Unit: I

Introduction of the language, numeric, arithmetic, assignment, and vectors, Matrices and Arrays, Non-numeric Values, Lists and Data Frames, Special Values, Classes, and Coercion, Basic Plotting.

Introduction of the language

What is R

- ✓ R is a programming language and environment specifically designed for statistical computing and data analysis. It is widely used by statisticians, data scientists, and researchers for a variety of tasks, including:

Features of R

1. It is a simple and effective programming language which has been well developed.
2. It is a well-designed, easy, and effective language which has the concepts of user-defined, looping, conditional, and various I/O facilities.
3. For different types of calculation on arrays, lists and vectors, R contains a suite of operators.
4. It provides effective data handling and storage facility.
5. It is an open-source, powerful, and highly extensible software.
6. It provides highly extensible graphical techniques.
7. R is an interpreted language.

Application of 'R' Programming

- ✓ Fintech Companies (financial services)
- ✓ Academic Research
- ✓ Government (FDA, National Weather Service)
- ✓ Retail
- ✓ Social Media
- ✓ Data Journalism
- ✓ Manufacturing
- ✓ Healthcare

R Advantages

1. **Excellent for Statistical Computing and Analysis**
   - ✓ R is a statistical language created by statisticians. R is the **most used** programming language for developing statistical tools.
2. **Open-source**
   - ✓ R is an open-source programming language. Anyone can work with R without any license or fee.
3. **A Large Variety of Libraries**
   - ✓ R's massive community support has resulted in a very large collection of libraries. R is famous for its graphical libraries These libraries support and enhance the R development environment. R has libraries with a huge variety of applications.
4. **Cross platform support**
   - ✓ R is **machine-independent**. It supports the cross-platform operation. Thus, it is usable on many different operating systems.
5. **Specialized for Data Analysis**

   Statistical Analysis: R was designed specifically for statistics and data analysis, making it powerful for a wide range of statistical techniques, such as regression, classification, hypothesis testing, time series analysis, and more.

6. **Can do Data Cleansing, Data Wrangling, and Web Scraping**
   - ✓ R can collect data from the internet through web scraping and other means. It can also perform data cleansing.
   - ✓ Data cleansing is the process of detecting and removing/correcting inaccurate or corrupt records. R is also useful for data wrangling which is the process of converting raw data into the desired format for easier consumption.
7. **Powerful Graphics**
   - ✓ R has extensive libraries that can produce production **quality graphs** and visualizations. These graphics can be of static as well as dynamic nature.
8. **Can Interact with Databases**
   - ✓ R contains several packages that enable it to interact with databases. Some of these packages are Roracle, Open Database Connectivity Protocol), RmySQL, etc
9. **Data Visualization**
   - ✓ **Powerful Plotting Tools**: R is renowned for its data visualization capabilities.

Disadvantages of R Programming

1. Steep Learning Curve

- ✓ **Difficult for Beginners**: R's syntax and structure can be challenging for beginners, particularly those with no prior programming experience. It requires an understanding of statistics and mathematical concepts to use effectively, which can make it less approachable compared to languages like Python or even Excel.

2. Some Packages may be of Poor Quality

- ✓ CRAN houses more than **10,000 libraries** and packages. Some of them are redundant as well.Due to the large quality,some of the packages may be of poor quality.

3. Poor Memory Management

- ✓ **Not Optimized for Memory Efficiency**: R often holds entire datasets in memory (RAM), which can be inefficient when working with very large datasets. This can lead to memory overloads or performance bottlenecks, especially for data sets that don't fit entirely in memory

4. Slow Speed

- ✓ **Slower Execution**: R can be slower compared to compiled languages like C, C++, or Java, especially when performing computationally intensive tasks

5. Poor Security

- ✓ R lacks basic security measures. So, making web-apps with it is not always safe.

6. No Dedicated Support Team

- ✓ R has no dedicated support team to help a user with their issues and problems. But thecommunity is quite large, so everybody helps each other out.

7. Limited Object-Oriented Programming (OOP) Support

- ✓ R supports object-oriented programming through systems like S3 and S4, but it doesn't have as sophisticated and standardized an OOP framework as languages like Python or Java

8. Poor Support for Mobile and Web Applications

Mobile Application Development: R is not well-suited for mobile application development (iOS, Android).

- ✓ **Web Application Limitations**: Although R provides packages like shiny for creating web applications, these tools are often not as flexible, scalable, or feature-rich as JavaScript-based frameworks

Data Types in R Programming

- ✓ In R, data types are essential for managing and analyzing data. R has several fundamental data types, each serving different purposes. Here's an overview of the main data types in R

1. **Numeric**

- ✓ The most common data type in R, used for continuous numbers. It includes both integers and real (decimal) numbers.
- ✓ Represents numbers (both integers and decimals).
- ✓ **Example**: x <- 5, y <- 3.1
- ✓ When you create a variable without specifying a type, R will default to numeric

2. **Integer**

- ✓ Specifically represents integer values. You can specify an integer by appending an L to the number.
- ✓ **Example**: x <- 5L
- ✓ Unlike other languages, R does not automatically treat whole numbers as integers unless explicitly specified with the L suffix

3. **Complex**

- ✓ Represents complex numbers, which have real and imaginary parts.
- ✓ **Example**: z <- 3 + 2i

4. **Logical**

- ✓ Represents boolean values: TRUE or FALSE.
- ✓ **Example**: is_greater <- 5 > 3 (returns TRUE)

5. **Character**

- ✓ Represents text strings or character data.
- ✓ **Example**: name <- "R programming"

6. Raw

- ✓ Represents raw bytes. This is less commonly used but can be useful for specific applications.
- ✓ **Example**: raw_data <- charToRaw("hello")

Variables in R Programming

- ✓ Variables are used to store the information to be manipulated and referenced in the R program.
- ✓ The R variable can store an atomic vector, a group of atomic vectors, or a combination of many R objects.

R supports three ways of variable assignment:

- ✓ Using the Assignment Operator (<-)
- ✓ Using the Equal Sign **(=)**
- ✓ Using the Assignment Operator (->)

Using the Assignment Operator (<-)

- ✓ The most common way to assign a value to a variable is using the <- operator:

Ex:

```
# Assigning a numeric value
x <- 10
```

Using the Equal Sign (=)

- ✓ You can also use the equal sign (=) for assignment, but using <- is generally preferred in R for clarity:

Example

```
# Assigning a numeric value
x =10
```

Using the Assignment Operator (->)

- ✓ The most common way to assign a value to a variable is using the <- operator:

Ex:

```
# Assigning a numeric value
10->x
```

The following rules need to be kept in mind while naming a R variable:

- ✓ A valid variable name consists of a combination of alphabets, numbers, dot(.), and underscore (_) characters. Example: var.1_ is valid
- ✓ Apart from the dot and underscore operators, no other special character is allowed. Example: var$1 or var#1 both are invalid
- ✓ Variables can start with alphabets or dot characters. Example: .var or var is valid
- ✓ The variable should not start with numbers or underscore. Example: 2var or _var is invalid.
- ✓ If a variable starts with a dot the next thing after the dot cannot be a number. Example: .3var is invalid

- ✓ The variable name should not be a reserved keyword in R. Example: TRUE, FALSE,etc.

Examples of valid variable names:

- ✓ my_var
- ✓ var1
- ✓ .dataFrame

Examples of invalid variable names:

- ✓ 1stVariable (starts with a number)
- ✓ my var (contains a space)

class () function:

- ✓ This built-in function is used to determine the data type of the variable provided to it. The R variable to be checked is passed to this as an argument and it prints the data type in return.

Syntax: class(variable)

Example:
var1 = "hello"
print(class(var1)) Output: “character”

Operators in R

- ✓ In R, operators are special symbols that perform operations on variables and values. They can be classified into several categories based on their functionality.

There are the following types of operators used in R:

- ✓ Arithmetic Operators
- ✓ Relational Operators
- ✓ Logical Operators
- ✓ Assignment Operators
- ✓ Miscellaneous Operators

Arithmetic Operators

- ✓ These operators perform basic mathematical calculations.

S. No	Operator	Description	Example

1.	+	This operator isused to add twovectors in R.	a <- c(2, 3.3, 4)
			b <- c(11, 5, 3)
			print(a+b)
			
			It will give us the following output:

			[1] 13.0 8.3 5.0

2.	-	This operator isused to divide a vector from another one.	a <- c(2, 3.3, 4)
			b <- c(11, 5, 3)
			print(a-b)
			
			It will give us the following output:
			[1] -9.0 -1.7 3.0

3.	*	This operator isused to multiply two vectors witheach other.	a <- c(2, 3.3, 4)
			b <- c(11, 5, 3)
			print(a*b)
			
			It will give us the following output:
			[1] 22.0 16.5 4.0

4.	/	This operator divides the vector from another one.	a <- c(2, 3.3, 4)
			b <- c(11, 5, 3)
			print(a/b)
			It will give us the following output:
			[1] 0.1818182 0.6600000 4.0000000

5.	%%	This operator isused to find theremainder of the first vector with the second vector.	a <- c(2, 3.3, 4)
			b <- c(11, 5, 3)
			print(a%%b)
			
			It will give us the following output:
			[1] 2.0 3.3 0

6.	%/%	This operator is used to find the division of the first vector with the second(quotient).	a <- c(2, 3.3, 4)
			b <- c(11, 5, 3)
			print(a%/%b)
			
			It will give us the following output:
			[1] 0 0 4

7.	^	This operator raised the first	a <- c(2, 3.3, 4)
			b <- c(11, 5, 3)

		vector to the exponent of the second vector.	print(a^b) **It will give us the following output:** [1] 0248.0000 391.3539 4.0000

Relational Operators

✓ These operators are used to compare two values or expressions.

S. No	Operator	Description	Example

1.	>	This operator will return TRUE when every element inthe first vector is greater thanthe corresponding element ofthe second vector.	a <- c(1, 3, 5)
			b <- c(2, 4, 6)
			print(a>b)
			
			It will give us the following output:
			[1] FALSE FALSE FALSE

2.	<	This operator will return TRUE when every element inthe first vector is less than thecorresponding element of the second vector.	a <- c(1, 9, 5)
			b <- c(2, 4, 6)
			print(a<b)
			
			It will give us the following output:
			[1] FALSE TRUE FALSE

3.	<=	This operator will return TRUE when every element inthe first vector is less than or equal to the corresponding element of another vector.	a <- c(1, 3, 5)
			b <- c(2, 3, 6)
			print(a<=b)
			
			It will give us the following output:
			[1] TRUE TRUE TRUE

4.	>=	This operator will return TRUE when every element in the first vector is greater than or equal to the corresponding element of another vector.	a <- c(1, 3, 5)
			b <- c(2, 3, 6)
			print(a>=b)
			
			It will give us the following output:
			[1] FALSE TRUE FALSE

5.	==	This operator will return TRUE when every element in the first vector is equal to thecorresponding element of thesecond vector.	a <- c(1, 3, 5)
			b <- c(2, 3, 6)
			print(a==b)
			
			It will give us the following output:
			[1] FALSE TRUE FALSE

6.	!=	This operator will return TRUE when every element in the first vector is not equal to the corresponding element of the second vector.	a <- c(1, 3, 5)
			b <- c(2, 3, 6)
			print(a>=b)
			
			It will give us the following output:
			[1] TRUE FALSE TRUE

Assignment Operators

S. No	Operator	Description	Example

1.	<- or = or <<-	These operators are known as left assignment operators.	a <- c(3, 0, TRUE, 2+2i) b <<- c(2, 4, TRUE, 2+3i)d = c(1, 2, TRUE, 2+3i)
			print(a)
			print(b)
			print(d)
			
			It will give us the following output:
			[1] 3+0i 0+0i 1+0i 2+2i [1] 2+0i 4+0i 1+0i 2+3i [1] 1+0i 2+0i 1+0i 2+3i

2.	-> or ->>	These operators are known as right assignment operators.	c(3, 0, TRUE, 2+2i) -> a c(2, 4, TRUE, 2+3i) ->> b print(a) print(b) **It will give us the following output:** [1] 3+0i 0+0i 1+0i 2+2i[1] 2+0i 4+0i 1+0i 2+3i

Logical Operators

S. No	Operator	Description	Example

1.	&	This operator is known as the Logical AND operator. This operator takes the first element of both the vector and returns TRUE if both the elements are TRUE.	a <- c(3, 0, TRUE, 2+2i)
			b <- c(2, 4, TRUE, 2+3i)
			print(a&b)
			It will give us the followingoutput:
			[1] TRUE FALSE TRUETRUE

2.	|	This operator is called the Logical OR operator. This operator takes the first element of both the vector and returnsTRUE if one of them is TRUE.	a <- c(3, 0, TRUE, 2+2i)
			b <- c(2, 4, TRUE, 2+3i)
			print(a|b)
			It will give us the followingoutput:
			[1] TRUE TRUE TRUE TRUE

3.	!	This operator is known as Logical NOT operator. This operator takes the first element of the vector and gives the opposite logical value as a result.	a <- c(3, 0, TRUE, 2+2i)
			print(!a)
			It will give us the following output:
			[1] FALSE TRUE FALSE FALSE

4.	&&	This operator takes the first element of both the vector and gives TRUE as a result, only if both are TRUE.	a <- c(3, 0, TRUE, 2+2i)
			b <- c(2, 4, TRUE, 2+3i)
			print(a&&b)
			It will give us the following output:
			[1] TRUE

5.	||	This operator takes the first element of both the vector and gives the result TRUE, if one of them is true.	a <- c(3, 0, TRUE, 2+2i)
			b <- c(2, 4, TRUE, 2+3i)
			print(a||b)
			It will give us the following output:
			[1] TRUE

Miscellaneous Operators

S. No	Operator	Description	Example

1.	:	The colon operator is used to create the series of numbers in sequencefor a vector.	v <- 1:8
			print(v)
			
			It will give us the following output:
			[1] 1 2 3 4 5 6 7 8

2.	%in%	This is used when we want to identify if an element belongs to a vector.	a1 <- 8
			a2 <- 12
			d <- 1:10
			print(a1%in%t) print(a2%in%t)
			
			It will give us the following output:
			[1] FALSE
			[1] FALSE

3.	%*%	It is used to multiply a matrix with its transpose.	M=matrix(c(1,2,3,4,5,6),nrow=2, ncol=3, byrow=TRUE) T=m%*%T(m) print(T) It will give us the following output: 14 32 32 77

Data Structures in R Programming language:

- ✓ R provides several data structures to organize and manage data effectively. Each structure has its own characteristics and use cases.

Types of Data Structure available in R

- ✓ Vector
- ✓ List
- ✓ Data frame
- ✓ Matrix
- ✓ Array
- ✓ Factor

Vector

- ✓ In R, a sequence of elements which share the same data type is known as vector

- ✓ Vectors in R are one-dimensional arrays that can hold multiple elements, but **all elements must be of the same type**

<u>How to create a Vector:</u>

<u>Using c():</u>

- ✓ You can create vectors using the c() function:

Ex:

```
numeric_vector <- c(1, 2, 3, 4, 5)
character_vector <- c("apple", "banana", "orange")
logical_vector <- c(TRUE, FALSE, TRUE)
```

<u>Using the Colon Operator (:)</u>

- ✓ The colon operator generates a sequence of numbers from a starting value to an ending value:

Ex:

```
vector1 <- 1:10
print(vector1)  # Output: 1 2 3 4 5 6 7 8 9 10
```

<u>Using the seq() Function</u>

- ✓ The seq() function provides more flexibility, allowing you to specify the starting point, ending point, and the increment (step size):

Ex:

```
vector3 <- seq(1, 10, by = 1)
vector4 <- seq(1, 10, by = 2)
```

<u>Atomic Vectors in R</u>

- ✓ In R, atomic vectors are the simplest data structures, and they can hold elements of a single data type.

<u>Different types of Atomic Vectors in R:</u>

Numeric Vectors

- ✓ Hold numeric values, which can be either integers or real numbers (floating point)
- ✓ Ex: numeric_vector <- c(1.5, 2.7, 3.1)

Integer Vectors

- ✓ Specifically hold integer values. You can denote integers by appending an L to the number
- ✓ Ex: integer_vector <- c(1L, 2L, 3L)

Logical Vectors

- ✓ Hold boolean values: TRUE or FALSE.

- ✓ Ex: logical_vector <- c(TRUE, FALSE, TRUE)

Character Vectors

- ✓ Hold text strings.
- ✓ Ex: character_vector <- c("apple", "banana", "cherry")

Complex Vectors

- ✓ Hold complex numbers, which consist of a real part and an imaginary part.
- ✓ Ex: complex_vector <- c(1+2i, 3+4i, 5+6i)

Raw Vectors

- ✓ Hold raw bytes. This type is less commonly used but can be useful for certain applications, like handling binary data.
- ✓ Ex: raw_vector <- as.raw(c(1, 2, 3))

Accessing elements of vectors:

- ✓ In R, vectors are ordered collections of elements. To access individual elements or subsets of elements within a vector, you use square brackets ([]).

Basic Indexing:

Single element:

- ✓ Use the index of the element within the vector:
- ✓ **Ex:**my_vector <- c(1, 2, 3, 4, 5)
  first_element <- my_vector[1] # Accesses the first element (1)

Negative indices:

- ✓ Use negative indices to exclude elements:
- ✓ Ex: elements_without_first <- my_vector[-1] # Excludes the first element

Logical Indexing:

- ✓ Use a logical vector to select elements based on a condition:
- ✓ even_numbers <- my_vector[my_vector %% 2 == 0] # Selects even numbers

Named Indexing:

- ✓ If your vector has names, you can access elements by name:
- ✓ Ex: named_vector <- c(a = 1, b = 2, c = 3)
  element_a <- named_vector["a"]

Accessing a Range of Elements

- You can access a range of elements using the colon operator:
- ✓ Ex: range_elements <- my_vector [2:4]
  print(range_elements) # Output: 20 30 40

Vector Operation:

Arithmetic Operations

- ✓ You can perform arithmetic operations directly on vectors. R applies these operations element-wise.
- ✓ Addition, Subtraction, Multiplication, and Division

Ex:

```
vector_a <- c(1, 2, 3)
vector_b <- c(4, 5, 6)
sum_vector <- vector_a + vector_b
print(sum_vector)  # Output: 5 7 9
diff_vector <- vector_a - vector_b
print(diff_vector)  # Output: -3 -3 -3
prod_vector <- vector_a * vector_b
print(prod_vector)  # Output: 4 10 18
div_vector <- vector_a / vector_b
print(div_vector)  # Output: 0.25 0.4 0.5
```

Vectorized Functions

- ✓ Many functions in R are vectorized, meaning they operate on entire vectors.

Ex:

```
sqrt_vector <- sqrt(vector_a)
print(sqrt_vector)  # Output: 1.0000 1.4142 1.7321
```

Logical Operations

- ✓ You can perform logical operations using comparison operators

Ex:

```
vector_c <- c(1, 2, 3, 4, 5)
# Logical comparisons
logical_vector <- vector_c > 2
print(logical_vector)  # Output: FALSE FALSE TRUE TRUE TRUE
```

.

Summary Statistics

- ✓ You can easily compute summary statistics on vectors.

Ex:

```
# Minimum and Maximum
min_value <- min(vector_c)
max_value <- max(vector_c)
print(c(min_value, max_value))  # Output: 1 5
```

Sorting and Reordering

- ✓ You can sort vectors using the sort() function.

Ex:

```
# Sorting a vector
sorted_vector <- sort(vector_c, decreasing = TRUE)
print(sorted_vector)  # Output: 5 4 3 2 1
```

Repeating and Sequencing

- ✓ You can repeat or create sequences of vectors using rep() and seq() functions.

Ex:

```
# Repeating elements
repeated_vector <- rep(vector_a, times = 3)
print(repeated_vector)  # Output: 1 2 3 1 2 3 1 2 3
# Creating a sequence
sequence_vector <- seq(1, 10, by = 2)
print(sequence_vector)  # Output: 1 3 5 7 9
```

Combining Vectors

- ✓ You can concatenate or combine vectors using the `c()` function.

Ex:

```
# Combining two vectors
combined_vector <- c(vector_a, vector_b)
print(combined_vector)  # Output: 1 2 3 4 5 6
```

R Lists

- ✓ In R, a **list** is a data structure that can hold elements of different types (e.g., numeric, character, logical, or even other lists).
- ✓ lists can store heterogeneous data, making them a powerful tool for handling complex data structures and results from analyses.

Creating Lists:

- ✓ You can create lists using the `list()` function:

Ex:

```
list_1<-list(1,2,3)
```

```
my_list <- list( name = "Alice",age = 30,city = "New York"  hobbies = c("reading", "painting"))
```

```
list_data←list("Shubham","Arpita",c(1,2,3,4,5),TRUE,FALSE,22.5,12L)
```

Giving a name to list elements

There are only three steps to print the list data corresponding to the name:

1. Creating a list.
2. Assign a name to the list elements with the help of names () function.
3. Print the list data.

Example

# Creating a list containing a vector, a matrix and a list.

```
list_data <- list(c("Shubham","Nishka","Gunjan"), matrix(c(40,80,60,70,90,80),
 nrow = 2),list("BCA","MCA","B.tech"))
```

# Giving names to the elements in the list.

```
names(list_data) <- c("Students", "Marks", "Course")
print(list_data)
```

Accessing List Elements

- ✓ Accessing elements in a list in R can be done using several methods, allowing for flexibility depending on how you want to access the data

Numeric Indexing

Ex:

```
# Creating a list
my_list <- list(name = "Alice", age = 30, scores = c(90, 85, 88))
# Accessing the first element (name)
name <- my_list[[1]]
print(name)  # Output: "Alice"
```

Named Indexing

Ex:

```
# Accessing the age using its name
age <- my_list[["age"]]
print(age)  # Output: 30
```

Using the Dollar Sign $

- ✓ The $ operator allows you to access list elements by their names directly. This is often more readable.

Ex:

```
# Accessing elements by name
scores <- my_list$scores
print(scores)  # Output: 90 85 88
```

Accessing Multiple Elements

- ✓ You can access multiple elements at once using a vector of indices.

Ex:

```
# Accessing multiple elements
selected_elements <- my_list[c("name", "scores")]
print(selected_elements)
```

Manipulating List Elements in R

- ✓ R allows us to add, delete, or update elements in the list.

Adding Elements:

- Use the `$` operator or `[[` and `]]` indexing:

```
Ex: my_list$d <- 4
my_list[[5]] <- "hello"
```

Removing Elements or delete list

- Use the – operator:

```
my_list <- my_list[-2]  # Removes the second element
```

Modifying Elements: update list

- Assign a new value to an existing element:

Ex:
my_list$b <- 10

Creating Sublists:

- Extract a subset of elements to create a new list:

Ex: sublist <- my_list[c(1, 3)]

Combining Lists:

- Use the `c()` function to combine two or more lists:
  combined_list <- c(my_list, other_list)

Converting list to vector

- ✓ There is a drawback with the list, i.e., we cannot perform all the arithmetic operations on list elements.
- ✓ This drawback can be overcome with the function unlist( ), this function converts the list into vectors.
- ✓ Example:

```
# Creating lists.

 list1 <- list(1:5)
print(list1)
```

```
list2 <-list(10:14)
 print(list2)
# Converting the lists to vectors.
v1 <-unlist(list1)
v2 <- unlist(list2)
result <- v1+v2
print(result)
```

Merging Lists

- ✓ R allows us to merge one or more lists into one list.
- ✓ To merge the lists, we have to pass all the lists into list function as a parameter, and it returns a list which contains all the elements which are present in the lists.

Example

```
# Creating two lists.
Even_list <- list(2,4,6,8)
Odd_list <- list(3,5,7,9)
# Merging the two lists.
merged.list <- list(Even_list,Odd_list)
```

R Matrix

- ✓ A **matrix** is a two-dimensional data structure in R, similar to a table, where elements are arranged in rows and columns.
- ✓ All elements in a matrix must be of the **same data type** (numeric, character, logical, etc.). Matrices are very useful for mathematical and statistical computations, particularly in linear algebra and machine learning.

Creating Matrices:

- ✓ You can create matrices using the matrix () function

Syntax:

```
matrix(data, nrow, ncol, byrow = FALSE, dimnames = NULL)
```

- ✓ **`data`:** A vector of values that you want to fill into the matrix.
- ✓ **`nrow`:** The number of rows in the matrix.
- ✓ **`ncol`:** The number of columns in the matrix.

- ✓ **byrow:** A logical value that determines if the matrix should be filled row-wise (`TRUE`) or column-wise (`FALSE`).
- ✓ **dimnames:** Optionally, you can specify the names of rows and columns.

Example: p <- matrix(c(5:16), nrow=4, ncol=3, byrow=TRUE)

Creating a Matrix with Named Rows and Columns

- ✓ You can assign names to the rows and columns when you create the matrix using the `dimnames` argument.

Ex:

```
my_matrix <- matrix(1:9, nrow = 3, ncol = 3, dimnames = list(c("Row1", "Row2",
"Row3"), c("Col1", "Col2", "Col3")))
print(my_matrix)
```

Accessing matrix elements in R

- ✓ Accessing elements in a matrix in R can be done in several ways, depending on whether you want to access a single element, an entire row or column, or multiple elements.

Basic Accessing Using Indices

- ✓ You can access elements of a matrix using row and column indices with the syntax matrix[row, column].

Ex

```
# Creating a matrix
my_matrix <- matrix(1:9, nrow = 3, ncol = 3)
print(my_matrix)
# Accessing the element in the 2nd row, 3rd column
element <- my_matrix[2, 3]
print(element)  # Output: 8
```

Accessing Entire Rows or Columns

- ✓ You can access entire rows or columns by omitting the index for the other dimension.

Accessing a Row

```
# Accessing the second row
second_row <- my_matrix[2, ]
print(second_row)  # Output: 2 5 8
```

Accessing a Column

```
# Accessing the third column
third_column <- my_matrix[, 3]
print(third_column)  # Output: 7 8 9
```

Accessing Multiple Elements

✓ You can access multiple elements using vectors of indices.

```
# Accessing multiple elements
sub_matrix <- my_matrix[c(1, 3), c(2, 3)]  # Rows 1 and 3, Columns 2 and 3
print(sub_matrix)
```

Accessing Named Elements

✓ If you have assigned names to the rows and columns, you can access elements using those names.

```
# Assigning names to rows and columns
rownames(my_matrix) <- c("Row1", "Row2", "Row3")
colnames(my_matrix) <- c("Col1", "Col2", "Col3")
# Accessing an element by row and column names
named_element <- my_matrix["Row2", "Col3"]
print(named_element)  # Output: 8
# Accessing an entire row or column by name
named_row <- my_matrix["Row2", ]
print(named_row)  # Output: 2 5 8
named_column <- my_matrix[, "Col3"]
print(named_column)  # Output: 7 8 9
```

Using Logical Indexing

✓ You can also access elements using logical conditions.

```
# Accessing elements greater than 5
greater_than_five <- my_matrix[my_matrix > 5]
print(greater_than_five)  # Output: 6 7 8 9
```

Modification of the matrix

Modifying Specific Elements

✓ You can change the value of specific elements in a matrix by accessing them with their row and column indices

```
my_matrix[1, 1] <- 10

print(my_matrix)
```

Modifying Entire Rows

✓ You can replace an entire row with new values.

```
# Modifying the second row
my_matrix[2, ] <- c(20, 21, 22)
print(my_matrix)
```

Modifying Entire Columns

Similarly, you can replace an entire column with new values.

```
# Modifying the third column
my_matrix[, 3] <- c(30, 31, 32)
print(my_matrix)
```

Adding Rows or Columns

- ✓ You can add new rows or columns using `rbind()` and `cbind()`.

Adding a Row

```
# Adding a new row
new_row <- c(7, 8, 9)
my_matrix <- rbind(my_matrix, new_row)
print(my_matrix)
```

Adding a Column

```
# Adding a new column
new_column <- c(10, 11, 12, 13)
my_matrix <- cbind(my_matrix, new_column)
print(my_matrix)
```

Removing Rows or Columns

- ✓ You can remove rows or columns by setting them to `NULL` or by using negative indexing.

Removing a Row

```
# Removing the second row
my_matrix <- my_matrix[-2, ]
print(my_matrix)
```

Removing a Column

```
# Removing the third column
my_matrix <- my_matrix[, -3]
print(my_matrix)
```

Matrix operations

- ✓ In R, we can perform the mathematical operations on a matrix such as addition,subtraction, multiplication, etc.

Element-wise Operations

```
# Create two matrices
A <- matrix(1:9, nrow = 3, byrow = TRUE)
B <- matrix(9:1, nrow = 3, byrow = TRUE)
print(A)
print(B)
# Element-wise addition
sum_matrix <- A + B
print(sum_matrix)
# Element-wise subtraction
diff_matrix <- A - B
print(diff_matrix)
# Element-wise multiplication
prod_matrix <- A * B
print(prod_matrix)
# Element-wise division
div_matrix <- A / B
print(div_matrix)
```

Matrix Multiplication

- ✓ **To perform matrix multiplication, use the %*% operator.**

```
# Matrix multiplication
C <- matrix(c(1, 2, 3, 4), nrow = 2)
D <- matrix(c(5, 6, 7, 8), nrow = 2)
product_matrix <- C %*% D
print(product_matrix)
```

Transpose a Matrix

- ✓ You can transpose a matrix using the t() function.

Ex

```
# Transposing a matrix
transposed_A <- t(A)
print(transposed_A)
```

Inversion of a Matrix

- ✓ For square matrices, you can calculate the inverse using the solve() function.

Ex:

```
# Creating a square matrix
square_matrix <- matrix(c(4, 7, 2, 6), nrow = 2)
# Inverting the matrix
inverse_matrix <- solve(square_matrix)
print(inverse_matrix)
```

Determinant of a Matrix

- ✓ To find the determinant of a square matrix, use the `det()` function.

Ex:

```
# Determinant of the square matrix
determinant_value <- det(square_matrix)
print(determinant_value)
```

Row and Column Operations

- ✓ Perform operations such as sums and means on rows or columns.

Sum of Rows

```
# Sum of each row
row_sums <- rowSums(A)
print(row_sums)
```

Mean of Columns

```
# Mean of each column
column_means <- colMeans(A)
print(column_means)
```

Applications of matrix

1. Matrix is the representation method which helps in plotting commonsurvey things.
2. In robotics and automation, Matrices have the topmost elements for the robot movements.
3. In computer-based application, matrices play a crucial role in the creationof realistic seeming motion.

Arrays

- ✓ In R, an array is a multi-dimensional data structure that can hold elements of the same data type.
- ✓ Arrays are particularly useful for representing data in multiple dimensions, such as 2D matrices or 3D datasets
- ✓ arrays can have **two or more dimensions**, making them useful for handling data in higher dimensions, such as 3D or even higher-dimensional datasets

Creating Arrays:

- ✓ You can create arrays using the `array()` function:

Syntax

```
array(data, dim = c(dim1, dim2, ...), dimnames = NULL)
```

- ✓ `data`: A vector of elements you want to fill into the array.
- ✓ `dim`: A vector specifying the dimensions of the array (e.g., `c(2, 3)` for a 2x3 matrix or `c(2, 3, 4)` for a 3D array).
- ✓ `dimnames`: Optional. A list of names for the dimensions.

Ex:

```
# Creating a 3D array
my_array <- array(1:24, dim = c(4, 3, 2))  # 4 rows, 3 columns, 2 layers
print(my_array)
```

Creation of an Array:

- ✓ **1D Array**
- ✓ **2D Array** (Matrix-like structure with rows and columns)
- ✓ **3D Array** (3 dimensions, like a stack of matrices)

1D Array

```
arr1 <- array(1:6, dim = c(6))
print(arr1)
# Output:
# [1] 1 2 3 4 5 6
```

2D Array (Matrix-like structure with rows and columns)

```
arr2 <- array(1:6, dim = c(2, 3))
print(arr2)
# Output:
#      [,1] [,2] [,3]
# [1,]    1    3    5
# [2,]    2    4    6
```

3D Array (3 dimensions, like a stack of matrices):

```
arr3 <- array(1:12, dim = c(2, 3, 2))
print(arr3)
# Output:
# , , 1
#
#      [,1] [,2] [,3]
# [1,]    1    3    5
# [2,]    2    4    6
#
# , , 2
#
#      [,1] [,2] [,3]
# [1,]    7    9   11
# [2,]    8   10   12
```

Naming rows and columns or Array with Dimension Names:

- ✓ In R, we can give the names to the rows, columns, and matrices of the array. This is done with the help of the dim name parameter of the array() function.

Example:

#Creating two vectors of different lengths

```
vec1 <-c(1,3,5)
vec2 <-c(10,11,12,13,14,15)
```

#Initializing names for rows, columns and matrices

```
col_names <- c("Col1","Col2","Col3")
row_names <- c("Row1","Row2","Row3")
matrix_names <- c("Matrix1","Matrix2")
```

#Taking the vectors as input to the array

```
res <- array(c(vec1,vec2),dim=c(3,3,2),dimnames=list(row_names,col_names,m
atrix_names))
print(res)
```

Accessing array elements

Accessing Specific Elements

- ✓ To access an element, use the syntax array[row, column, layer].

Ex:

```
# Accessing the element in the 2nd row, 3rd column, 1st layer
element <- my_array[2, 3, 1]
print(element)  # Output: 10
```

Accessing Entire Rows or Columns

- ✓ **You can access entire rows or columns by using , to omit the other indices.**

Accessing an Entire Row

Ex:

```
# Accessing the 1st row across all columns and layers
first_row <- my_array[1, , ]
print(first_row)
```

Accessing an Entire Column

```
# Accessing the 2nd column across all rows and layers
second_column <- my_array[, 2, ]
print(second_column)
```

Slicing the Array

- ✓ You can extract subsets of the array using ranges for indices.

```
# Accessing a sub-array (rows 1-2, columns 1-2, all layers)
sub_array <- my_array[1:2, 1:2, ]
print(sub_array)
```

Using Logical Indexing

- ✓ You can also use logical conditions to access elements that meet specific criteria.

```
# Accessing elements greater than 10
greater_than_ten <- my_array[my_array > 10]
print(greater_than_ten)
```

Using Dimnames for Access

- ✓ If you have assigned names to the dimensions, you can access elements using those names.

```
    # Assigning dimnames
dimnames(my_array) <- list(c("Row1", "Row2", "Row3", "Row4"),c("Col1", "Col2",
"Col3"), c("Layer1", "Layer2"))
# Accessing an element using dimnames
named_element <- my_array["Row2", "Col3", "Layer1"]
print(named_element)  # Output: 10
```

Manipulation on Array

Modifying Elements

- ✓ You can change specific elements by accessing them with their indices.

```
# Modifying the element in the 1st row, 1st column, 1st layer
my_array[1, 1, 1] <- 100
print(my_array)
```

Modifying Rows or Columns

- ✓ You can replace entire rows or columns with new values.

Modifying Rows

```
# Modifying the 2nd row
my_array[2, , ] <- c(20, 21, 22)
print(my_array)
```

Modifying Columns

```
# Modifying the 3rd column
my_array[, 3, 1] <- c(30, 31, 32, 33)
print(my_array)
```

Adding Layers, Rows, or Columns

- ✓ You can add new layers, rows, or columns using the `abind` package or by manipulating the dimensions.

Example Using abind:

```
# Load the abind package
# install.packages("abind")  # Uncomment to install
library(abind)
# Adding a new layer
new_layer <- array(25:36, dim = c(4, 3, 1))
combined_array <- abind(my_array, new_layer, along = 3)
print(combined_array)
```

Removing Layers, Rows, or Columns

- ✓ You can remove layers, rows, or columns by setting them to `NULL` or using negative indexing.

Removing Layers

```
# Removing the 2nd layer
my_array <- my_array[, , -2]
print(my_array)
```

Removing Row

```
# Removing the 3rd row
my_array <- my_array[-3, , ]
print(my_array)
```

Operations on Array

element-wise Operations

- ✓ You can perform arithmetic operations on each element of the array.

```
# Adding a constant to each element
added_array <- my_array + 10
print(added_array)
# Subtracting a constant from each element
subtracted_array <- my_array – 5
print(subtracted_array)
# Multiplying each element by a constant
multiplied_array <- my_array * 2
print(multiplied_array)
# Dividing each element by a constant
divided_array <- my_array / 2
print(divided_array)
```

Data Frame

- ✓ Data frames in R are one of the most commonly used data structures for storing tabular data.
- ✓ They are similar to spreadsheets or SQL tables and can contain different types of data in each column (numeric, character, factor, etc.)
- ✓ A data frame is a special case of the list in which each component has equal length

There are following characteristics of a data frame.

- ✓ The columns name should be non-empty.
- ✓ The rows name should be unique.
- ✓ The data which is stored in a data frame can be a factor, numeric, or character type.
- ✓ Each column contains the same number of data items.

Creating a Data Frame

- ✓ You can create a data frame using the data.frame() function.

Syntax:

```
data.frame(column1 = vector1, column2 = vector2, ...)
```

- `column1`, `column2`, etc. are the names of the columns.
- `vector1`, `vector2`, etc. are the vectors (data) that will form the columns.

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

Accessing Data Frame Elements

- ✓ You can access columns directly using the $ operator or the double square brackets [[ ]].

```
# Creating a sample data frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Height = c(5.5, 6.0, 5.8),
  stringsAsFactors = FALSE
)
# Accessing a specific column using the $ operator
ages <- df$Age
```

```
print(ages)  # Output: 25 30 35

# Accessing a specific column using [[ ]]
heights <- df[["Height"]]
print(heights)  # Output: 5.5 6.0 5.8
```

Using Indices

You can access specific rows and columns using numerical indices with single brackets `[ ]`.

Accessing Rows

```
# Accessing the first row
first_row <- df[1, ]
print(first_row)
# Accessing multiple rows (1st and 3rd)
subset_rows <- df[c(1, 3), ]
print(subset_rows)
```

Accessing Columns by Index

```
Accessing the second column
cond_column <- df[, 2]
int(second_column)  # Output: 25 30 35
```

Accessing Specific Elements

```
# Accessing a specific element (2nd row, 3rd column)
element <- df[2, 3]
print(element)  # Output: 6.0
```

Using Logical Conditions

- ✓ You can filter rows based on specific conditions using logical indexing.

```
# Accessing rows where Age is greater than 28
filtered_df <- df[df$Age > 28, ]
print(filtered_df)
```

Using the subset() Function

- ✓ The `subset()` function is useful for extracting specific subsets of a data frame based on conditions.

```
# Using subset to filter data frame
subset_df <- subset(df, Age < 35)
print(subset_df)
```

Using the **head()** and **tail()** Functions

- ✓ You can view the first or last few rows of a data frame.

```
# Viewing the first 2 rows
```

```
head_rows <- head(df, 2)
print(head_rows)

# Viewing the last row
tail_rows <- tail(df, 1)
print(tail_rows)
```

Modification in Data Frame

- ✓ Modifying a data frame in R is essential for data cleaning and preparation. Below are various ways to modify data frames, including adding, changing, and removing rows and columns.

Adding Columns

- ✓ You can add new columns using the `$` operator or by directly assigning a new vector.

```
# Sample data frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Height = c(5.5, 6.0, 5.8))
# Adding a new column for Weight
df$Weight <- c(130, 160, 150)
print(df)
```

Modifying Existing Columns

- ✓ You can modify existing columns by directly assigning new values.

```
# Incrementing the Age column by 1 year
df$Age <- df$Age + 1
print(df)
```

Removing Columns

- ✓ To remove a column, you can set it to `NULL` or use the `dplyr` package.

```
# Removing the Weight column
df$Weight <- NULL
print(df)

# Alternatively, using dplyr
library(dplyr)
df <- select(df, -Height)  # Removing Height column
print(df)
```

Adding Rows

- ✓ You can add new rows using the rbind() function.

```
# Creating a new row
```

```
new_row <- data.frame(Name = "David", Age = 28, Height = 5.9, stringsAsFactors = FALSE)

# Adding the new row to the data frame
df <- rbind(df, new_row)
print(df)
```

Modifying Rows

- ✓ To modify specific rows, use their indices.

```
# Changing the Age of the first row
df[1, "Age"] <- 26
print(df)
```

Removing Rows

- ✓ You can remove rows by specifying their indices.

```
# Removing the second row
df <- df[-2, ]
print(df)
```

Renaming Columns

- ✓ To rename columns, use `names()` or `colnames()`.

```
# Renaming the Height column to "Tallness"
names(df)[names(df) == "Height"] <- "Tallness"
print(df)

# Using colnames() to rename
colnames(df)[2] <- "Years"
print(df)
```

Factors in R

- ✓ A **factor** is an R data structure used to represent categorical data. Factors are important when dealing with variables that take on a limited number of unique values, known as **levels**.
- ✓ Factors are typically used for representing variables such as "gender", "region", "status", or any type of data that falls into a fixed set of categories.

Creating a Factor

- ✓ You can create a factor using the `factor()` function.

Syntax :

```
factor(x, levels = NULL, labels = NULL, ordered = FALSE)
```

where

- x: A vector (usually a character or numeric vector) to convert into a factor.

- levels: A vector of all possible levels (categories) of the factor.
- labels: Optional. A vector of labels corresponding to the levels.
- ordered: A logical value indicating whether the factor should be treated as ordered (TRUE) or unordered (FALSE).

```
survey <- c("High", "Low", "Medium", "Low", "High")
survey_factor <- factor(survey, levels = c("Low", "Medium", "High"))
print(survey_factor)
```

Accessing Factor Levels

- ✓ You can access the levels of a factor using the `levels()` function.

```
levels(gender_factor)
```

Modifying Factors

- ✓ **Changing Levels**: You can change the levels of a factor using the `levels()` function.

```
levels(gender_factor) <- c("F", "M")
print(gender_factor)
```

Renaming Levels: You can rename the levels of a factor by assigning a new set of level names.

```
levels(gender_factor)[levels(gender_factor) == "F"] <- "Female"
levels(gender_factor)[levels(gender_factor) == "M"] <- "Male"
print(gender_factor)
```

Non-Numeric Values

- ✓ In R, **non-numeric values** refer to data types that cannot be represented by numbers. These types are used to store various kinds of data that are not inherently numerical.
- ✓ R provides a few important data types to handle non-numeric values, such as **character strings**, **factors**, **logical values**, and **dates**.

Character Data Type (Strings)

- ✓ **Character** data type represents text or strings of characters.
- ✓ Character values are enclosed in either single quotes (') or double quotes (").

```
name <- "John"
```

```
city <- 'New York'
```

Logical Data Type (Boolean)

- ✓ **Logical** data type represents Boolean values: TRUE and FALSE.
- ✓ These values are used in conditional statements, comparisons, and logical operations.
- ✓ Logical-values represent binary states like :->yes/no that indicate ->one/zero
- ✓ Logical-values are used to indicate whether a condition has been met or not.

  TRUE and FALSE Notation
- ✓ TRUE and FALSE are abbreviated as T and F, respectively.

Example

```
is_raining <- TRUE
is_sunny <- FALSE
```

Creating Vectors using logical values

- ✓ Vectors can be filled with logical-values using T or F.
- ✓ Example: myvec <- c(T,T,F,F,F)

Using Relational Operators

- ✓ Relational operators are used to find the relationship between two operands.
- ✓ The output of relational expression is either TRUE or FALSE

Operator	**Meaning of Operator**	**Example**
>	Greater than	4 > 5 returns FALSE
<	Less than	4 < 5 returns TRUE
>=	Greater than or equal to	4 >= 5 returns FALSE
<=	Less than or equal to	4 <= 5 returns TRUE
==	Equal to	4 == 5 returns FALSE
!=	Not equal to	4 != 5 returns TRUE

Factors

- ✓ **Factors** represent categorical data with a fixed set of unique values, or "levels". These are useful for representing variables such as "gender", "status", "region", etc.
- ✓ Factors can be unordered (nominal data) or ordered (ordinal data)

```
# Creating a factor for gender
gender <- factor(c("Male", "Female", "Male", "Female"))
print(gender)
# Output: [1] Male Female Male Female
# Levels: Female Male
```

Dates and Times

- ✓ **Date** and **Time** objects are used to represent and manipulate date and time information.

NULL

- ✓ NULL represents the absence of a value or an empty object.
- ✓ It is often used to indicate that a variable has not been initialized or does not contain any valid data.

```
x <- NULL
print(x)
# Output: NULL
```

Coercion in R

- ✓ Coercion in R refers to the automatic or explicit conversion of one data type to another.
- ✓ This is often necessary when performing operations that involve different types of data.
- ✓ R can automatically coerce data types in various situations, particularly when combining them in vectors or data frames.

Types of Coercion

- ✓ Automatic Coercion
- ✓ Explicit Coercion

Automatic Coercion

- ✓ R often performs coercion automatically when you attempt to mix different data types.
- ✓ The language has a set of rules for how different types should interact, and it will coerce one type to another based on these rules.
- ✓ The process typically happens from more specific types to more general types (i.e., from lower to higher levels of abstraction)

```
# Combining logical, numeric, and character values
mixed_vector <- c(TRUE, 2.5, "Hello")
print(mixed_vector)  # Output: "TRUE" "2.5" "Hello" logical and numeric convert to character
```

Numeric to Character Coercion: If you try to combine a number and a character, R will automatically coerce the number to a character to make the operation possible.

Example

```
x <- 5
y <- "apple"
z <- x + y  # Error: cannot coerce type 'double' to 'character'
# Implicit Coercion
result <- paste(x, y)  # '5' becomes character
print(result)  # "5 apple"
```

Explicit Coercion

- ✓ You can explicitly convert data types using functions like as.numeric(), as.character(), as.logical(), and as.factor().

```
# Explicit coercion examples
num_vector <- c(1, 2, 3)
char_vector <- as.character(num_vector)
print(char_vector)  # Output: "1" "2" "3"
# Coercing character to numeric
num_from_char <- as.numeric(c("4", "5", "6"))
print(num_from_char)  # Output: 4 5 6
# Coercing logical to numeric
logical_vector <- c(TRUE, FALSE, TRUE)
num_from_logical <- as.numeric(logical_vector)
print(num_from_logical)  # Output: 1 0 1
```

Coercion Rules in R

R follows a set of rules when it comes to coercing data types:

- ✓ **Character is the most general type:** If you combine a character with any other type, the other type will generally be coerced to character.
- ✓ **Numeric is more general than integer:** If you mix integers and numerics, the integer will usually be coerced to numeric.
- ✓ **Logical can be coerced to numeric:** TRUE becomes 1, and FALSE becomes 0.

Special Values

- ✓ In R, there are several **special values** that represent specific concepts like missing data, infinite values, or undefined results.
- ✓ These special values are crucial for handling exceptional cases in data analysis and can influence computations, conditionals, and data handling in R.

Here are some key special values in R:

NA (Not Available): -

- ✓ Represents a **missing** or **undefined value** in R. It is used to indicate that a value is absent or not applicable.
- ✓ **Can occur in any data type (numeric, character, logical, etc.).**
- ✓ Handling missing data in vectors, data frames, or matrices.
- ✓ Example: X< - c(1,2,3) X[4] Output: NA
- ✓ `is.na()`: Checks if a value is `NA`.

Inf and **-Inf**:

- ✓ Represent positive and negative infinity, respectively.
- ✓ These can result from operations like division by zero (e.g., `1 / 0` gives Inf, while `-1 / 0` gives -Inf). Use `is. infinite ()` to check for these values.
- ✓ Used to represent extremely large or small numbers that exceed the range of standard numeric types.

```
z <- 1 / 0
is.infinite(z)  # Returns TRUE
```

NULL:

- Represents the absence of a value or an undefined state. It is different from NA and is often used in lists or function arguments to indicate "no value."
- `is.null()`: Checks if a value is `NULL`.

```
a <- NULL
is.null(a)  # Returns TRUE
```

NaN (Not a Number)

- ✓ Represents an **undefined** or **unrepresentable number**, such as the result of 0 divided by 0 or the square root of a negative number (for non-complex numbers).
- ✓ is.nan(): Checks if a value is NaN.

```
y <- 0 / 0
print(y)  # NaN
z <- sqrt(-1)
print(z)  # NaN
# Checking for NaN
is.nan(y)  # TRUE
is.nan(NA)  # FALSE
```

TRUE and FALSE (Logical Constants)

- ✓ These are the two logical constants used to represent **boolean truth** (TRUE) and **falsehood** (FALSE).

```
x <- TRUE
y <- FALSE
# Logical operations
x & y  # FALSE
x | y  # TRUE
# Checking the type of logical values
is.logical(x)  # TRUE
```

String

- ✓ In R, **strings** are used to represent text, and they are treated as **character vectors**.
- ✓ A string in R is simply a sequence of characters enclosed in quotes

Creating Strings

- ✓ You can create strings using single (`'`) or double (`"`) quotes.

```
string1 <- "Hello, World!"
string2 <- 'R is great!'
```

Basic String Functions

- ✓ **Concatenation**: Use the paste() or paste0() functions to combine strings.

```
combined <- paste(string1, string2)  # Adds a space by default
combined_no_space <- paste0(string1, string2)  # No space
```

String Length: Use `nchar()` to find the length of a string.

```
length <- nchar(string1)  # Returns 13
```

Substrings: Use `substr()` to extract a part of a string.

```
sub <- substr(string1, 1, 5)  # Returns "Hello"
```

String Replacement: Use `gsub()` for global replacement or `sub()` for the first occurrence.

```
modified <- gsub("World", "R", string1)  # Returns "Hello, R!"
```

Changing Case: Use `toupper()` and `tolower()` to change string case.

```
upper <- toupper(string2)  # Returns "R IS GREAT!"
lower <- tolower(string2)  # Returns "r is great!"
```

String Splitting and Joining

- ✓ **Splitting**: Use `strsplit()` to split a string into substrings.

```
split_string <- strsplit(string1, ",")  # Returns a list
```

Joining: Use `paste()` to join elements of a vector into a single string.

```
words <- c("R", "is", "awesome")
sentence <- paste(words, collapse = " ")  # Returns "R is awesome"
```

String Formatting

- You can format strings using `sprintf()` or `format()`.

```
formatted_string <- sprintf("The value of pi is approximately %.2f", pi)  # Returns "The value
of pi is approximately 3.14"
```

cat() vs. paste()

- ✓ cat() is used for printing (displaying) text to the console. It concatenates and prints its arguments with optional separators.
- ✓ paste() is used for concatenating strings into a single character vector. The concatenated string can be assigned to a variable for further use. It doesn't print directly to the console.

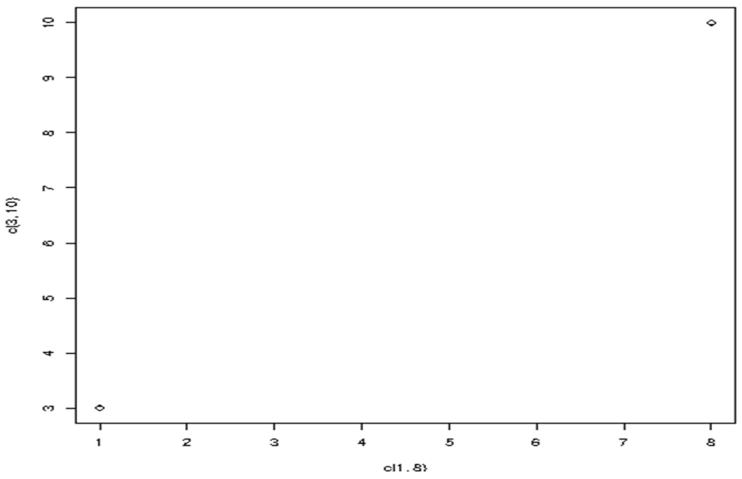
Basic plotting in R:

- ✓ The plot() function is used to draw points (markers) in a diagram.
- ✓ The function takes parameters for specifying points in the diagram.
- ✓ Parameter 1 specifies points on the **x-axis**.
- ✓ Parameter 2 specifies points on the **y-axis**.
- ✓ At its simplest, you can use the plot() function to plot two numbers against each other:

Example Draw

two points in the diagram, one at position (1, 3) and one in position (8, 10):

1) plot(c(1, 8), c(3, 10))



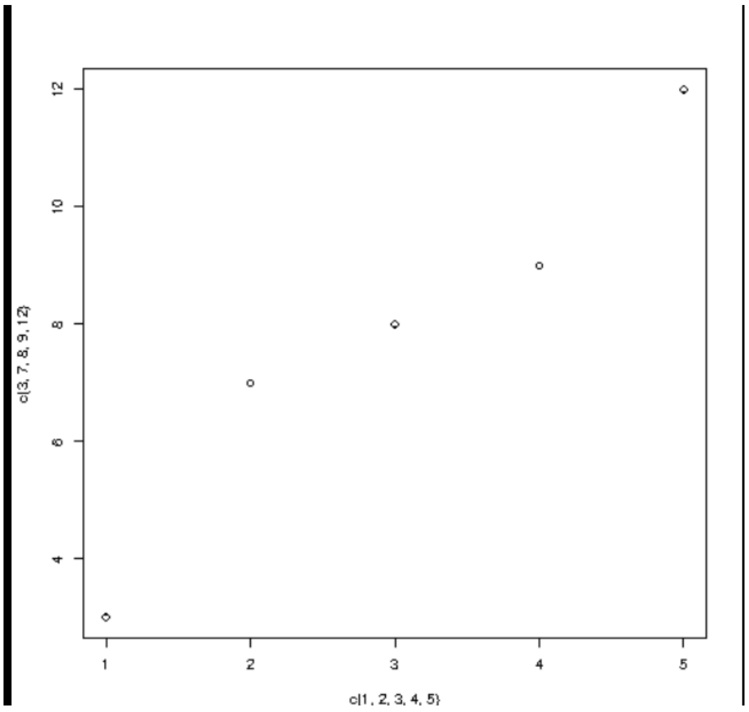
Multiple Points

- ✓ You can plot as many points as you like, just make sure you have the same number of points in both axis:

```
plot(c(1, 2, 3, 4, 5), c(3, 7, 8, 9, 12))
```

or

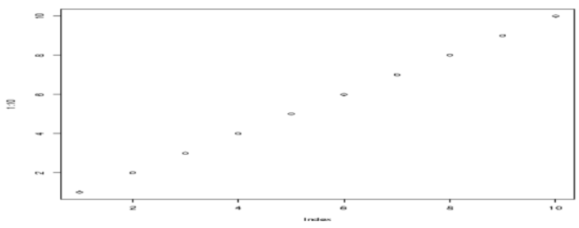
```
x <- c(1, 2, 3, 4, 5)
y <- c(3, 7, 8, 9, 12)

plot(x, y)
```



Sequences of Points

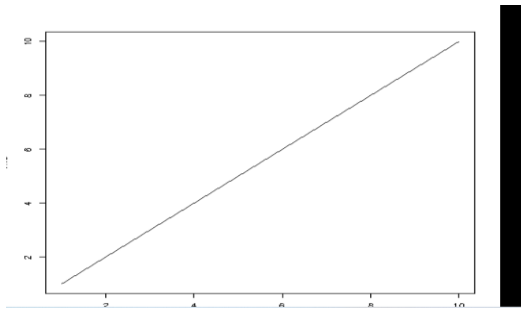
- ✓ If you want to draw dots in a sequence, on both the **x-axis** and the **y-axis**, use the : operator:

```
plot(1:10)
```



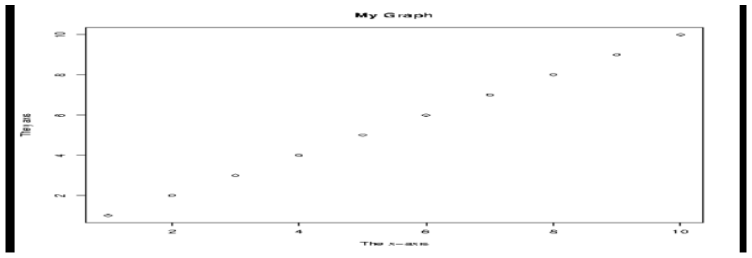
Draw a Line

- ✓ The plot() function also takes a type parameter with the value l to draw a line to connect all the points in the diagram:
- ✓ `plot(1:10, type="l")`



Plot Labels

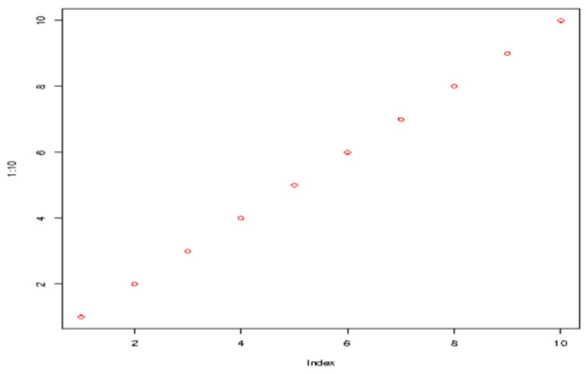
- ✓ The plot() function also accept other parameters, such as main, xlab and ylab if you want to customize the graph with a main title and different labels for the x and y-axis:
- ✓ `plot(1:10, main="My Graph", xlab="The x-axis", ylab="The y axis")`



Colors

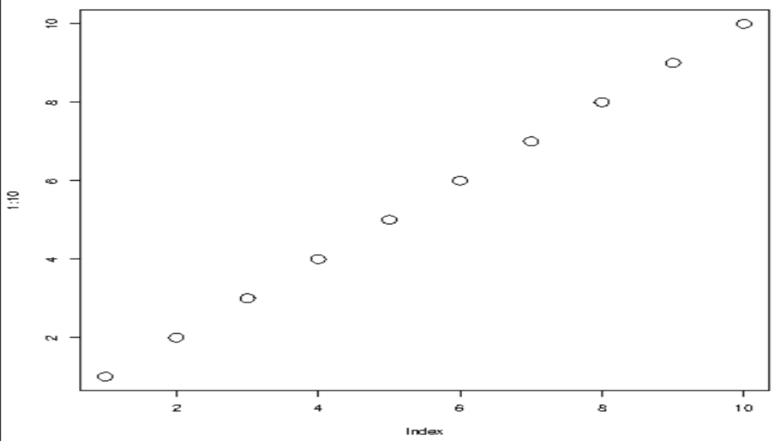
- ✓ Use col="*color*" to add a color to the points:

- ✓ `plot(1:10, col="red")`



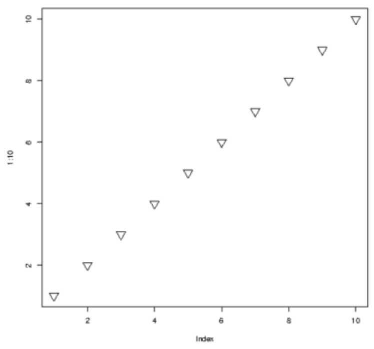
Size

- ✓ Use cex=*number* to change the size of the points (1 is default, while 0.5 means 50% smaller, and 2 means 100% larger):
- ✓ `plot(1:10, cex=2)`

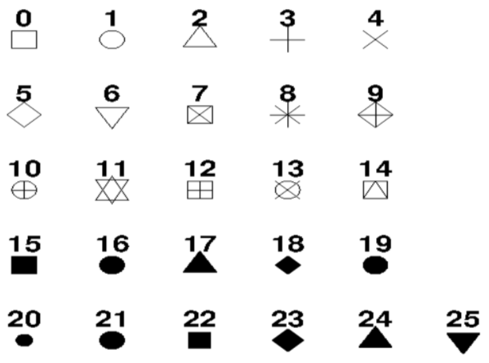


Point Shape

- ✓ Use pch with a value from 0 to 25 to change the point shape format:
- ✓ plot(1:10, pch=25, cex=2)

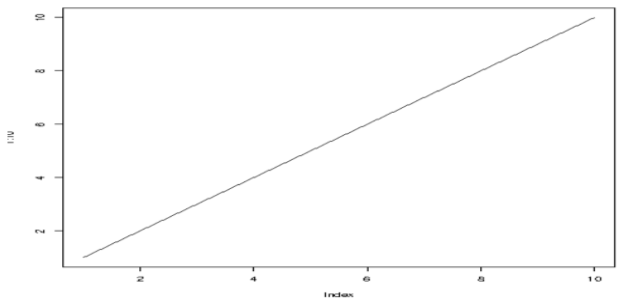


Different pch shapes with number



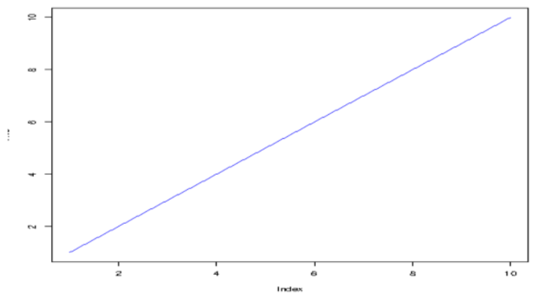
Line Graphs

- ✓ A line graph has a line that connects all the points in a diagram.
- ✓ To create a line, use the plot() function and add the type parameter with a value of "l":
- ✓ plot(1:10, type="l")



Line Color

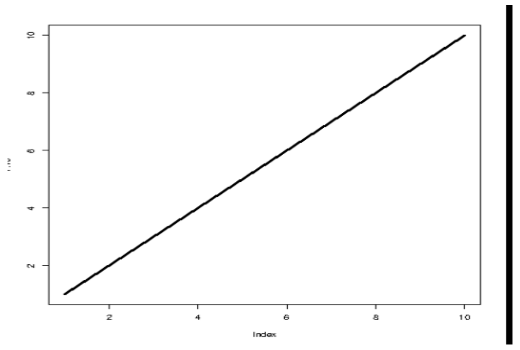
- ✓ The line color is black by default. To change the color, use the col parameter:
- ✓ plot(1:10, type="l", col="blue")



Line Width

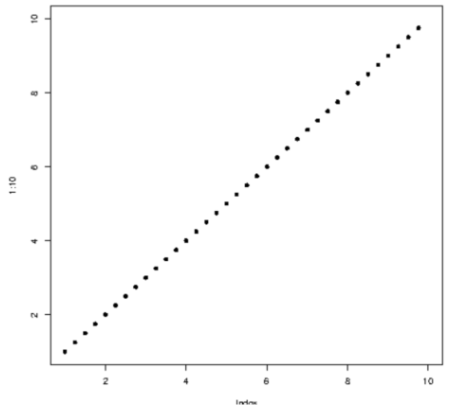
- ✓ To change the width of the line, use the lwd parameter (1 is default, while 0.5 means 50% smaller, and 2 means 100% larger):

Example plot(1:10, type="l", lwd=2)



Line Styles

- ✓ The line is solid by default. Use the lty parameter with a value from 0 to 6 to specify the line format.
- ✓ For example, lty=3 will display a dotted line instead of a solid line:
- ✓ Example plot(1:10, type="l", lwd=5, lty=3)



Available parameter values for lty:

- ✓ 0 removes the line
- ✓ 1 displays a solid line
- ✓ 2 displays a dashed line
- ✓ 3 displays a dotted line
- ✓ 4 displays a "dot dashed" line
- ✓ 5 displays a "long dashed" line
- ✓ 6 displays a "two dashed" line

Multiple Lines

- ✓ To display more than one line in a graph, use the plot() function together with the lines() function:

```
line1 <- c(1,2,3,4,5,10)
line2 <- c(2,5,7,8,9,10)
plot(line1, type = "l", col = "blue")
lines(line2, type="l", col = "red")
```