

Adding and deleting rows and columns

Adding and deleting rows and columns in a matrix in R can be easily achieved using functions like rbind() and cbind() for adding rows and columns, and indexing for deleting them. Here's how you can do it:

---

1. Adding Rows to a Matrix

Use the rbind() function to add one or more rows.

Example: Adding a Row

```
R
# Create a matrix
matrix_example <- matrix(1:6, nrow = 2, ncol = 3)
print(matrix_example)

# Add a new row
new_row <- c(7, 8, 9)
matrix_with_row <- rbind(matrix_example, new_row)
print(matrix_with_row)
```

Output:

```
less
Original Matrix:

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

Matrix After Adding Row:
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]    7    8    9
```

2. Adding Columns to a Matrix

Use the cbind() function to add one or more columns.

Example: Adding a Column

```
R
# Add a new column
new_column <- c(10, 11)
matrix_with_column <- cbind(matrix_example, new_column)
print(matrix_with_column)
```

Output:

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5   10
[2,]    2    4    6   11
```

---

3. Deleting Rows from a Matrix

You can delete rows by indexing and omitting the rows you want to remove.

Example: Deleting a Row

```
R
# Delete the second row
matrix_without_row <- matrix_example[-2, ]
print(matrix_without_row)
```

Output:

```
css
     [,1] [,2] [,3]
[1,]    1    3    5
```

---

4. Deleting Columns from a Matrix

Similarly, delete columns by indexing and omitting the columns.

Example: Deleting a Column

```
R
# Delete the third column
matrix_without_column <- matrix_example[, -3]
print(matrix_without_column)
```

Output:

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

---

5. Adding Multiple Rows or Columns

You can add multiple rows or columns at once using rbind() and cbind().

Example: Adding Multiple Rows

```
R
# Add multiple rows
new_rows <- matrix(c(7, 8, 9, 10, 11, 12), nrow = 2)
matrix_with_rows <- rbind(matrix_example, new_rows)
print(matrix_with_rows)
```

Example: Adding Multiple Columns

```
R
# Add multiple columns
new_columns <- matrix(c(10, 11, 12, 13), ncol = 2)
matrix_with_columns <- cbind(matrix_example, new_columns)
print(matrix_with_columns)
```

---

6. Replacing Rows or Columns

Replace rows or columns by assigning new values.

Example: Replace a Row

```
R
matrix_example[1, ] <- c(100, 101, 102)
print(matrix_example)
```

Example: Replace a Column

```
R
matrix_example[, 2] <- c(200, 201)
print(matrix_example)
```

These techniques provide flexibility in managing matrix structures, making it easy to modify and adapt data in R

Vector/Matrix Distinction

In R, **vectors** and **matrices** are fundamental data structures, but they are distinct in their structure and intended usage. Understanding the distinction between them is crucial for effective data manipulation and computation. Here's a breakdown:

---

1. Vector:

A vector is a **one-dimensional** array that contains elements of the same type (e.g., numeric, character, logical).

Key Characteristics:

- One-dimensional: A single row or column of data.
- Homogeneous: All elements must be of the same type.
- Created using the c() or similar functions.

Example:

```
R
# Create a numeric vector
v <- c(1, 2, 3, 4)
print(v)
```

Output:

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

2. Matrix:

A matrix is a **two-dimensional** array with rows and columns, where all elements are of the same type.

Key Characteristics:

- Two-dimensional: Has rows and columns (e.g., m×nm \times nm×n).
- Homogeneous: All elements must be of the same type.
- Created using the matrix() function, or by combining vectors with rbind() or cbind().

Example:

```
# Create a 2x2 numeric matrix
```

```
m <- matrix(1:4, nrow = 2)
print(m)
```

Output:

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

Key Distinctions:

Feature	**Vector**	**Matrix**
Dimension	1D (length)	2D (rows and columns)
Creation	c(), seq(), rep()	matrix(), rbind(), cbind()
Homogeneity	Homogeneous	Homogeneous
Access	Single index: v[1]	Row and column indices: m[1, 2]
Structure	Linear sequence of elements	Rectangular grid of elements

---

3. Conversion Between Vectors and Matrices

Convert a Vector to a Matrix

You can convert a vector into a matrix using matrix().

```
R
v <- c(1, 2, 3, 4)
m <- matrix(v, nrow = 2, ncol = 2)
print(m)
```

Output:

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

Flatten a Matrix to a Vector

Use the as.vector() function or simply reference the matrix.

```
R
v_from_matrix <- as.vector(m)
print(v_from_matrix)
```

Output:

```
csharp
 [1] 1 2 3 4
```

---

4. Dimensionality

The dim() function reveals the dimensions of an object.

- **Vector:** A vector has no dim attribute; only its length matters.

```
R
length(v)  # Returns 4
dim(v)     # Returns NULL
```

- **Matrix:** A matrix always has dimensions.

```
R
dim(m)  # Returns c(2, 2)
```

---

5. When to Use Vectors or Matrices?

- **Use Vectors:** When working with linear data or a single set of observations.
- **Use Matrices:** When dealing with tabular data, linear algebra, or computations requiring row-column relationships.

Summary

- **Vectors**: Simplest structure, 1D, homogeneous.
- **Matrices**: More structured, 2D, homogeneous, optimized for mathematical operations.

Understanding these distinctions ensures you use the appropriate structure for your tasks in R!

Avoiding Dimension Reduction

In R, **dimension reduction** occurs when operations on matrices or arrays simplify the structure, often converting matrices to vectors. This can happen when selecting a single row or column from a matrix, as R by default drops the dimensions. To avoid this, you can explicitly control this behavior using the drop = FALSE argument. Here's how to handle it:

1. The Default Behavior

When selecting a single row or column from a matrix, R reduces the matrix to a vector.

Example: Dimension Reduction

```
R
# Create a matrix
m <- matrix(1:9, nrow = 3, ncol = 3)
print(m)

# Select the first row
row1 <- m[1, ]
print(row1)
```

Output:

```
Original Matrix:
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

Row Selection (Reduced to Vector):
[1] 1 4 7
```

In this example, m[1,] extracts the first row as a vector, dropping the matrix structure.

---

2. Preventing Dimension Reduction

Use drop = FALSE when indexing to preserve the matrix structure.

Example: Retaining Matrix Structure

```
R
# Select the first row and keep it as a matrix
row1_matrix <- m[1, , drop = FALSE]
print(row1_matrix)

# Select the first column and keep it as a matrix
col1_matrix <- m[, 1, drop = FALSE]
print(col1_matrix)
```

Output:

```
less
Row as Matrix:
     [,1] [,2] [,3]
[1,]    1    4    7

Column as Matrix:
     [,1]
[1,]    1
[2,]    2
[3,]    3
```

---

3. General Syntax

To avoid dimension reduction in any operation:

- When extracting **rows**: matrix[row, , drop = FALSE]
- When extracting **columns**: matrix[, col, drop = FALSE]
- When extracting **elements** but retaining structure: Use slicing instead of single indexing.

---

4. Arrays and Dimension Preservation

In multi-dimensional arrays, a similar issue occurs, but you can use drop = FALSE to retain higher-dimensional structures.

Example: 3D Array

R

```
# Create a 3D array


array_example <- array(1:24, dim = c(3, 4, 2))

# Select a 2D slice without reducing dimensions
slice <- array_example[,,1, drop = FALSE]
print(dim(slice))  # Dimensions are preserved
```

---

5. Why Avoid Dimension Reduction?

- **Data Consistency**: Prevents errors in downstream operations by preserving the expected structure.
- **Code Robustness**: Avoids implicit conversions that could lead to unexpected behavior.

---

6. Forcing Dimension Reduction

If you want to enforce dimension reduction (e.g., explicitly convert to a vector), you can use as.vector() or deliberately omit drop.

Summary

- By default, R simplifies dimensions when selecting rows or columns from matrices.
- Use drop = FALSE to preserve the matrix or array structure.
- Always check your intended structure with dim() to ensure correctness in your workflow.

In R, **dimension reduction** occurs when operations on matrices or arrays simplify the structure, often converting matrices to vectors. This can happen when selecting a single row or column from a matrix, as R by default drops the dimensions. To avoid this, you can explicitly control this behavior using the drop = FALSE argument. Here's how to handle it:

1. The Default Behavior

When selecting a single row or column from a matrix, R reduces the matrix to a vector.

Example: Dimension Reduction

```
R
# Create a matrix
m <- matrix(1:9, nrow = 3, ncol = 3)
print(m)

# Select the first row
row1 <- m[1, ]
print(row1)
```

Output: Original Matrix:

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

Row Selection (Reduced to Vector):

```
[1] 1 4 7
```

In this example, m[1,] extracts the first row as a vector, dropping the matrix structure.

2. Preventing Dimension Reduction

Use drop = FALSE when indexing to preserve the matrix structure.

Example: Retaining Matrix Structure

```
R
# Select the first row and keep it as a matrix
row1_matrix <- m[1, , drop = FALSE]
print(row1_matrix)

# Select the first column and keep it as a matrix
col1_matrix <- m[, 1, drop = FALSE]
print(col1_matrix)
```

Output:

```
less
Copy code
Row as Matrix:
     [,1] [,2] [,3]
[1,]    1    4    7

Column as Matrix:
     [,1]
[1,]    1
[2,]    2
```

```
[3,]    3
```

3. General Syntax

To avoid dimension reduction in any operation:

- When extracting **rows**: matrix[row, , drop = FALSE]
- When extracting **columns**: matrix[, col, drop = FALSE]
- When extracting **elements** but retaining structure: Use slicing instead of single indexing.

---

4. Arrays and Dimension Preservation

In multi-dimensional arrays, a similar issue occurs, but you can use drop = FALSE to retain higher-dimensional structures.

Example: 3D Array

```
R
# Create a 3D array
array_example <- array(1:24, dim = c(3, 4, 2))

# Select a 2D slice without reducing dimensions
slice <- array_example[,,1, drop = FALSE]
print(dim(slice))  # Dimensions are preserved
```

---

5. Why Avoid Dimension Reduction?

- **Data Consistency**: Prevents errors in downstream operations by preserving the expected structure.
- **Code Robustness**: Avoids implicit conversions that could lead to unexpected behavior.

---

6. Forcing Dimension Reduction

If you want to enforce dimension reduction (e.g., explicitly convert to a vector), you can use as.vector() or deliberately omit drop.

Summary

- By default, R simplifies dimensions when selecting rows or columns from matrices.
- Use drop = FALSE to preserve the matrix or array structure.
- Always check your intended structure with dim() to ensure correctness in your workflow.

Higher Dimensional arrays

Higher-dimensional arrays in R are an extension of matrices, allowing for data storage and manipulation in more than two dimensions. They are particularly useful for representing complex data, such as tensors or datasets with multiple attributes.

1. Creating Higher-Dimensional Arrays

Use the array() function to create arrays with more than two dimensions.

Syntax:

```
R
array(data, dim, dimnames)
```

- data: The elements to be included in the array.
- dim: A vector specifying the dimensions (e.g., number of rows, columns, and additional dimensions).
- dimnames: Optional names for dimensions.

Example: Creating a 3D Array

```
R
# Create a 3D array with dimensions 3x3x2
array_3d <- array(1:18, dim = c(3, 3, 2))
print(array_3d)
```

Output

```
, , 1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
, , 2
     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
```

---

2. Naming Dimensions

You can add meaningful names to the dimensions for better readability.

Example: Adding Dimension Names

```
R
# Add names to dimensions
dimnames(array_3d) <- list(
  rows = c("R1", "R2", "R3"),
  cols = c("C1", "C2", "C3"),
  layers = c("L1", "L2")
)
print(array_3d)
```

Output:

```
markdown
, , L1
```

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

, , L2
     C1 C2 C3
R1   10 13 16
R2   11 14 17
R3   12 15 18
```

3. Accessing Elements

Access elements using indices for all dimensions.

Access Specific Element

```
# Access the element at row 2, column 3, layer 1
array_3d[2, 3, 1]
```

Output:

```
 [1] 8
```

Access Entire Slices

R
Copy code

```
# Access the entire first layer
array_3d[, , 1]

# Access the second column across all layers
array_3d[, 2, ]
```

---

4. Manipulating Higher-Dimensional Arrays

Reshaping

Change the dimensions of an array using the dim() function.

R

```
# Reshape array to 2x3x3
dim(array_3d) <- c(2, 3, 3)
print(array_3d)
```

Combining Arrays

Use functions like abind() from the abind package to combine arrays.

R

```
library(abind)

# Combine along a new dimension
array_combined <- abind(array_3d, array_3d, along = 4)
print(dim(array_combined))
```

---

5. Applying Functions

Use apply() to apply functions along specific dimensions of an array.

Example: Apply Function Across a Dimension

```
R
# Sum along rows for each layer
row_sums <- apply(array_3d, c(1, 3), sum)
print(row_sums)
```

---

6. Common Functions for Arrays

- **dim()**: Get or set the dimensions of an array.
- **dimnames()**: Get or set dimension names.
- **array()**: Create an array.
- **apply()**: Apply a function along specific dimensions.

---

7. Visualization

Use visualization libraries like lattice or ggplot2 for higher-dimensional array data. Flatten data into a 2D structure (e.g., using as.data.frame()) for plotting.

Example: Higher-Dimensional Data Analysis

```
R
# Create a 4D array
array_4d <- array(1:24, dim = c(3, 2, 2, 2))
print(array_4d)


# Extract a 2D slice
slice_2d <- array_4d[,,1,2]
print(slice_2d)
```

Arrays in R are versatile and extend the capabilities of matrices to higher dimensions, making them useful for multi-attribute data representation and manipulation.

Lists

Lists in R

A **list** in R is a versatile data structure that can store elements of different types (e.g., numbers, strings, vectors, matrices, other lists). Unlike vectors or matrices, lists are heterogeneous, making them useful for grouping related but diverse data.

---

1. Creating Lists

Use the list() function to create lists.

Example: Creating a Simple List

```
R
# Create a list with different data types
my_list <- list(
  name = "Alice",
  age = 30,
  scores = c(85, 90, 78),
  matrix = matrix(1:4, nrow = 2)
)
print(my_list)
```

Output:

```
$name

[1] "Alice"

$age
[1] 30


$scores
[1] 85 90 78

$matrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

---

2. General List Operations

a. Naming Elements

You can assign or modify names of list components.

```
R
names(my_list) <- c("Name", "Age", "Scores", "Matrix")

print(my_list)
```

b. Combining Lists

Use c() to combine lists.

```
R
list1 <- list(a = 1, b = 2)

list2 <- list(c = 3, d = 4)
combined_list <- c(list1, list2)
```

print(combined_list)

c. Length of a List

The length() function returns the number of elements in a list.

```
R
print(length(my_list))  # Output: 4
```

---

3. Accessing List Components and Values

a. Using $

Access elements by name.

```
R
print(my_list$Name)  # Output: "Alice"
```

***b. Using [[ ]]*

Access elements by index or name.

```
R
print(my_list[[1]])  # Output: "Alice"
print(my_list[["Name"]])  # Output: "Alice"
```

c. Using [ ]

Returns a sublist, not the element itself.

```
R
print(my_list[1])  # Returns a list containing the first element
```

d. Nested Access

Access elements inside nested lists.

```
R
nested_list <- list(a = list(b = list(c = 5)))
print(nested_list$a$b$c)  # Output: 5
```

---

4. Applying Functions to Lists

***a.** lapply()*

Applies a function to each element and returns a list.

```
R
# Square each element
result <- lapply(my_list$scores, function(x) x^2)
print(result)
```

***b.** sapply()*

Applies a function to each element and simplifies the output to a vector or matrix.

```
R
result <- sapply(my_list$scores, function(x) x^2)
print(result)
```

***c.** vapply()*

A safer version of sapply() with pre-specified output type.

```
R
result <- vapply(my_list$scores, function(x) x^2, numeric(1))
print(result)
```

***d.** mapply()*

Applies a function to multiple list-like objects.

```
R
list1 <- list(1, 2, 3)
list2 <- list(4, 5, 6)
result <- mapply(function(x, y) x + y, list1, list2)
print(result)
```

---

5. Recursive Lists

A **recursive list** (or nested list) is a list that contains other lists as elements.

Example: Creating a Recursive List

```
R
recursive_list <- list(
  a = list(x = 1, y = 2),
  b = list(z = 3, w = list(p = 4, q = 5))
)
print(recursive_list)
```

Accessing Nested Components

Use $ or [[ ]] repeatedly or in combination.

```
R
print(recursive_list$b$w$p)  # Output: 4
print(recursive_list[[2]][[2]][["p"]])  # Output: 4
```

---

6. Modifying Lists

You can modify or extend lists by assigning values.

Add or Modify Elements

```
R
# Add a new element
my_list$new_element <- "Hello"
print(my_list)

# Modify an existing element
my_list$Age <- 31
print(my_list)
```

Remove Elements

Set an element to NULL to remove it.

```
R
my_list$Scores <- NULL
print(my_list)
```

Summary

- **Lists** can store heterogeneous data, including vectors, matrices, and other lists.
- Access elements using $, [[ ]], or [ ].
- Use functions like lapply() and sapply() to apply operations to lists.
- Recursive lists allow nested structures, enabling complex data representations.

UNIT III

Creating Data Frames – Matrix-like operations in frames – merging Data frames – Applying functions to Data Frames – Factors and Tables – Factors and levels – Common Functions used with factors – Working with tables – Other factors and table related functions – Control statements – Arithmetic and Boolean operators and values – Default Values for arguments – Returning Boolean Values – Functions are objects – Recursion

Creating Data Frames

In R, data frames are one of the most commonly used data structures for storing and manipulating tabular data. A data frame is essentially a collection of vectors of equal length, where each vector can be of a different data type (e.g., numeric, character, factor, etc.). Below are several ways to create data frames in R:

1. Creating a Data Frame from Vectors

You can create a data frame by combining vectors using the data.frame() function.

```
# Create vectors

name <- c("Alice", "Bob", "Charlie")

age <- c(25, 30, 35)
```

```
height <- c(5.5, 6.0, 5.8)

# Combine vectors into a data frame

df <- data.frame(Name = name, Age = age, Height = height)

# Print the data frame

print(df)
```

Output:

```
     Name Age Height
1   Alice  25    5.5
2     Bob  30    6.0
3 Charlie  35    5.8
```

2. Creating a Data Frame from a List

You can also create a data frame from a list of vectors.

```
# Create a list of vectors

my_list <- list(Name = c("Alice", "Bob", "Charlie")

  Age = c(25, 30, 35),

  Height = c(5.5, 6.0, 5.8))

# Convert the list to a data frame

df <- data.frame(my_list)

# Print the data frame

print(df)
```

Output:

```
Name Age Height
1   Alice  25    5.5
2     Bob  30    6.0
3 Charlie  35    5.8
```

You can convert a matrix into a data frame using the as.data.frame() function.

```
# Create a matrix

my_matrix <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)

# Convert the matrix to a data frame

df <- as.data.frame(my_matrix)

# Print the data frame

print(df)
```

Output:

```
  V1 V2 V3

1  1  3  5

2  2  4  6
```

4. Creating a Data Frame from External Data

You can read data from external files (e.g., CSV, Excel) into a data frame using functions like read.csv() or read.table().

```
# Read a CSV file into a data frame

df <- read.csv("path/to/your/file.csv")

# Print the data frame

print(df)

   Name Age Height

1   Alice  25    5.5

2     Bob  30    6.0

3 Charlie  35    5.8
```

5. Creating an Empty Data Frame

You can create an empty data frame and then add columns to it.

```
# Create an empty data frame

df <- data.frame()

# Add columns to the data frame

df$Name <- c("Alice", "Bob", "Charlie")
```

```
df$Age <- c(25, 30, 35)

# Print the data frame

print(df)
```

Output:

```
  Name Age
1   Alice  25
2     Bob  30
3 Charlie  35
```

6. Creating a Data Frame with **tibble** (from the **tibble** package)

The tibble package provides a modern alternative to data frames. You can create a tibble using the tibble() function.

```
# Install and load the tibble package

install.packages("tibble")

library(tibble)

# Create a tibble

df <- tibble(Name = c("Alice", "Bob", "Charlie"),

             Age = c(25, 30, 35)

  Height = c(5.5, 6.0, 5.8))

# Print the tibble

print(df)
```

Output:

```
# A tibble: 3 × 3
  Name     Age Height
  <chr>  <dbl>  <dbl>
1 Alice     25    5.5
2 Bob       30    6
```

```
3 Charlie   35    5.8
```

The data.table package provides an enhanced version of data frames. You can create a data table using the data.table() function.

```
# Install and load the data.table package

install.packages("data.table")

library(data.table)

# Create a data table

dt <- data.table(Name = c("Alice", "Bob", "Charlie"),

                 Age = c(25, 30, 35),

                 Height = c(5.5, 6.0, 5.8))

# Print the data table

print(dt)
```

Output

```
      Name Age Height

1:   Alice  25    5.5

2:     Bob  30    6.0

3: Charlie  35    5.8
```

Summary

- Use data.frame() to create a data frame from vectors or a list.
- Use as.data.frame() to convert a matrix to a data frame.
- Use read.csv() or read.table() to read external data into a data frame.
- Use tibble() from the tibble package for a modern alternative to data frames.
- Use data.table() from the data.table package for enhanced data frames.

These methods provide flexibility in creating and manipulating data frames in R, depending on specific needs.

Matrix-like operations in frames

In R, data frames are designed to store tabular data, similar to matrices, but with the added flexibility of allowing columns to contain different data types (e.g., numeric, character, factor). While data frames are not matrices, you can perform matrix-like operations on them by converting them to matrices or using specific functions that work with data frames. Below are some common matrix-like operations can be performed on data frames in R:

1. **Convert a Data Frame to a Matrix**

To perform matrix operations, you can convert a data frame to a matrix using the `as.matrix()` function. Note that all columns must be of the same data type (e.g., numeric) for this to work properly.

```
# Create a data frame

df <- data.frame(A = c(1, 2, 3), B = c(4, 5, 6), C = c(7, 8, 9))

# Convert to a matrix

mat <- as.matrix(df)

# Print the matrix

print(mat)
```

Output:

```
  A B C
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

2. **Matrix-like Operations on Data Frames**

Even without converting to a matrix, you can perform some matrix-like operations directly on data frames.

Transpose a Data Frame

Use the `t()` function to transpose a data frame. Note that the result will be a matrix, not a data frame.

```
# Transpose the data frame

transposed_df <- t(df)

# Print the transposed result

print(transposed_df)
```

Output:

```
  [,1] [,2] [,3]
A    1    2    3
B    4    5    6
C    7    8    9
```

Row and Column Sums

Use `rowSums()` and `colSums()` to calculate sums of rows and columns, respectively.

```
# Column sums

col_sums <- colSums(df)

# Row sums

row_sums <- rowSums(df)
```

```
print(col_sums)
print(row_sums)
```

Output:

```
A  B  C
6 15 24
[1] 12 15 18
```

#### **Row and Column Means**

Use `rowMeans()` and `colMeans()` to calculate means of rows and columns, respectively.

```
# Column means
col_means <- colMeans(df)
# Row means
row_means <- rowMeans(df)
print(col_means)
print(row_means)
```

Output:

```
A B C
2 5 8
[1] 4 5 6
```

### 3. **Subsetting Data Frames (Similar to Matrix Indexing)**

You can subset data frames using row and column indices, similar to matrix indexing.

```
# Subset the first two rows and columns
subset_df <- df[1:2, 1:2]
print(subset_df)
```

Output:

```
  A B
1 1 4
2 2 5
```

### 4. **Matrix Multiplication**

To perform matrix multiplication, convert the data frame to a matrix and use the `%*%` operator.

```
# Create two data frames
df1 <- data.frame(A = c(1, 2), B = c(3, 4))
df2 <- data.frame(C = c(5, 6), D = c(7, 8))
# Convert to matrices
mat1 <- as.matrix(df1)
```

```
mat2 <- as.matrix(df2)
# Perform matrix multiplication
result <- mat1 %*% mat2
print(result)
```

Output:

```
    C  D
[1,] 23 31
[2,] 34 46
```

### 5. **Element-wise Operations**

You can perform element-wise operations (e.g., addition, subtraction, multiplication) on data frames, similar to matrices.

```
# Create two data frames
df1 <- data.frame(A = c(1, 2, 3), B = c(4, 5, 6))
df2 <- data.frame(A = c(7, 8, 9), B = c(10, 11, 12))

# Element-wise addition
result <- df1 + df2
print(result)
```

Output:

```
  A  B
1 8 14
2 10 16
3 12 18
```

### 6. **Apply Functions to Rows or Columns**

Use the `apply()` function to apply a function to rows or columns of a data frame.

```
# Apply the sum function to columns
col_sums <- apply(df, 2, sum)

# Apply the sum function to rows
row_sums <- apply(df, 1, sum)
print(col_sums)
print(row_sums)
```

Output:

```
A  B  C
```

```
6 15 24
[1] 12 15 18
```

### 7. **Diagonal Operations**

If your data frame is square (same number of rows and columns), you can extract or modify the diagonal using the `diag()` function.

```
# Extract the diagonal
diagonal <- diag(as.matrix(df))
print(diagonal)
```

Output:

```
 [1] 1 5 9
```

### 8. **Using `dplyr` for Matrix-like Operations**

The `dplyr` package provides a powerful set of tools for manipulating data frames. While not strictly matrix-like, it allows for similar operations in a more user-friendly way.

```
# Install and load dplyr
install.packages("dplyr")
library(dplyr)
# Create a data frame

df <- data.frame(A = c(1, 2, 3), B = c(4, 5, 6), C = c(7, 8, 9))
# Calculate row sums using dplyr
df <- df %>% mutate(RowSum = rowSums(.))
print(df)
```

Output:

```
  A B C RowSum
1 1 4 7     12
2 2 5 8     15
3 3 6 9     18
```

### Summary

- Use `as.matrix()` to convert a data frame to a matrix for matrix-specific operations.
- Perform matrix-like operations (e.g., transposition, row/column sums, matrix multiplication) on data frames.

- Use `apply()` for applying functions to rows or columns.
- Use `dplyr` for advanced data frame manipulations.

These techniques allow to work with data frames in a way that mimics matrix operations while retaining the flexibility of data frames.

MERGING DATA FRAMES

Merging data frames is a common operation in R, especially when working with datasets that need to be combined based on shared columns or keys. R provides several functions to merge data frames, including `merge()`, `dplyr` functions, and `data.table` methods. Below are the most common ways to merge data frames in R:

### 1. **Using `merge()`**

The `merge()` function is a base R function for combining two data frames based on common columns (keys).

#### **Basic Syntax**

```
merge(x, y, by = "key_column", all = FALSE, all.x = FALSE, all.y = FALSE)
```

- `x`, `y`: The data frames to merge.

- `by`: The column(s) to merge by (common key).

- `all`: If `TRUE`, includes all rows from both data frames (full outer join).

- `all.x`: If `TRUE`, includes all rows from `x` (left outer join).

- `all.y`: If `TRUE`, includes all rows from `y` (right outer join).

#### **Example**

```
# Create two data frames

df1 <- data.frame(ID = c(1, 2, 3), Name = c("Alice", "Bob", "Charlie"))

df2 <- data.frame(ID = c(2, 3, 4), Age = c(25, 30, 35))


# Merge by the "ID" column (inner join by default)


merged_df <- merge(df1, df2, by = "ID")

print(merged_df)
```

Output:

```
  ID    Name Age
1  2     Bob  25
2  3 Charlie  30
```

#### **Types of Joins with `merge()`**

- **Inner Join**: Only rows with matching keys in both data frames.

  merge(df1, df2, by = "ID")

- **Left Join**: All rows from `df1` and matching rows from `df2`.

  merge(df1, df2, by = "ID", all.x = TRUE)

- **Right Join**: All rows from `df2` and matching rows from `df1`.

  merge(df1, df2, by = "ID", all.y = TRUE)

-

Full Outer Join: All rows from both data frames.

  merge(df1, df2, by = "ID", all = TRUE)

  ### 2. **Using `dplyr` for Merging**

The `dplyr` package provides a more intuitive and readable way to merge data frames using functions like `left_join()`, `right_join()`, `inner_join()`, and `full_join()`.

#### **Install and Load `dplyr`**

```
install.packages("dplyr")

library(dplyr)
```

#### **Example**

```
# Create two data frames

df1 <- data.frame(ID = c(1, 2, 3), Name = c("Alice", "Bob", "Charlie"))

df2 <- data.frame(ID = c(2, 3, 4), Age = c(25, 30, 35))

# Left join

left_joined <- left_join(df1, df2, by = "ID")


# Right join

right_joined <- right_join(df1, df2, by = "ID")

# Inner join

inner_joined <- inner_join(df1, df2, by = "ID")

# Full join

full_joined <- full_join(df1, df2, by = "ID")

print(left_joined)

print(right_joined)


print(inner_joined)

print(full_joined)
```

Output:

```
# Left Join
  ID    Name Age
1  1   Alice  NA
2  2     Bob  25
3  3 Charlie  30
# Right Join
  ID    Name Age
1  2     Bob  25
2  3 Charlie  30
3  4    <NA>  35
# Inner Join

  ID    Name Age
1  2     Bob  25
2  3 Charlie  30
# Full Join
  ID    Name Age
1  1   Alice  NA
2  2     Bob  25
3  3 Charlie  30
4  4    <NA>  35
```

### 3. **Using `data.table` for Merging**

The `data.table` package provides a fast and efficient way to merge data frames.

#### **Install and Load `data.table`**

```
install.packages("data.table")
library(data.table)
```

#### **Example**

```
# Convert data frames to data.tables



dt1 <- as.data.table(df1)
dt2 <- as.data.table(df2)
# Merge using data.table syntax
merged_dt <- merge(dt1, dt2, by = "ID", all = TRUE)
```

```
print(merged_dt)
```

Output:

```
   ID    Name Age
1:  1   Alice  NA
2:  2     Bob  25
3:  3 Charlie  30
4:  4    <NA>  35
```

# 4. **Merging on Multiple Columns**

You can merge data frames on multiple columns by passing a vector of column names to the `by` argument.

#### **Example**

```
# Create data frames with multiple keys
df1 <- data.frame(ID = c(1, 2, 3), Year = c(2021, 2022, 2023), Value = c(10, 20, 30))
df2 <- data.frame(ID = c(1, 2, 3), Year = c(2021, 2022, 2023), Score = c(100, 200, 300))
# Merge on multiple columns
merged_df <- merge(df1, df2, by = c("ID", "Year"))
print(merged_df)
```

Output:

```
 ID Year Value Score
1  1 2021    10   100
2  2 2022    20   200
3  3 2023    30   300
```

### 5. **Handling Non-Matching Column Names**

If the key columns have different names in the two data frames, you can use the `by.x` and `by.y` arguments in `merge()`.

#### **Example**

```
# Create data frames with different key column names
df1 <- data.frame(ID1 = c(1, 2, 3), Name = c("Alice", "Bob", "Charlie"))
df2 <- data.frame(ID2 = c(2, 3, 4), Age = c(25, 30, 35))
# Merge using different key column names
```

```
merged_df <- merge(df1, df2, by.x = "ID1", by.y = "ID2")
print(merged_df)
```

Output:

```
  ID1    Name Age
1   2     Bob  25
2   3 Charlie  30
```

### Summary

- Use `merge()` for basic merging operations in base R.
- Use `dplyr` functions (`left_join()`, `right_join()`, etc.) for more readable and intuitive merging.
- Use `data.table` for fast and efficient merging, especially with large datasets.
- Merge on multiple columns by passing a vector of column names to the `by` argument.
- Handle non-matching column names using `by.x` and `by.y`.

These methods provide flexibility for combining data frames in R, depending on your specific needs.

Applying functions to Data Frames

In R, you can apply functions to DataFrames (which are typically represented as `data.frame` or `tibble` objects) in various ways depending on what you want to achieve. Below are some common methods for applying functions to DataFrames:

### 1. **Applying a Function to Each Column or Row**

- **`apply()`**: Apply a function to the rows or columns of a DataFrame.

```
# Example: Calculate the mean of each column
df <- data.frame(a = 1:5, b = 6:10, c = 11:15)
apply(df, 2, mean)  # 2 means apply to columns
```

  - `1` applies the function to rows.
  - `2` applies the function to columns.

- **`sapply()`**: Simplifies the result of applying a function to each column.

```
sapply(df, mean)
```

- **`lapply()`**: Returns a list after applying a function to each column.

```
lapply(df, mean)
```

- **`rowMeans()`, `rowSums()`, `colMeans()`, `colSums()`**: Specialized functions for row/column operations.

```
rowMeans(df)
colSums(df)
```

### 2. **Applying a Function to Subsets of a DataFrame**

- **`tapply()`**: Apply a function to subsets of a vector based on a factor.

```
df <- data.frame(group = c("A", "A", "B", "B"), value = c(1, 2, 3, 4))
tapply(df$value, df$group, mean)
```

- **`aggregate()`**: Apply a function to subsets of a DataFrame.

```
aggregate(value ~ group, data = df, mean)
```

- **`by()`**: Apply a function to subsets of a DataFrame.

```
by(df, df$group, function(x) mean(x$value))
```

### 3. **Applying a Function to Each Element**

- **`mapply()`**: Apply a function to multiple vectors or lists element-wise.

```
mapply(function(x, y) x + y, df$a, df$b)
```

  - **`Map()`**: Similar to `mapply()` but returns a list.

```
Map(function(x, y) x + y, df$a, df$b)
```

### 4. **Using `dplyr` for DataFrame Operations**

The `dplyr` package provides a more intuitive and efficient way to work with DataFrames.

- **`mutate()`**: Apply a function to create or modify columns.

```
library(dplyr)
df <- df %>% mutate(new_col = a + b)
```

- **`summarize()`**: Apply a function to summarize data.

```
df %>% summarize(mean_a = mean(a), mean_b = mean(b))
```

- **`group_by()` + `summarize()`**: Apply a function to grouped data.

```
   df %>% group_by(group) %>% summarize(mean_value = mean(value))
```

- **`across()`**: Apply a function to multiple columns.

```
df %>% summarize(across(everything(), mean))
```

### 5. **Using `purrr` for Functional Programming**

The `purrr` package provides tools for functional programming.

- **`map()`**: Apply a function to each column.

```
library(purrr)
map(df, mean)
```

- **`map_dfr()`**: Apply a function and return a DataFrame.

map_dfr(df, ~ .x * 2)

### 6. **Custom Functions**

You can define your own functions and apply them to DataFrames.

```
my_function <- function(x) {
  x * 2
}
df <- df %>% mutate(new_col = my_function(a))
```

### Example: Combining Methods

```
# Create a DataFrame
df <- data.frame(a = 1:5, b = 6:10, c = 11:15)
# Apply a custom function to each column
df <- df %>% mutate(across(everything(), ~ .x * 2))


# Summarize the DataFrame
df %>% summarize(across(everything(), list(mean = mean, sum = sum)))
```

This will double each value in the DataFrame and then calculate the mean and sum for each column.

These are some of the most common ways to apply functions to DataFrames in R. The choice of method depends on the specific task and the structure of your data.

Factors and Tables

In R, factors and tables are essential concepts that help in managing categorical data.

1. Factors

Factors in R are used to represent categorical data. They are R's way of handling variables that have a fixed number of unique values or levels. For example, a "Gender" variable with values "Male" and "Female" can be represented as a factor. Factors are useful because they store both the values and the underlying levels, which makes them efficient when working with categorical data.

Creating a Factor

You can create a factor using the factor() function. For example:

r
Copy
```
gender <- factor(c("Male", "Female", "Female", "Male"))
print(gender)
```

This creates a factor variable gender with two levels: Male and Female.

Levels of a Factor

You can access the levels of a factor using the levels() function:

r
Copy
```
levels(gender)  # Shows "Male" and "Female"
```

Changing Factor Levels

You can also modify the levels of a factor:

r
Copy
```
gender <- factor(c("Male", "Female", "Female", "Male"), levels = c("Male", "Female"))
```

2. Tables

In R, a table is an object that shows the frequency of each category in a factor or vector. You can create a table using the table() function. A table is essentially a contingency table, where each element represents the count of occurrences of a particular level of a factor.

Creating a Table from a Factor

Here’s how you can create a frequency table from a factor:

r
Copy
```
table(gender)
```

This will output the number of occurrences of each level of the gender factor:

markdown
Copy
```
Female   Male
   2      2
```

Creating a Table from a Vector

You can also create a table from a general vector, not just a factor:

r
Copy
```
ages <- c(23, 34, 23, 45, 34, 23, 45, 45, 23)
table(ages)
```

This will display the frequency of each unique age:

Copy
```
23  34  45
4   2   3
```

Cross-tabulation (Contingency Table)

You can create a cross-tabulation using two categorical variables (factors):

```
r
Copy
education <- factor(c("High School", "College", "College", "High School", "College"))


table(gender, education)
```

This will show how many males and females have each level of education.

Table Summary

You can also summarize a table by using the summary() function:

```
r
Copy
summary(table(gender))
```

Example combining both:

```
r
Copy
# Creating factors
gender <- factor(c("Male", "Female", "Female", "Male"))
education <- factor(c("High School", "College", "College", "High School"))

# Create a table (contingency table)
table(gender, education)
```

This would give a 2x2 table showing the cross-tabulation of gender vs. education.

Key Points:

- **Factors** store categorical data and represent both the values and their levels.
- **Tables** summarize the frequency distribution of categorical data, helping in visualizing the count of occurrences.

Common Functions used with factors in R

When working with factors in R, there are several common functions you can use to manage, analyze, and modify categorical data. Here's a list of useful functions for factors and what they do:

1. factor()

The primary function to create a factor from a vector or other data type.

```
r
Copy
# Create a factor from a character vector
gender <- factor(c("Male", "Female", "Female", "Male"))
```

2. levels()

Returns the levels of a factor, i.e., the distinct categories or values that the factor can take.

r
Copy
levels(gender) # Returns "Male" "Female"

You can also set the levels of a factor using the levels() function:

r

Copy
levels(gender) <- c("Male", "Female")

3. nlevels()

Returns the number of levels in a factor.

r
Copy
nlevels(gender) # Returns 2

4. summary()

Provides a summary of a factor or a table, showing the frequency of each level.

r
Copy
summary(gender)

This will give you the count of each level:

markdown
Copy

```
 Female   Male
     2      2
```

5. table()

Creates a frequency table showing the count of each level in a factor or vector.

r
Copy
table(gender)

This will return the count of each category:

markdown

```
Copy
 Female   Male
     2      2
```

You can also use table() to create cross-tabulations (contingency tables) with two or more factors:

```
r
Copy
education <- factor(c("High School", "College", "College", "High School"))
table(gender, education)
```

6. as.factor()

Converts an object (such as a character vector or numeric vector) into a factor.

```
r


age_group <- c("Young", "Middle-aged", "Old", "Young", "Middle-aged")
age_factor <- as.factor(age_group)
```

7. as.character()

Converts a factor back to a character vector.

```
r
Copy
gender_char <- as.character(gender)
```

8. relevel()

Changes the reference level of a factor. This is especially useful in statistical modeling when you want to set a specific reference category.

```
r
Copy
education <- factor(c("High School", "College", "College", "High School"))
education <- relevel(education, ref = "College")  # Set "College" as the reference level
```

9. levels<- (assignment operator)

This allows you to directly modify the levels of a factor.

```
r
Copy
# Change levels of a factor
levels(gender) <- c("Male", "Female", "Other")
```

10. droplevels()

Drops unused levels from a factor. It’s useful when you subset a factor, and there are levels that are no longer present.

```
r
Copy
# Remove unused levels from a factor
gender_sub <- gender[gender == "Female"]
gender_sub <- droplevels(gender_sub)
```

11. contrasts()

Returns or sets the contrasts used in a factor for modeling purposes. This is more relevant for advanced statistical analysis and modeling.

```
r
Copy
contrasts(gender)  # Show contrasts for the 'gender' factor
```

12. levels() vs **labels()**

- levels() gives the raw level names.
- labels() can be used for labeled factors (especially useful with ordered factors).

```
r
Copy
# Ordered factor example
education_ordered <- factor(c("High School", "College", "Graduate"),
                        levels = c("High School", "College", "Graduate"),

    ordered = TRUE)
levels(education_ordered)
```

13. ordered()

Creates an ordered factor, which is useful when the categories have a natural ordering (e.g., "Low", "Medium", "High").

```
r
Copy
satisfaction <- factor(c("Low", "High", "Medium", "Medium", "High"),
                     levels = c("Low", "Medium", "High"), ordered = TRUE)
```

14. addNA()

This function is used to add a NA level to a factor.

```
r
Copy
gender_with_na <- addNA(gender)
```

15. factor() with **exclude** argument

You can use the exclude argument to specify certain levels that should be excluded from the factor.

r
Copy
```
# Exclude a specific value (e.g., "Male") from the factor
gender <- factor(c("Male", "Female", "Female", "Male"), exclude = "Male")
```

Example of Working with Factors

Here’s a practical example that demonstrates how these functions might be used:

r
Copy
```
# Step 1: Create a factor
gender <- factor(c("Male", "Female", "Female", "Male"))

# Step 2: Display levels
print(levels(gender))  # "Male" "Female"

# Step 3: Count levels
print(nlevels(gender))  # 2

# Step 4: Convert factor to character
gender_char <- as.character(gender)
print(gender_char)

# Step 5: Relevel the factor (change reference level)

gender <- relevel(gender, ref = "Female")
print(gender)

# Step 6: Create a frequency table
print(table(gender))

# Step 7: Drop unused levels

gender_sub <- gender[gender == "Female"]
gender_sub <- droplevels(gender_sub)
print(gender_sub)
```

Conclusion

These functions are foundational when working with factors in R. Whether you are preparing data for statistical modeling or simply summarizing categorical data, knowing how to manipulate and summarize factors will make your analysis much easier.

Working with tables

In R, **tables** are a fundamental way to summarize and analyze categorical data. A **table** is an object that represents the frequency distribution of a factor or vector. Tables are created using the table() function, and there are several other functions and methods for manipulating and working with tables. Here's a guide on how to work with tables in R.

1. Creating a Table using **table()**

The table() function creates a table (also called a frequency table or contingency table) by counting the number of occurrences of each unique value in a vector or factor.

Example 1: Table for a Single Factor

```
r
Copy
# Create a vector with categorical data
gender <- c("Male", "Female", "Female", "Male", "Female", "Male")

# Create a frequency table
gender_table <- table(gender)
print(gender_table)
```

Output:

```
markdown
Copy
gender
Female   Male
     3      3
```

This table tells you how many "Female" and "Male" values are in the gender vector.

Example 2: Table for Multiple Factors (Cross-Tabulation)

You can use table() with two or more vectors (factors) to create a cross-tabulation (contingency table). This shows how the categories of two factors are related.

```
r
Copy
# Create another factor (education)
education <- c("College", "High School", "College", "High School", "College", "High School")

# Create a cross-tabulation between gender and education
gender_education_table <- table(gender, education)
print(gender_education_table)
```

Output:

```
markdown

Copy
        education
gender    College High School
  Female        2           1
  Male          1           2
```

This table shows the count of females and males in each education category.

2. Accessing Elements of a Table

Once a table is created, you can access specific elements or perform operations on it.

Access a specific cell in the table

To access a specific count from a table (e.g., the count of "Female" and "College"), use the [] indexing.

```
r
Copy
# Access the count of "Female" and "College"
gender_education_table["Female", "College"]  # Returns 2
```

Convert a Table to a Data Frame

You can convert a table to a data frame for easier manipulation or to work with other R functions that require data frames.

```
r
Copy
# Convert the table to a data frame
gender_education_df <- as.data.frame(gender_education_table)
print(gender_education_df)
```

Output:

```
nginx
Copy
  gender    education Freq
1 Female      College    2
2 Female High School    1
3   Male      College    1
4   Male High School    2
```

3. Using **addmargins()**

The addmargins() function is used to add margin totals (row sums and column sums) to the table. This is useful for getting the total counts for each row and column.

```
r
Copy
# Add margin totals (row sums and column sums)
gender_education_with_margins <- addmargins(gender_education_table)
print(gender_education_with_margins)
```

Output:

```
mathematica
Copy
        education
gender    College High School Sum
  Female        2           1   3
  Male          1           2   3
  Sum           3           3   6
```

4. Using **prop.table()** for Proportions

The prop.table() function calculates the proportions (relative frequencies) of each cell in the table, based on the total sum or along the rows/columns.

Example: Proportions across the entire table

r
Copy
```
# Proportions across the entire table
gender_education_proportions <- prop.table(gender_education_table)
print(gender_education_proportions)
```

Output:

markdown
Copy
```
        education
gender    College High School
  Female 0.3333333 0.1666667
  Male   0.1666667 0.3333333
```

This shows the proportion of each combination of gender and education category.

Example: Proportions by rows

You can also calculate row-wise proportions by passing the margin = 1 argument to prop.table(), which normalizes the table by rows.

r
Copy
```
# Proportions by rows
gender_education_row_proportions <- prop.table(gender_education_table, margin = 1)

print(gender_education_row_proportions)
```

Output:

markdown
Copy
```
        education
gender    College High School
  Female 0.6666667 0.3333333
  Male   0.3333333 0.6666667
```

This normalizes the table so that the proportions in each row sum to 1.

5. Using **ftable()** for Flat Tables

The ftable() function can be used to create flat tables (fancy tables) that make it easier to view multi-dimensional tables in a compact format.

r
Copy

# Create a flat table (for better visualization of a multi-dimensional table)
gender_education_flat <- ftable(gender, education)
print(gender_education_flat)
Output:
markdown
Copy

```
        education
gender   College High School
  Female       2         1
  Male         1         2
```

6. Table Manipulation with **dplyr**

The dplyr package allows for easy manipulation of tables as well, especially when you convert tables to data frames.

Example: Using dplyr to summarize a table
r
Copy

```
library(dplyr)

# Convert table to data frame
gender_education_df <- as.data.frame(gender_education_table)

# Summarize data using dplyr
gender_education_df %>%
  group_by(gender) %>%
  summarize(total = sum(Freq))
```

Output:

csharp
Copy

```
# A tibble: 2 × 2
  gender total
  <fct>  <int>
1 Female     3
2 Male       3
```

7. **Handling Missing Data**

If your data has NA values, table() will count them as a separate level. You can handle missing values by using exclude to remove NA values from the table.

r
Copy

```
# Create a vector with NAs
data_with_na <- c("Male", "Female", "Male", NA, "Female")

# Create a table excluding NAs
table_without_na <- table(data_with_na, exclude = NA)
print(table_without_na)
```

Output:

```
markdown
Copy
data_with_na
Female   Male
     2      2
```

Summary of Common Table Functions:

- table(): Create a frequency or contingency table.
- addmargins(): Add margin totals to a table (row/column sums).
- prop.table(): Calculate proportions based on the table.
- ftable(): Convert a multi-dimensional table into a flat, easy-to-read format.
- as.data.frame(): Convert a table into a data frame for further manipulation.

- summary(): Provides a summary of the table.
- exclude **in** table(): Exclude NA values or specific values from a table.

Conclusion:

Tables are a powerful tool in R for summarizing categorical data. They allow you to easily calculate frequencies, proportions, and cross-tabulations. By using functions like table(), addmargins(), and prop.table(), you can explore your data and gain meaningful insights.

Other factors and table related functions

In R, tables and factors are essential for data manipulation and statistical analysis. Here's an overview of some common functions related to factors and tables, as well as other operations you might use:

Factors in R

Factors are used to represent categorical variables. They store both the values of a categorical variable and the set of possible levels.

Key functions for working with factors:

1. factor(): Creates a factor from a vector.

   ```
   R

   Copy
   x <- c("low", "medium", "high", "medium")
   f <- factor(x)
   print(f)
   ```

2. levels(): Returns or sets the levels of a factor.

   ```
   R
   Copy
   levels(f)
   ```

3. table(): Creates a frequency table of a factor or vector.

   R
   Copy

   ```
   table(f)
   ```

4. as.factor(): Coerces a vector into a factor.

   R
   Copy
   ```
   x <- c("red", "blue", "blue", "green")
   f <- as.factor(x)
   ```

5. nlevels(): Returns the number of levels of a factor.

   R
   Copy
   ```
   nlevels(f)
   ```

6. relevel(): Reorders the levels of a factor, making the specified level the reference.

   R
   Copy
   ```
   f <- relevel(f, ref = "high")
   ```

7. levels<-: Modify the levels of a factor.

   R
   Copy
   ```
   levels(f) <- c("low", "medium", "high")
   ```

8. ordered(): Converts a factor into an ordered factor.

   R
   Copy
   ```
   f_ordered <- factor(x, ordered = TRUE, levels = c("low", "medium", "high"))
   ```

Working with Tables in R

Tables are a great way to summarize categorical data, especially with the table() function.

Key table functions:

table(): Creates a contingency table from a vector, matrix, or data frame.

   R
   Copy

```
data <- c("apple", "banana", "apple", "banana", "orange")
table(data)
```

1. addmargins(): Adds margin sums (like row and column totals) to a table.

   R
   Copy
   ```
   tab <- table(c("A", "A", "B", "B", "C"))
   addmargins(tab)
   ```

2. prop.table(): Converts a table to proportions (relative frequencies).

   R

   Copy
   ```
   prop.table(table(data))
   ```

3. ftable(): Creates a flat contingency table (more readable than multi-dimensional arrays).

   R
   Copy
   ```
   ftable(mtcars$mpg, mtcars$cyl)
   ```

4. table() **with** useNA **parameter**: Can be used to include or exclude missing values (NA).

   R
   Copy
   ```
   data <- c("apple", "banana", NA, "orange", "banana", NA)
   table(data, useNA = "ifany")
   ```

Other Useful Table-Related Functions

1. xtabs(): Creates contingency tables from a formula and data frame.

   R
   Copy
   ```
   data(mtcars)
   xtabs(~ cyl + gear, data = mtcars)
   ```

2. dplyr::count(): A dplyr function that counts occurrences of distinct combinations of factor levels.

   R
   Copy
   ```
   library(dplyr)
   count(mtcars, cyl, gear)
   ```

3. summary(): Provides a summary for factors, vectors, and tables.

R
Copy
```
summary(f)
summary(mtcars)
```

4. aggregate(): Computes summary statistics by group.

   R
   Copy
   ```
   aggregate(mpg ~ cyl, data = mtcars, FUN = mean)
   ```

5. melt() **and** dcast() **from the** reshape2 **package**: Used to reshape data (wide to long format and vice versa).

   R
   Copy
   ```
   library(reshape2)
   molten_data <- melt(mtcars)
   dcast(molten_data, variable ~ value)
   ```

Example: Analyzing Categorical Data with Tables

Let's say you have the following data about fruit preference among a group of people:

R
Copy
```
# Example data
fruit <- c("apple", "banana", "orange", "banana", "apple", "apple")
```

1. **Create a table of counts**:

   R
   Copy
   ```
   fruit_table <- table(fruit)

   print(fruit_table)
   ```

2. **Proportions**:

   R
   Copy
   ```
   prop.table(fruit_table)
   ```

3. **Add margins (totals)**:

   R
   Copy
   ```
   addmargins(fruit_table)
   ```

Conclusion

These functions allow for flexible handling of categorical data and summarization in R. Understanding and manipulating factors and tables effectively helps in preparing data for analysis, performing statistical tests, and interpreting results.

CONTROL STATEMENTS

In R, control statements are used to control the flow of execution based on certain conditions. The basic control structures in R are if, else, ifelse, for, while, and repeat loops. These structures allow you to make decisions and repeat actions in your programs.

Here’s an overview of key control statements in R:

1. if Statement

The if statement allows you to execute code only when a condition is true.

R
Copy
```
x <- 10
if (x > 5) {
  print("x is greater than 5")
}
```

In this example, the code inside the if block will execute because x > 5.

2. else Statement

The else statement follows an if and is executed when the if condition is **false**.

R
Copy
```
x <- 3
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

Here, the code inside the else block runs because x is not greater than 5.

3. else if Statement

If you have multiple conditions to check, you can chain else if statements after an initial if. Only one block will execute based on the conditions.

R
Copy
```
x <- 7
if (x > 10) {
  print("x is greater than 10")
} else if (x > 5) {
  print("x is greater than 5 but less than or equal to 10")


} else {
  print("x is less than or equal to 5")
}
```

In this case, the else if condition is true, so it prints "x is greater than 5 but less than or equal to 10".

4. ifelse() Function

The ifelse() function is a vectorized version of the if statement. It is used to test a condition and returns one value if true, another value if false.

R
Copy
```
x <- 10
result <- ifelse(x > 5, "Greater than 5", "Less than or equal to 5")
print(result)
```

This will return "Greater than 5" because x is greater than 5.

5. for Loop

The for loop is used to iterate over a sequence (such as a vector, list, or range of numbers).

R
Copy
```
for (i in 1:5) {
  print(paste("Iteration", i))
}
```

This loop prints the statement "Iteration 1", "Iteration 2", and so on, until i reaches 5.

6. while Loop

The while loop continues to execute as long as a condition is TRUE. Be careful not to create infinite loops!

R

Copy

```
x <- 1
while (x <= 5) {
  print(paste("x is", x))

  x <- x + 1
}
```

This will print values of x from 1 to 5.

7. repeat Loop

The repeat loop is similar to a while loop but does not evaluate the condition until the end of each iteration. It is typically used with a break statement to stop the loop.

R

Copy

```
x <- 1
repeat {
  print(paste("x is", x))
  x <- x + 1
  if (x > 5) {
    break
  }
}
```

This will print values of x from 1 to 5, and the loop stops once x exceeds 5 due to the break statement.

8. break Statement

The break statement is used to exit a loop prematurely, usually when a certain condition is met.

R

Copy

```
for (i in 1:10) {

if (i == 6) {
    break
  }
  print(i)
}
```

This will print numbers from 1 to 5 and then break out of the loop when i is 6.

9. next Statement

The next statement is used to skip the current iteration of a loop and move on to the next one.

R

Copy
```
for (i in 1:5) {
  if (i == 3) {

    next
  }
  print(i)
}
```

This will print 1, 2, 4, and 5, skipping the iteration where i is 3.

Example: Combining Control Statements Here's an example that combines several control statements:

R
Copy
```
x <- 10

# Using if-else to check a condition
if (x > 5) {
  print("x is greater than 5")

  # Using a for loop to print numbers


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

    # Break if i equals 2
    if (i == 2) {
      break
    }
  }
} else {
  print("x is less than or equal to 5")
}
```

In this example, the program checks if x > 5, prints numbers using a for loop, and breaks out of the loop when i is 2.

Summary of Control Structures:

- if: Conditionally execute a block of code.
- else: Execute code when the if condition is false.
- else if: Chain multiple conditions.
- ifelse(): Vectorized if-else logic.
- for **loop**: Iterate over a sequence.
- while **loop**: Loop while a condition is true.
- repeat **loop**: Loop until manually interrupted with break.

- break: Exit a loop.

- next: Skip to the next iteration in a loop.These control structures help to make your R programs flexible and dynamic.

UNIT IV: Classes in R

S3 Classes – S4 Classes – Managing your objects – Input/output – accessing keyboard and monitor – reading and writing files – accessing the internet – String Manipulation – Graphics – Creating Graphs– Customizing Graphs – Saving Graphs to files – Creating Three-Dimensional plots

This unit focuses on the concept of classes, object management, input and output handling, string manipulation, graphics, and plotting in R. R is not just a statistical computing tool; it is also a functional and object-oriented programming language. Understanding classes and data structures is key to organizing and managing code efficiently.

S3 CLASSES

S3 classes are the simplest and most common form of object-oriented system in R. They provide a flexible way to define how functions behave for different kinds of objects. S3 is an informal class system that relies on naming conventions rather than formal definitions.

Defining S3 Classes:
An S3 object is typically created by assigning a class attribute to a list or vector.
For example:

```
person <- list(name="John", age=25)

 class(person) <- "Person"
```

In this example, an object “person” is given a class called “Person”. Methods can then be defined to behave differently for objects of class “Person”.

Creating Methods for S3:
S3 methods are functions that end with the class name.
For example, to define a print method for the “Person” class:

```
   print.Person <- function(x) {
       cat("Name:", x$name, "
Age:", x$age, "
")
   }
```

When print(person) is called, R automatically looks for a function named print.Person().

Advantages of S3 Classes:
- Simple and easy to use.
- Flexible: no need for formal definitions.
- Works well for small projects or prototypes.

Limitations:
- No strict structure enforcement.
- Errors can occur if conventions are not followed.

S4 CLASSES

S4 classes were introduced to add more formalism and reliability to object-oriented programming in R. Unlike S3, S4 classes require explicit class and method definitions.

Defining S4 Classes:
S4 classes are created using setClass():
```
    setClass("Person", slots = list(name="character", age="numeric"))
```

Creating Objects:
```
    p1 <- new("Person", name="John", age=25)
```

Accessing Slots:
Slots in an S4 object can be accessed using the @ operator:
```
    p1@name
```

Defining Methods for S4:
Methods are defined using setMethod():
```
    setMethod("show", "Person", function(object) {
        cat("Name:", object@name, "
Age:", object@age, "
")
    })
```

Advantages of S4 Classes:
- Enforces structure and type safety.
- Methods are clearly defined.
- Suitable for large and complex applications.

Limitations:
- More verbose and complex than S3.

MANAGING OBJECTS
Objects are central to R programming. They store data and functions. The environment stores all objects currently in use.

Key Functions:
- ls(): Lists all objects in the current environment.
- rm(): Removes objects.
- exists(): Checks if an object exists.
- assign(): Assigns values to variable names dynamically.
- get(): Retrieves objects by name.

Example:
```
    x <- 10
```

```
assign("y", 20)
ls()
rm(x)
```

INPUT AND OUTPUT
Input and output operations are essential for interacting with users and external data.

Reading Input from Keyboard:
The readline() function allows user input:

```
name <- readline(prompt="Enter your name: ")
```

Writing Output to Monitor:
The cat() and print() functions display information on screen:

```
cat("Welcome", name)
```

READING AND WRITING FILES
R provides functions to read and write data from text, CSV, and other files.

Common Functions:
- read.table(), read.csv() – for reading data.
- write.table(), write.csv() – for writing data.

Example:

```
data <- read.csv("data.csv")
write.csv(data, "output.csv")
```

ACCESSING THE INTERNET
R can interact with web resources using packages like httr or RCurl.

Example:

```
library(httr)
response <- GET("https://example.com")
 content <- content(response, "text")
```

STRING MANIPULATION
Strings are sequences of characters used for text data.

Common Functions:
- nchar(): Number of characters.
- substr(): Extract part of a string.
- paste(): Combine strings.
- strsplit(): Split strings.

Example:

```
s <- "Data Science with R"
substr(s, 1, 4)
```

GRAPHICS IN R

R is known for its powerful graphical capabilities. The base graphics system allows creating a variety of plots.

Creating Graphs:

```
x <- 1:10
y <- x^2
plot(x, y, main="Simple Plot", xlab="X", ylab="Y")
```

CUSTOMIZING GRAPHS
Customization allows better visualization of data.

Options include:
- main: Title of the graph.
- xlab, ylab: Axis labels.
- col: Color of points or lines.
- type: Type of plot (p for points, l for lines).

Example:

```
plot(x, y, type="b", col="blue", main="Customized Plot")
```

SAVING GRAPHS TO FILES
Graphs can be saved to various file formats.

Example:

```
 png("graph.png")
 plot(x, y)
dev.off()
```

Other formats include pdf(), jpeg(), and tiff().

CREATING THREE-DIMENSIONAL PLOTS
3D plots help visualize multivariate data. Packages like plotly or rgl are used.

Example using plotly:

```
library(plotly)
plot_ly(x=~x, y=~y, z=~(x+y), type="scatter3d", mode="markers")
```

3D plots provide interactive visualization that enhances data understanding.

CONCLUSION
This unit explained how R handles object-oriented programming with S3 and S4 classes, along with object management, input/output operations, file handling, internet access, string manipulation, and graphics. Mastering these concepts helps in developing structured, interactive, and visually appealing R programs.

UNIT V: Modelling in R

Interfacing R to other languages – Parallel R – Basic Statistics – Linear Model – Generalized Linear models – Non-linear Models – Time Series and Auto-Correlation – Clustering.

This unit focuses on different modeling techniques in R. It explores how R connects to other languages, performs parallel computation, and applies statistical and predictive models such as linear, generalized linear, and non-linear models. Additionally, it explains time series analysis, auto-correlation, and clustering methods.

INTERFACING R TO OTHER LANGUAGES

R is highly flexible and can interface with other programming languages such as C, C++, Java, and Python. This feature allows R programmers to extend its capabilities and improve performance when handling complex or time-consuming computations.

Interfacing with C/C++:
R provides .C() and .Call() functions to call C routines from R code. This enables developers to write performance-intensive tasks in C/C++ and integrate them with R.

Example:

```
result <- .C("myCFunction", as.integer(x), as.double(y))
```

Rcpp package simplifies this process by allowing C++ functions to be written and called directly in R without complex syntax.

Interfacing with Python:
The reticulate package provides a bridge between R and Python. It allows R users to import Python modules, run Python scripts, and exchange data between the two languages.

Example:

```
library(reticulate)
py_run_string("print('Hello from Python!')")
```

Advantages:
- Reuse existing code written in other languages
- Improve performance for computationally heavy tasks.
- Integrate different programming environments.

PARALLEL R

Parallel computing allows R to execute multiple operations simultaneously, reducing execution time for large datasets or intensive computations.

Base R provides parallel capabilities through the 'parallel' package. It includes functions such as mclapply(), parLapply(), and clusterApply().

Example:

```
library(parallel)
cl <- makeCluster(4)
parLapply(cl, 1:10, function(x) x^2)
```

```
stopCluster(cl)
```

The foreach and doParallel packages also support parallel loops, allowing tasks to run concurrently on multiple cores or machines.

Benefits of Parallel R:
- Improved computational efficiency.
- Handles large-scale data analysis.
- Utilizes multi-core processors effectively.

BASIC STATISTICS IN R

R is primarily a statistical computing environment. It offers tools for descriptive and inferential statistics, hypothesis testing, and data summarization.

Common Statistical Functions:
- mean(), median(), mode() – measures of central tendency.
- var(), sd(), range() – measures of dispersion.
- cor(), cov() – relationships between variables.

Example:
```
x <- c(5, 10, 15, 20)
mean(x)
sd(x)
cor(x, x^2)
```

Hypothesis Testing:
R supports various statistical tests such as t-test, chi-square test, and ANOVA.

Example:
```
t.test(x, y)
chisq.test(table(data$group, data$outcome))
```

LINEAR MODEL

A linear model describes a relationship between a dependent variable and one or moreindependent variables using a straight-line equation.

The lm() function in R is used to fit linear models.

Example:
```
model <- lm(y ~ x1 + x2, data=dataframe)
summary(model)
```

Interpretation:
- Coefficients represent the effect of predictors.
- The R-squared value indicates model fit.
- Residuals show differences between observed and predicted values.

Advantages:
- Simple and interpretable.
- Useful for prediction and trend analysis.

GENERALIZED LINEAR MODELS (GLM)
GLMs extend linear models by allowing response variables to have error distributions other than normal. Common GLMs include logistic regression and Poisson regression.

Example of Logistic Regression:

```
glm_model <- glm(y ~ x1 + x2, family=binomial, data=dataframe)
summary(glm_model)
```

Example of Poisson Regression:

```
glm_poisson <- glm(count ~ x1 + x2, family=poisson, data=dataframe)
```

Components of GLM:
- Random component: Specifies the probability distribution.
- Systematic component: Defines predictors.
- Link function: Connects the linear predictor and mean of the distribution.

Advantages:
- Handles binary, count, and non-normal data.
- Flexible and widely applicable in real-world modeling.

NON-LINEAR MODELS
Non-linear models are used when the relationship between variables cannot be represented by a straight line. R provides the nls() function for fitting non-linear models.

Example:

```
model <- nls(y ~ a * exp(b * x), data=dataframe, start=list(a=1, b=0.1))
```

Features:
- Captures complex relationships.
- Useful in growth curves, enzyme kinetics, and population studies.

Challenges:
- Requires good starting values.
- Can converge slowly or fail with poor initialization.

TIME SERIES AND AUTO-CORRELATION
A time series is a sequence of data points recorded at successive time intervals. R provides powerful tools for analyzing and forecasting time series data.

Creating a Time Series Object:

```
ts_data <- ts(data, start=c(2020,1), frequency=12)
```

Plotting and Analyzing:

```
plot(ts_data)
```

Decomposition:
Time series can be decomposed into trend, seasonal, and irregular components using the decompose() function.

Auto-correlation:
Auto-correlation measures how observations relate to past values in the series.

Example:
```
acf(ts_data)
pacf(ts_data)
```

Forecasting Models:
- ARIMA (Auto-Regressive Integrated Moving Average)
- Exponential Smoothing

Example:
```
library(forecast)
fit <- auto.arima(ts_data)
forecast(fit, h=12)
```

Advantages of Time Series Modeling:
- Identifies trends and seasonal patterns.
- Useful for forecasting and prediction.

CLUSTERING
Clustering is an unsupervised learning technique that groups similar observations based on their characteristics. It helps identify natural groupings in data.

Common Clustering Methods:
1. K-Means Clustering
2. Hierarchical Clustering
3. DBSCAN (Density-Based Clustering)

Example (K-Means):
```
set.seed(123)
data <- matrix(rnorm(100), ncol=2)
kmeans_result <- kmeans(data, centers=3)
plot(data, col=kmeans_result$cluster)
```

Hierarchical Clustering:
```
dist_matrix <- dist(data)
hc <- hclust(dist_matrix, method="complete")
plot(hc)
```

Evaluation of Clustering:
- Within-cluster sum of squares (WSS)
- Silhouette score

Applications:
- Market segmentation
- Image recognition

- Bioinformatics

CONCLUSION

Modeling in R is a cornerstone of data analysis and predictive analytics. From statistical summaries to complex machine learning models, R provides a wide range of tools for modeling, forecasting, and pattern recognition. Understanding how to interface R with other languages, use parallel computation, and apply linear, generalized, and non-linear models allows analysts to build efficient and scalable data-driven solutions. Time series analysis and clustering further enhance the ability to understand data trends and relationships effectively.