Unit-I.

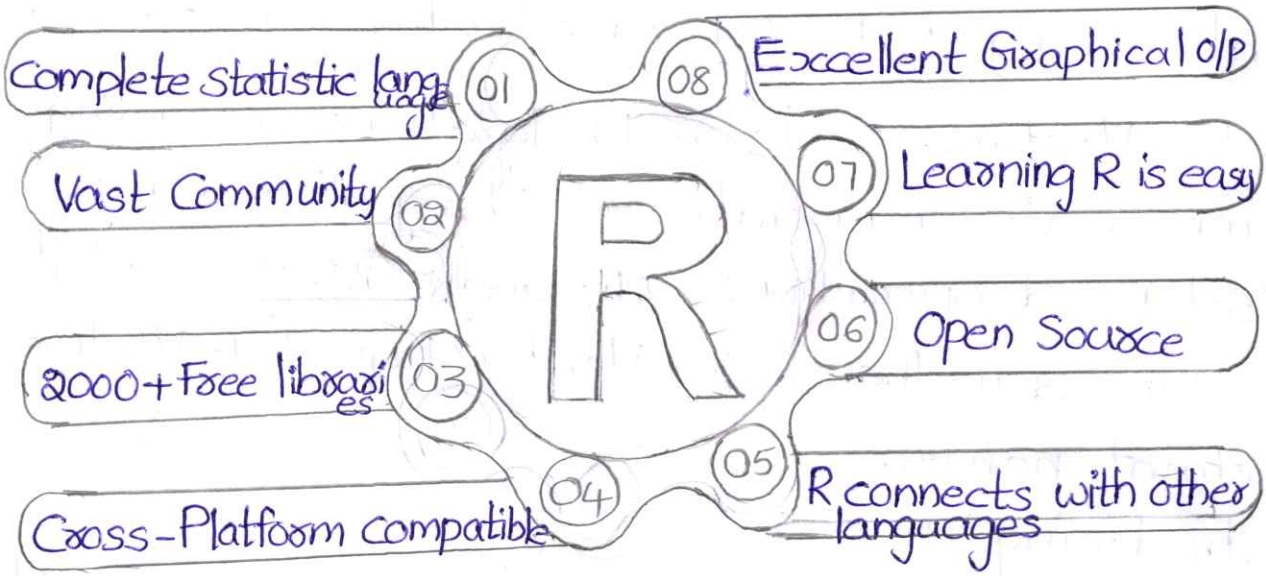
Introducing to R - R Data Structure - Help functions in R - Vectors - Scalars - Declarations - recycling - Common Vector operations - Using all and any - Vectorized operations - NA and NULL values - Filtering - Vectorised if-then else - Vector Equality - Vector Element names.

Introducing to R :

R is a programming language and Software environment that has become the first choice for statistical computing and data analysis. Developed in the early 1990s by Ross Ihaka and Robert Gentleman, R was built to Simplify complex data manipulation and Create clear, Customizable visualizations. Over time, it has gained popularity among statisticians, data Scientists and researches because of its capabilities and the vast array of packages available.

R programming language

As data-driven decision-making has grown, R has established itself as an important tool in various industries, including finance and health care, due to its ability to handle large datasets and perform in-depth statistical analysis.



Why Choose R Programming?

R is a unique language that offers a wide range of features for data analysis, making it an essential tool for professionals in various fields. Here's why R is preferred:

Free and Open-Source:

R is open to everyone, meaning users can modify, share and distribute their work freely.

Designed for Data:

R is built for data analysis, offering a Comprehensive set of tools for statistical Computing and graphics.

Large Package Repository:

offers thousands of add-on packages for specialized tasks.

Cross-Platform Compatibility:

R can work on Windows, Mac and Linux operating systems.

Great for Visualizations:

R makes it easy to create informative, interactive charts and plots.

key Features of R:

Cross-Platform Support:

R works on multiple operating systems, making it versatile for different environments.

Interactive Development:

R allows users to interactively experiment with data and see the results immediately.

Data Wrangling:

Tools like dplyr and tidyr help simplify data cleaning and transformation.

Statistical Modeling:

R has built-in support for various statistical models like regression, time-series analysis and clustering.

Reproducible Research:

With R Markdown, users can combine code, output and narrative in one document, ensuring their analysis is reproducible.

Example program in R:

To understand how R works, here's a basic example where we calculate the mean and standard deviation of a dataset.

We first create a vector data that contains numerical values.
We use the mean( ) function to calculate the mean of the dataset.
The sd( ) function calculates the standard deviation.

```
data ← c (5, 10, 15, 20, 25, 30, 35, 40, 45, 50).
mean_data <- mean (data)
print (paste ("Mean:", mean_data))
std_dev <- sd (data)
print (paste ("Standard Deviation :", std_dev))
```

Output: [1] "Mean:" 27.5"

[1] "Standard Deviation :15.138251770487 5"

Applications of R:
R is used in a variety of fields, including:
Data Science & Machine learning:
R is widely used for data analysis, statistical modeling and machine learning tasks.
Finance:
Financial analysts use R for quantitative modeling and risk analysis.
Healthcare:
In clinical research, R helps analyze medical data analysis and publishing reproduci -ble and text hypotheses.
Academia:
Researchers and statisticians use R for data analysis and publishing reproducible research.
Advantages of R programming:
Comprehensive Statistical tools:
R includes many statistical functions and models, making it the ideal choice for data analysis

Customizable Visualizations:

R's visualization tools allows for customizations for a simple bar chart or a detailed heatmap.

Extensive Community Support:

R has a large user base and there are countless resources, forums and tutorials available.

Highly Extendable:

The availability of over 15,000 R packages means we can extend R's functionality to suit any project or need.

Disadvantages of R programming:

Memory Intensive:

R can be slow with very large datasets, consuming a lot of memory.

Limited Support for Error Handling:

Unlike some other programming languages, R has less robust error handling features.

Steeper learning Curve:

Beginners might face challenges with some of R's complex features and syntax.

Performance:

R's performance can lag behind languages like python or C++ when it comes to speed, especially for large-scale operations.

In this article, we've seen how R is an tool for data analysis, statistical computing and visualization. It's open-source nature, comprehensive feature set and active community make it an excellent choice for both beginners and seasoned professionals.

R Data Structures:

R has several fundamental data structures, each with its own characteristics and uses. The most common include vectors, lists, matrices, data frames, and factors. These structures are essential for organizing and manipulating data in R for various statistical computations and data analysis tasks.

Here's a breakdown of the key data structures:

1. Vectors:

Definition: Vectors are the most basic data structure in R, capable of holding a sequence of elements of the same data type (eg: numeric, character, logical).

characteristics: They are one-dimensional and can be created using the c() function.

Example: my_vector <- c(10, 20, 30, 40, 50).

2. Lists:

Definition: Lists are versatile data structures that can contain elements of diffegent data types, including other lists, matrices, or data frames.

Characteristics: They are one-dimensional and can be created using the list() function.

Example: my_list <- list(name = "John", age = 30, hobbies = c("reading", "hiking")).

3. Matrices:

Definition: Matrices are two-dimensional, rectangular data structures where all elements must be of the same data type.

characteristics: They have rows and columns, and can be created using the matrix() function.

Example: my_matrix <- matrix (1:9, nrow = 3, ncol = 3)

4. DataFrames:

Definition:

Data frames are two-dimensional, tabular data structures where columns can contain different data types (like a matrix but with heterogeneous columns).

Characteristics:

They are often used to store datasets, and can be created using the data.frame() function.

Example:

my_data_frame <- data.frame (name = c ("Alice", "Bob"), age = c(25, 30), city = c("New York", "London"))

5. Arrays:

Definition:

Arrays are multi-dimensional data structures that can holds elements of the some data types.

Characteristics:

They extend matrices to more than two dimensions

Example:

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

6. Factors:

Definition:

Factors are used to represent categorical data, where the values can be chosen from a limited set of categories or levels.

Characteristics:

They are often used to store variables like gender, color, or country.

Example:

my_factor <- factor (c("red", "blue", "red", "green"), levels = c("red", "blue", "green", "yellow")).

Understanding these data structures is fundamental to working with R and effectively.

managing and analyzing data. Each structure offers unique capabilities for organizing and manipulating data based on its characteristics and the specific needs of the task at hand.

Help Functions in R:

The help() function in R provides access to the built-in documentation for functions, datasets, and other objects. It's the primary way to learn about R's functionality. You can use it by calling help(topic) or using the shorthand ?topic, where topic is the name of the function or object you're interested in.

Code

```
help(mean) # Displays the help page for the 'mean' function.
?plot # Displays the help page for the 'plot' function.
```

key features of the help() function:

Access to documentation:

It opens the help page in the R console or Rstudio's help pane.

Functionality:

You can use it to get help on functions, datasets, packages, and even operators.

Package context:

You can specify a package to get help on a function within that specific package.

Informative help pages:

R's help pages are structured with sections like "Description", "Usage", "Arguments", "Value" and "Examples".

Search functionality:
help.search() allows you to search for documenta
-tion matching a given character string.
Example Usage:
Help for a function: help(lm) or ?lm for the linear
model function.
Help for a package: help(package = "MASS") for the
MASS package.
Finding a function using search: help.search("linear
model")
Using examples: example(plot) to see examples of
how to use the plot function.
Getting help on operators: help("+") or help("<-")
(qotes are needed for operators).
Overview: Parameter Value:
The help() function takes the parameter value
function name which represents the name of any
R function.
Overview:
The help() function in R is used to get help on
any given R function passed to it.
Syntax:
help(function name)
Return value:
The help() function returns access to official
documentation pages of the function passed
to it.
Example:
In the code below, we'll use the help function to
get help on the following R functions:
eval() function
dump() function

Ex: implementing the help() function.

```
help(eval)
```

Run.

Explanation:

In the code above, we use the help() function to provide the official documentation page for the R function eval().

Now, lets use the help() function to get help on the dump(). In other words, we use the help() function to provide the official documentation page for the R function dump.

Ex: implementing the help() function.

```
help(dump)
```

Run.

Explanation:

In the code above, we use the help() function to provide help on the R function dump() by simply returning the R official documentation page for the dump() function.

Functions in R programming:

A function accepts input arguments and produces the output by executing valid R commands that are inside the function. Functions are useful when we want to perform a certain task multiple times.

In R programming language when we are creating a function name and the file in which we are creating the functions are useful need not be the same and we can have one or more functions in R.

Creating a Function in R programming:

Functions are Created in R by using the command function(). The general structure of the function file is a FOLLOWS.

```
f = function (arguments) {
        Statements
}
```

Here f = function name.

Parameters or Arguments in R-functions:

In programming, parameters and arguments refer to the values passed into a function. they are often used interchangeably, but there is a subtle difference:

Parameters are the variables defined in the function definition.

Arguments are the actual values passed to the function when it is called.

A function can have multiple parameters, and these are separated by commas within the paran these.

Example:

```
add_num <- function (a,b)
{
sum_result <- a+b
return (sum_result)
}
Sum = add_num (35,34)
  print (sum)
```

Output : [1] 69.

Function Parameter Rules:
Number of Parameters: A function should be called with the correct number of parameters. If the number desn't match, an error occurs.
Default Parameter Values: Some functions have default values for parameters. If no argument is passed, these defaults are used.
Return Value: the return() function sends the result back from the function.
Read More: R Function Parameters.
Calling a Function in R:
After creating a function, we have to call the function to use it. Calling a function in R is done by writing its name and passing possible parameters value.
Passing Arguments to Functions in R programming language:
There are Several ways we can pass the argum-ents to the function:
Case 1: Generally in R, the arguments are passed to the function in the same order as in the function defination.
Case 2: If we do not want to follow any order what we can do is we can pass the arguments using the names of the arguments in any order.
Case 3: If the arguments are not passed the default values are used to execute the function
Examples:

```
Rectangle = function (length = 5, width = 4){
    area = length * width
```

```
return (area)
}
```

1) Case 1 :

```
print (Rectangle (2,3))
```

2) Case 2 :

```
print (Rectangle (width = 8, length = 4))
```

3) Case 3 :

```
print (Rectangle ())
```

Output :

```
[1] 6
[1] 32
[1] 20.
```

Types of Function in R language :

Built-in Function : Built-in functions in R are pre-defined functions that are available in R programming languages to perform common tasks or operations.

User-defined function : R language allow us to write our own function.

1. Built-in Function in R Programming Language :

Built-in Function are the functions that are already existing in R language and we just need to call them to use.

Here we will use built-in functions like sum(), max() and min().

```
print (sum (4:6))
print (max (4:6))
print (min (4:6))
```

Output :

```
[1]
[1]
[1] 4.
```

Other Built-in Functions in R:

Let's look at the list of built-in R-functions and their uses:

Category	Function
Mathematical Functions.	abs(), sqrt(), round(), exp(), log(), cos(), sin(), tan().
statistical Functions	mean(), median(), cor(), var().
Data Manipulation Functions.	unique(), subset(), aggregate(), order().
File Input/output functions	unique(), subset(), read.csv(), write.csv(), read.table(), write.table().

2. User-defined Functions in R programming language.

User-defined functions are the functions that are created by the user. The user defines the working, parameters, default parameter, etc. of that user-defined function. They can be only used in the specific code.

```
evenOdd = function(x){
   if (x %% 2 == 0)
      return("even")
   else
      return("odd")
}
print(evenOdd(4))
print(evenOdd(3))
```

Output:

```
[1] "even"
[1] "odd"
```

R Function Examples:

Now let's look at some use cases of functions in R with some examples.

1. Single Input Single Output:

Create a function that takes a single input and returns a single output. For example, a function to calculate the area of a circle:

```
areaOfCircle = function(radius){
    area = pi * radius ^ 2
    return(area)
}
print(areaOfCircle(2))
```

Output:

```
[1] 12.56637
```

2. Multiple Input Multiple Output:

Create a function that takes multiple inputs and returns multiple outputs using a list. For example, a function to calculate the area and perimeter of a rectangle:

```
Rectangle = function(length, width){
    area = length * width
    perimeter = 2 * (length + width)
    result = list("Area" = area, "Perimeter" = perimeter)
    return(result)
}
resultList = Rectangle(2,3)
print(resultList["Area"])
print(resultList["Perimeter"])
```

Output:

```
$Area
[1] 6

$Perimeter.
[1] 10.
```

3. Inline Functions in R programming language:
For small, quick functions, use inline functions.
These are defined directly in the expression.

```
f=function(x)x^2*4+x/3.
print(f(4))
print(f(-2))
print(0)
```

Output:

```
[1] 65.33333
[1] 15.33333
[1] 0.
```

Lazy Evaluations of Functions in R programming language.
In R, functions are executed lazily, meaning that if some arguments are missing, the function still executes as long as those arguments are not involved in the execution. For example, Consider the following function Cylinder, which Calculates the volume of a cylinder using diameter and length. the argument radius, the function will still execute because it doesn't affect the volume calculation.

```
Cylinder=function(diameter, length, radius){
volume =pi* diameter^2 * length/4.
return(volume)
}
print(Cylinder(5,10))
```

Output:

```
[1] 196.3495.
```

If we do not pass the argument and then use it in the definition of the function it will

throw an errors that this "radius" is not passed and it is being used in the function definition.

Example:

```
Cylinder =function (diameter, length, radius) {
volume =pi* diameter^2* length/4.
print(radius)
return (volume)
}
 print(Cylinder(5,10))
```

Examples:

In R, the help() function (or the ? shortcut) provides access to the documentation for functions, data, and other objects. For vectors, you can use it to learn about vector operations functions like c(), length(), sort(), and more. To get help on vector operations, functions like c(), use help(c) or ?c for the c() function, or help(sort) or ? sort for the sort() function, for example.

Using the help() function:

1. Accessing documentation for a specific function:

To get help on a particular function, use the help() function with the function name as an argument. For example, help(mean) or ?mean will display the help page for the mean function.

2. Accessing documentation for a package:

To access the help documentation for an entire package, use help(package = "package_name"). For instance, help(package = "base") will.

show the help pages for the base package.

3. Searching the help System:

The help.search() function allows you to search the help system for topics related to a given character string. For example, help.search("vector") will search for documentation containing the word "vector".

Example for vectors:

help(c) or ?c : Provides information on the c() function, which is used to combine elements into a vector.

help(length) or ? length : Displays information about the length function, which returns the length of a vector.

help(sort) or ?sort : Explains the sort function, used to sort the elements of a vector.

help(seq) or ? seq : Gives details about the seq function, which is used to generate sequences of numbers.

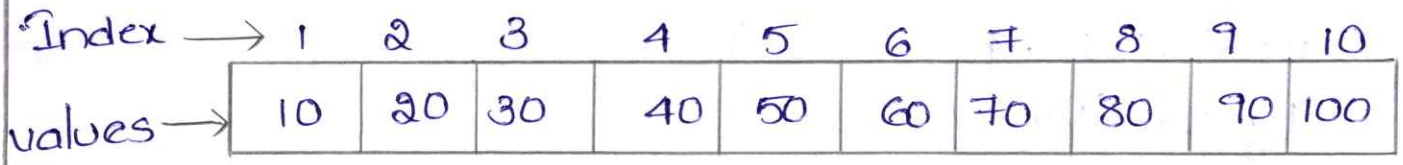
help(is.vector) or ? is.vector : Explains the is.vector function, used to check if a variable is a vector.

help(vector) or ?vector: provides information on the vector function, which is used to create vectors.

R-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.

Vectors in R



R-Vector.

1. Creating a vector in R:

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. We can also use seq() function or use colons ":" also as shown in the example.

```
X ← C(61, 4, 21, 67, 89, 2)
cat ('using c function', x, '\n')
Y ← seq (1, 10, length.out = 5)
cat ('using seq() function', Y, '\n')
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.

2. 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. The L suffix in R

is used to specify that a number is an integer and not a numeric (floating-point) value.

```
v1 <- c(4,5,6,7)
typeof(v1)
v2 <- c(1L,4L,2L,5L)
typeof(v2)
```

Output:

[1] "double"

[1] "integer".

Character vectors:

Character vectors in R contain alphanumeric values and special characters. In R, when a vector contains elements of mixed types (like characters and numbers), In R, when a vector contains elements of automatically coerces the entire vector to a single type. Since characters are more general than numbers in R, the vector is coerced to a character vector.

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

Output: [1] "character".

logical Vectors:

logical vectors in R contain Boolean values such as TRUE, FALSE and NA for Null values. In R, NA is a special value used to represent missing or undefined data. when used in a logical vector, NA is treated as a logical value because it is specifically designed to work with logical vectors and other types of data.

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

Output: [1] "logical".

3. 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.

```
x <- c(1,2,3,4,5)
length(x)
y <- c("apple","banana","cherry")
length(y)
z <- c(TRUE,FALSE,TRUE,TRUE)
length(z)
```

Output:

```
>
[1]
>
[1]
>
[1] 4.
```

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.

```
x <- c(2,5,18,1,12)
cat('Using Subscript operator',x[2],'\n')
y <- c(4,8,2,1,17)
Cat ('Using combine() function',y[c(4,1)],'\n').
```

Output:

```
Using Subscript operator 6
Using combine() function 1 4
```

5. 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 diffegent ways through which we can modify a vector:

```
X ← c (2,7,9,7,8,2)
X [3] ← 1
X [2] ← 9
Cat ('Subscript operator', X, '\n')
X [1:5] ← 0
 Cat ('Logical indexing', X, '\n')
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.

6. Deleting a R vector:

Deleting 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.

```
M ← C (8,10,2,5)
M ← NULL
print (cat ('Output vector', M, "\f"))
```

Output:

Output vector NULL.

7. Sorting elements of a R vector.

Sort( ) function is used with the help of which we can sort the values in ascending or descen -ding order.

```
x ← c (8,2,7,1,11,2)
A ← sort (x)
cat ('ascending order', A, '\n')
B ← sort (x, decreasing = TRUE)
cat ('descending order', B)
```

Output:

ascending order 1 2 2 7 8 11

decending order 11 8 7 2 2 1.

Scalars:

In R, the concept of a "scalar" is represented by a vector of length 1. R doesn't have a seperate Scalar data-type; instead, even single numeric or characteg values are treated as vectors of length one.

1. No Scalar Type:

Unlike some programming languages that have a distinct Scalar type, R treats single values as vectors.

2. Vectors of Length 1:

When you assign a single value (eg., a number, a characteg string) to a variable, R internally represents it as a vector containing only that single element.

Examples:

* x ← 10 (numeric Scalar/vector)
* Y ← "hello" (character scalar/vector)
* Z ← TRUE (logical scalar/vector)

Operations:

you can perform arithmetic and other operations on these "scalar" vectors, just as you would with larger vector.

Declaration:

Typically, compiled languages require that you declare variables; that is, warn the interpreter/compiler of the variables' existence before using

them. This is the case in our earlier C example:

```
int x;
int y[3];
```

As with most scripting languages (such as Python and Perl), you do not declare variables in R.

For instance, consider this code:

```
z<-3.
```

This code, with no pervious reference to z, is perfectly legal (and commonplace).

However, if you reference specific elements of a vector, you must warn R. For instance, say we wish y to be a two-component vector with values 5 and 12. The following will not work:

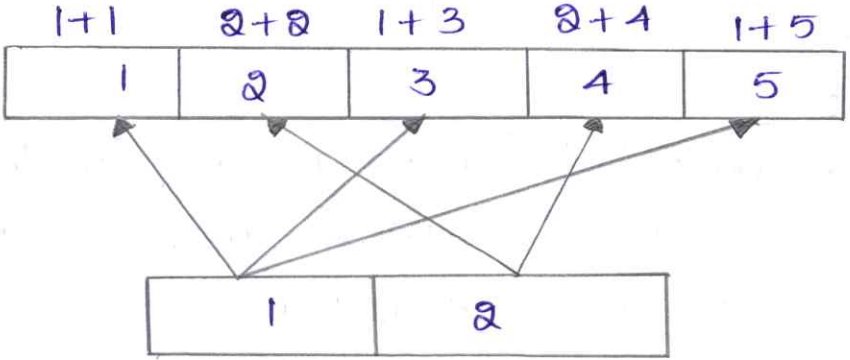
```
>y[1]<-5
>y[2]<-12
```

Instead, you must create y first, for instance this way:

```
>y<-vector(length=2)
>y[1]<-5
>y[2]<-12.
```

Vector Recycling in R:

When we perform some kind of operations like addition, subtraction.. etc on two vectors of unequal length. The vector with a small length will be repeated as long as the operation completes on the longer vector. If we perform an addition operation on a vector of equal length the first value of vector 1 is added with the first value of vector 2 like that. The below image demonstrated operation on unequal vectors operation on equal vector.



So, the repetition of small length vector as long as completion of operation as long length vector is known as vector recycling. This is the special property of vectors is available in R language. Let us see the implementation of vector recycling.

Example 1:

```
#Creating vector with
# 1 to 6 values
vec1 = 1:6
# Creating vector with 1:2
# values
vec2 = 1:2
# adding vector1 and vector2
 print(vec1+ vec2)
```

Output:

```
>vec1 = 1:6
> vec 2 = 1:2
> print (vec1 + vec2)
[1] 2 4 4 6 6 8.
```

In vector recycling, the length of the long length vector should be the multiple of the length of a small length vector. If not we will get a warning that longer object length is not a multiple of shorter object length. Here the longer object length is multiple of the shortest object length. so, we didn't get a warning message.

Example 2:

```
# Creating vector with 20
# to 25 values.
vec 1 = 20:25
# Creating vector with 4 to
# 6 values.
vec 2 = 4:6
# adding vector 1 and vecto2
 print(vec1 + vec2)
```

Output:

```
> vec1 = 20:25
>vec2 = 4:6
```

> print (vec1 + vec2)
[1] 24 26 28 27 29 31.

Common Vector Operations in R Programming:

Common Vector Operation in R include arithmetic operations (addition, Subtraction, multiplica -tion, division), accessing and modifying vector elements, Creating Sequences, concatenating vectors, and using vectorized operations for efficient computation.

Here's a more detailed breakdown:

1. Arithmetic Operations:

- Element-wise operations: R vectors support standard arithmetic operations like addition (+), Subtraction (-), multiplication (*), division (/), and exponentiation (^) directly between vectors of the same length.
- Scalar arithmetic: you can also perform arithmetic operations that can be applied elements between a vector and a single number (Scalar).
- Mathematical functions: R provides a range of mathe -matical functions that can applied element-wise to vectors, such as log(), exp(), sin(), cos(), tan(), and sqrt().
- Specialized functions: Functions like max(), min(), sum(), mean(), prod() operate on vectors to compute the maximum, minimum, sum, mean, or product of their elements, respectively.

2. Accessing and Modifying Vectors:

- Indexing: Individual elements of a vector can be accessed using square brackets [ ] and their index (position).
- Slicing: A range of elements can be extracted using a Sequence of indices within the brackets.
- Modifying: Individual elements or ranges of elements can be replaced with new values using assignment

With the ← operator and the appropriate index or slice.

3. Creating Vectors:

- c() function: The c() function is the most common way to create vectors in R by combining individual elements.
- : Operator: The : operator generates a sequence of integers.
- Seq() function: The Sequence() function creates sequences with more flexibility (eg., Specifying start, end, and increment).

4. Concatenating Vectors:

- c() function: The c() function can also be used to combine two or more vectors into a single larger vector.

5. Vectorized Operations:

- Efficiency: R excels at vectorized opeeptions, meaning that operations on vectors and performed element-wise without the need for explicit loops.
- Benefits:

Vectorization makes code elemen cleaner, more concise, and significantly faster for many computa-tions.

6. Other Useful Operations:

- length(): Returns the numbeq of elements in a vector.
- sort(): Sorts the elements of a vector in asending order.
- rev(): Reverses the order of elements in a vector.
- Unique(): Returns a vector with duplicate elements remov-ed.

Examples of Modification:

Modification Type	Example	Output.
change Single element	v[2] ← 25	V becomes: 10 25 30 40 5
change multiple elements	v[c(1,3)] ← c (15,35)	v becomes: 15 25 35 40 50
Modify using condition	v[v > 40] ← 100	V becomes: 15 25 35 40 100.

Examples:

1)-Arithmetic operations

```
v1 <- c(2,4,6,8)
v2 <- c(1,2,3,4)
# Element-wise Addition.
print(v1+v2) # output: 3 6 9 12.
# Element-wise Subtraction.
print(v1-v2) # output: 1 2 3 4.
# Element-wise Multiplication.
print(v1*v2) # output: 2 8 18 32.
# Element-wise Division.
print(v1/v2) # output: 2 2 2 2.
```

2)-Accessing Elements from a Vector.

```
v <- c(10,20,30,40,50)
Get elements greater than 25.
  print(v[v>25])
```

Output: [1] 30 40 50

Using all and any in R:

⇒ all() Returns TRUE all element of a logical vector are TRUE.

⇒ any() returns TRUE if atleast one element of a logical vector are TRUE.

Syntax:

all(x) TRUE if all(x==TRUE)

any(x) TRUE if atleast one (x=TRUE).

```
eg: v <- c(1,2,3,4)
      all(v>0)
output: [1] T T T T.
      any(v>0)
output: [1] T T T T.
eg: if (all(v>0))
    {
```

```
print f ("All positive")
}
if (any (v<0))
{ printf ("Atleast one positive").
}.
```

output: [1] "All positive".

Vectorized operations:

The R operation on vector performed element wise without explicit loops.

* It applies the R operation on each element of vector automatically, it is fast and efficient.
* There are four types of operations.

1. Arithmetic Operation:

eg: $v_1 \leftarrow c(1,2,3)$

$v_2 \leftarrow c(4,5,6)$.

$v_1 + v_2$

$v_1 - v_2$

$v_1 * v_2$

v_1 / v_2.

output: [1] 5 7 9

Output: [1] -3 -3 -3

Output: [1] 4 10 18

Output: [1] 0.25 0.40 0.50.

2. Scalar & Vector:

eg: $v \leftarrow c(2,4,6)$

$v + 10$

o/p: [1] 12 14 16

$v * 3$

o/p: [1] 6 12 18.

3. logical operation:

eg: v ← c(1, 5, 10)

v > 5

o/p: [1] FALSE FALSE TRUE

v ← c(5, 10)

v == 5

o/p: [1] TRUE FALSE

Mathematical functions:

eg: v ← c(0, pi/2, pi)

o/p: [1] 0.0000 2.30555, 4.607550.

Functions applied Element-wise:

eg: v1 ← c(1, 2, 3, 4)

v2 ← c(10, 20, 30, 40)

sin(v1).

sqrt(v2)

o/p: [1] 0.841 0.909 0.1411 0.7568

o/p: [1] 3.162 4.472 5.477 6.325.

Benefits:

1. Faster execution
2. Time Complexity
3. No loops are used.
4. Readability.

Without Vectorization:

eg: v ← c(1, 2, 3)

result ← c()

for (i in 1: length(v)) {

result[i] ← v[i] * 2.

}

result.

o/p: [1] 2 4 6.

NA and NULL values:

Features	NA	NULL.
Meaning	Missing / undefined value	Empty object.
Type	logical placeholder of length 1	An object of a class value.
length	length = 1	length = 0.
Usecase	When a value is missing in a vector dataframe.	To indicate the actions of an object or empty list element.
Example	c(1, 2, NA, 3)	NULL.

NA: missing value.

```
eg: x<-c (1, 2, NA, 4)
      is.na(x)
    Sum (is.na(x))
    x[is.na[x)] <-0
o/p: [1] F F T F
o/p: [1] length = 1.
```

NULL: Empty object.

```
eg: Y <- Null
      length(y) = 0
      is. NULL (Y) = TRUE.
o/p: [1] 0
       TRUE.
```

Key Differences:

1. NOT NULL is a placeholder for a value that exists

a list of missing.

2. NULL is used when a value or absence of an object or empty a list of element.

eg: x ← c (1,2, NA, 4)
print(x)

olp: [1] 1, 2, NA, 4.

eg: x ← c (1,2,NA,4)
mean(x)

olp: [1] 7/3 = 2.33.

Filtering:

Filtering involves typically refers to subsetting rows, data-frames based on some condition.

Syntax:

df [df $ age >20]

* logical condition inside uses a Square bracket [ ].

eg: v ← c (10,20,30,40, 50)
v [v >20]

olp: [1] 30 40 50
v [v == 20]

olp: [1] 20 30 40 50

eg: df ← data.frame = c ("Alice", "Bob", 'charel"),
age = c(25, 30, 55)
df [df $ age ≥ 30].

olp: [1] charel
35.

Vectorized if then-else:

If if else( ) function is used to create a vectorized condition statement it applies to test condition across each element of a vector and returns on a value for true element.

and another for false element. In operation applied element. element wise across entire vector of a array without need the explicit loop.

Syntax :

if_else (test, Yes, No)

test = logical vector.

Yes = value returns for true statement.

NO = value returns for false statement.

eg: number ← c (10, 15, 20, 25, 30)

result ← if else (number > 20, "Above 20", "below 30")

print (result)

olp : [1] 25

eg: number ← c (5, 10, NA, 20, 25)

result ← if else (is.NA (number), "missing")

if else (numbers < = 20, "20 below", "Above 20")

print (result)

olp : [1] 20.

Vector Equality :

In vector use == to compare each element of two vectors. Exactly equality vector or identi-cal in length type, values and Order. There are different ways depending on equality of vector.

1. Element Wise equality.
2. Set Equality.
3. Exact Equality.

1. Element Wise Equality :

* compares each element of one vector with the corresponding element of another vector.
* Returns a logical vector (TRUE / FALSE).

eg: v1 ← c (1, 2, 3)

v2 ← c (1, 4, 3)

v1 == v2.

o/p: [1] TRUE FALSE TRUE.

2. Set Equality:

* checks if two vectors contain the same elements, regardless of order and duplicates.
* Returns a single logical value (TRUE / FALSE)

eg: v1 ← c(1,2,3)

v2 ← c(3,2,1)

setequal(v1, v2)

o/p: [1] TRUE.

3. Exact Equality:

* checks if two vectors are exactly the same.
  → same elements
  → same order
  → same type.

eg: v1 ← c(1,2,3)

v2 ← c(3,2,1)

identical(v1, v2)

o/p: [1] FALSE.

Vector Element names:

In R, vector elements can be given names using the names() function. This allows for more descripti -ve access and manipulation of vector elements, rather than relying solely on their numerical indices.

1. Assigning Names:

* Use the names() function with the vector and a character vector of names as arguments. The length of the names vector must match the length of the vector being named.

Example : Creating a vector and naming its elements.

```
my_vector <- c(10, 20, 30, 40)
names(my_vector) <- c("a", "b", "c", "d")
print(my_vector)
#output :
# a b c d
# 10 20 30 40
```

2. Accessing Elements by Name :

* After naming the elements, you can access them using elements name within square brackets.

```
# Accessing elements by name.
print(my_vector["b"]) # output : 20.
print(my_vector[c("a", "c")]) # output : a c 10 30.
```

3. Removing Names :

* To remove names from a vector, assign NULL to the names() attribute.

```
# Removing names.
names(my_vector) <- NULL.
print(my_vector) # output : 10 20 30 40.
```

4. Creating Named Vectors Directly :

* you can also create vectors directly using the names() function during vector creation.

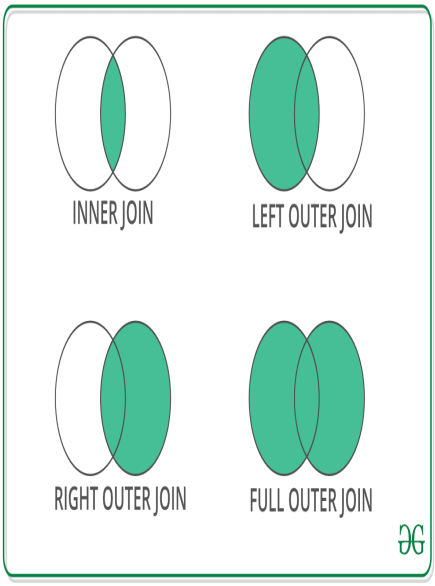
```
# Creating a named vector directly.
scores <- c(math = 95, Science = 88, english = 92)
print(scores)
# Output :
# math Science english.
# 95 88 92.
```

UNIT-III

Merging DataFrame

Merging DataFrames in Pandas is similar to performing SQL joins. It is useful when we need to combine two DataFrames based on a common column or index. The merge() function provides flexibility for different types of joins.

There are four basic ways to handle the join (inner, left, right and outer) depending on which rows must retain their data.



1. Merging DataFrames Using One Key

We can merge DataFrames based on a common column by using the on argument. This allows us to combine the DataFrames where values in a specific column match.

```
import pandas as pd


data1 = {'key': ['K0', 'K1', 'K2', 'K3'],
        'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
       'Age':[27, 24, 22, 32],}


data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
        'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
       'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}
```

```
df = pd.DataFrame(data1)

df1 = pd.DataFrame(data2)

print(df, "\n\n", df1)
```

Output:

```
  key    Name  Age
0  K0     Jai   27
1  K1  Princi   24
2  K2  Gaurav   22
3  K3    Anuj   32

   key    Address Qualification
0  K0     Nagpur         Btech
1  K1     Kanpur           B.A
2  K2  Allahabad          Bcom
3  K3    Kannuaj        B.hons
```

Now here we are using .merge() with one unique key combination.

```
res = pd.merge(df, df1, on='key')

res
```

Output:

	key	Name	Age	Address	Qualification
0	K0	Jai	27	Nagpur	Btech
1	K1	Princi	24	Kanpur	B.A
2	K2	Gaurav	22	Allahabad	Bcom
3	K3	Anuj	32	Kannuaj	B.hons

2. Merging DataFrames Using Multiple Keys

We can also merge DataFrames based on more than one column by passing a list of column names to the on argument.

```
import pandas as pd

data1 = {'key': ['K0', 'K1', 'K2', 'K3'],

        'key1': ['K0', 'K1', 'K0', 'K1'],

        'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

       'Age':[27, 24, 22, 32],}
```

```
data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
        'key1': ['K0', 'K0', 'K0', 'K0'],
        'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
       'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}
df = pd.DataFrame(data1)
df1 = pd.DataFrame(data2)
print(df, "\n\n", df1)
```

```
  key key1    Name  Age
0  K0   K0     Jai   27
1  K1   K1  Princi   24
2  K2   K0  Gaurav   22
3  K3   K1    Anuj   32

   key key1    Address Qualification
0  K0   K0     Nagpur         Btech
1  K1   K0     Kanpur           B.A
2  K2   K0  Allahabad          Bcom
3  K3   K0    Kannuaj        B.hons
```

Now we merge dataframe using multiple keys.

```
res1 = pd.merge(df, df1, on=['key', 'key1'])
res1
```

Output

	key	key1	Name	Age	Address	Qualification
0	K0	K0	Jai	27	Nagpur	Btech
1	K2	K0	Gaurav	22	Allahabad	Bcom

3. Merging DataFrames Using the how Argument

We use how argument to merge specifies how to find which keys are to be included in the resulting table. If a key combination does not appear in either the left or right tables, the values in the joined table will be NA. Here is a summary of the how options and their SQL equivalent names:

MERGE METHOD	JOIN NAME	DESCRIPTION

MERGE METHOD	JOIN NAME	DESCRIPTION
left	LEFT OUTER JOIN	Use keys from left frame only
right	RIGHT OUTER JOIN	Use keys from right frame only
outer	FULL OUTER JOIN	Use union of keys from both frames
inner	INNER JOIN	Use intersection of keys from both frames

```
import pandas as pd
data1 = {'key': ['K0', 'K1', 'K2', 'K3'],
        'key1': ['K0', 'K1', 'K0', 'K1'],
        'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
       'Age':[27, 24, 22, 32],}
data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
        'key1': ['K0', 'K0', 'K0', 'K0'],
        'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
       'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}
df = pd.DataFrame(data1)
df1 = pd.DataFrame(data2)
print(df, "\n\n", df1)
```

Output:

```
   key key1    Name  Age
0  K0   K0     Jai   27
1  K1   K1  Princi   24
2  K2   K0  Gaurav   22
3  K3   K1    Anuj   32

   key key1    Address Qualification
0  K0   K0     Nagpur         Btech
1  K1   K0     Kanpur           B.A
2  K2   K0  Allahabad          Bcom
3  K3   K0    Kannuaj        B.hons
```

Now we set how = 'left' in order to use keys from left frame only. In this it includes all rows from the left DataFrame and only matching rows from the right.

res = pd.merge(df, df1, how='left', on=['key', 'key1'])

res

Output:

	key	key1	Name	Age	Address	Qualification
0	K0	K0	Jai	27	Nagpur	Btech
1	K1	K1	Princi	24	NaN	NaN
2	K2	K0	Gaurav	22	Allahabad	Bcom
3	K3	K1	Anuj	32	NaN	NaN

Now we set how = 'right' in order to use keys from right frame only. In this it includes all rows from the right DataFrame and only matching rows from the left.

res1 = pd.merge(df, df1, how='right', on=['key', 'key1'])

res1

Output

	key	key1	Name	Age	Address	Qualification
0	K0	K0	Jai	27.0	Nagpur	Btech
1	K1	K0	NaN	NaN	Kanpur	B.A
2	K2	K0	Gaurav	22.0	Allahabad	Bcom
3	K3	K0	NaN	NaN	Kannuaj	B.hons

Now we set how = 'outer' in order to get union of keys from dataframes. In this it combines all rows from both DataFrames, filling missing values with NaN.

res2 = pd.merge(df, df1, how='outer', on=['key', 'key1'])

res2

Output:

	key	key1	Name	Age	Address	Qualification
0	K0	K0	Jai	27.0	Nagpur	Btech
1	K1	K0	NaN	NaN	Kanpur	B.A
2	K1	K1	Princi	24.0	NaN	NaN
3	K2	K0	Gaurav	22.0	Allahabad	Bcom
4	K3	K0	NaN	NaN	Kannuaj	B.hons
5	K3	K1	Anuj	32.0	NaN	NaN

Now we set how = 'inner' in order to get intersection of keys from dataframes. In this it only includes rows where there is a match in both DataFrames.

```
res3 = pd.merge(df, df1, how='inner', on=['key', 'key1'])

res3
```

Output:

	key	key1	Name	Age	Address	Qualification
0	K0	K0	Jai	27	Nagpur	Btech
1	K2	K0	Gaurav	22	Allahabad	Bcom

Joining DataFrame

The .join() method in Pandas is used to combine columns of two DataFrames based on their indexes. It's a simple way of merging two DataFrames when the relationship between them is primarily based on their row indexes. It is used when we want to combine DataFrames along their indexes rather than specific columns.

1. Joining DataFrames Using .join()

If both DataFrames have the same index, we can use the .join() function to combine their columns. This method is useful when we want to merge DataFrames based on their row indexes rather than columns.

```
import pandas as pd

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
    'Age':[27, 24, 22, 32]}

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
    'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

df = pd.DataFrame(data1,index=['K0', 'K1', 'K2', 'K3'])

df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])

print(df, "\n\n", df1)
```

```
       Name  Age
K0      Jai   27
K1   Princi   24
K2   Gaurav   22
K3     Anuj   32

        Address Qualification
K0   Allahabad           MCA
K2     Kannuaj           Phd
K3   Allahabad          Bcom
K4     Kannuaj       B.hons
```

Now we are using .join() method in order to join dataframes

res = df.join(df1)

res

	Name	Age	Address	Qualification
K0	Jai	27	Allahabad	MCA
K1	Princi	24	NaN	NaN
K2	Gaurav	22	Kannuaj	Phd
K3	Anuj	32	Allahabad	Bcom

Now we use how = 'outer' in order to get union

res1 = df.join(df1, how='outer')

res1

	Name	Age	Address	Qualification
K0	Jai	27.0	Allahabad	MCA
K1	Princi	24.0	NaN	NaN
K2	Gaurav	22.0	Kannuaj	Phd
K3	Anuj	32.0	Allahabad	Bcom
K4	NaN	NaN	Kannuaj	B.hons

2. Joining DataFrames Using the "on" Argument

If we want to join DataFrames based on a column (rather than the index), we can use the on argument. This allows us to specify which column(s) should be used to align the two DataFrames.

```
import pandas as pd

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
        'Age':[27, 24, 22, 32],
        'Key':['K0', 'K1', 'K2', 'K3']}

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
        'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}
```

df = pd.DataFrame(data1)

df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])

print(df, "\n\n", df1)

Output:

```
     Name  Age Key
0     Jai   27  K0
1  Princi   24  K1
2  Gaurav   22  K2
3    Anuj   32  K3

       Address Qualification
K0  Allahabad           MCA
K2    Kannuaj           Phd
K3  Allahabad          Bcom
K4    Kannuaj        B.hons
```

Now we are using .join with “on” argument.

res2 = df.join(df1, on='Key')

res2

Output:

	Name	Age	Key	Address	Qualification
0	Jai	27	K0	Allahabad	MCA
1	Princi	24	K1	NaN	NaN
2	Gaurav	22	K2	Kannuaj	Phd
3	Anuj	32	K3	Allahabad	Bcom

3. Joining DataFrames with Different Index Levels (Multi-Index)

In some cases, we may be working with DataFrames that have multi-level indexes. The .join() function also supports joining DataFrames that have different index levels by specifying the index levels.

import pandas as pd

data1 = {'Name':['Jai', 'Princi', 'Gaurav'],

'Age':[27, 24, 22]}

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kanpur']

```
 'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}
df = pd.DataFrame(data1, index=pd.Index(['K0', 'K1', 'K2'], name='key'))
index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
                                   ('K2', 'Y2'), ('K2', 'Y3')],
                                   names=['key', 'Y'])
df1 = pd.DataFrame(data2, index= index)
print(df, "\n\n", df1)
```

Output:

```
       Name  Age
key
K0      Jai   27
K1   Princi   24
K2   Gaurav   22

           Address Qualification
key Y
K0  Y0  Allahabad           MCA
K1  Y1    Kannuaj           Phd
K2  Y2  Allahabad          Bcom
    Y3     Kanpur        B.hons
```

Now we join singly indexed dataframe with multi-indexed dataframe.

```
result = df.join(df1, how='inner')
result
```

Output:

		Name	Age	Address	Qualification
key	**Y**				
K0	**Y0**	Jai	27	Allahabad	MCA
K1	**Y1**	Princi	24	Kannuaj	Phd
K2	**Y2**	Gaurav	22	Allahabad	Bcom
	Y3	Gaurav	22	Kanpur	B.hons

Applying Functions to Data Frames

As with lists, you can use the lapply and sapply functions with data frames.

Using lapply() and sapply() on Data Frames

The data frames are special cases of lists, with the list components consisting of the data frame's columns. Thus, if you call lapply() on a data frame with a specified function f(), then f() will be called on each of the frame's columns, with the return values placed in a list.

For instance, with our previous example, we can use lapply as follows:

```
> d
  kids ages
1 Jack   12
2 Jill   10
> dl <- lapply(d,sort)
> dl
$kids
[1] "Jack" "Jill"
$ages
[1] 10 12
```

1. Using apply()

The apply() function works mainly with matrices, but it also works with data frames (which are internally lists of equal-length vectors).

Syntax:

apply(dataframe, MARGIN, function)

MARGIN = 1 → apply function row-wise

MARGIN = 2 → apply function column-wise

Example:

```
df <- data.frame(
  A = c(1, 2, 3),
  B = c(4, 5, 6),
  C = c(7, 8, 9)
)
# Column-wise sum
apply(df, 2, sum)

# Row-wise mean
apply(df, 1, mean)
```

2. Using lapply()

lapply() applies a function to each column of the data frame and returns a list.

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

◈ 3. Using sapply()

sapply() is similar to lapply(), but it simplifies the result (vector or matrix if possible).

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

◈ 4. Using mapply()

mapply() applies a function to multiple columns element-wise.

```
df$A <- c(1,2,3)
df$B <- c(4,5,6)

# Add corresponding elements of A and B
mapply(sum, df$A, df$B)
```

◈ 5. Using dplyr (Tidyverse approach)

If you're using dplyr, you can apply functions easily with mutate(), summarise(), and across().

```
library(dplyr)

# Apply mean to all columns
df %>% summarise(across(everything(), mean))

# Apply log() to all numeric columns
df %>% mutate(across(where(is.numeric), log))
```

Apply function to every value in R dataframe

In R Programming Language to apply a function to every integer type value in a data frame, we can use lapply function from dplyr package. And if the datatype of values is string then we can use paste() with lapply. Let's understand the problem with the help of an example.

Dataset in use:

	A	B	C	D
1.	1	8	21	4
2.	9	2	0	6
3.	6	3	14	3
4.	5	6	5	7
5.	9	4	3	1
6.	6	3	2	3

after applying value*7+1 to each value of the dataframe

Expected result:

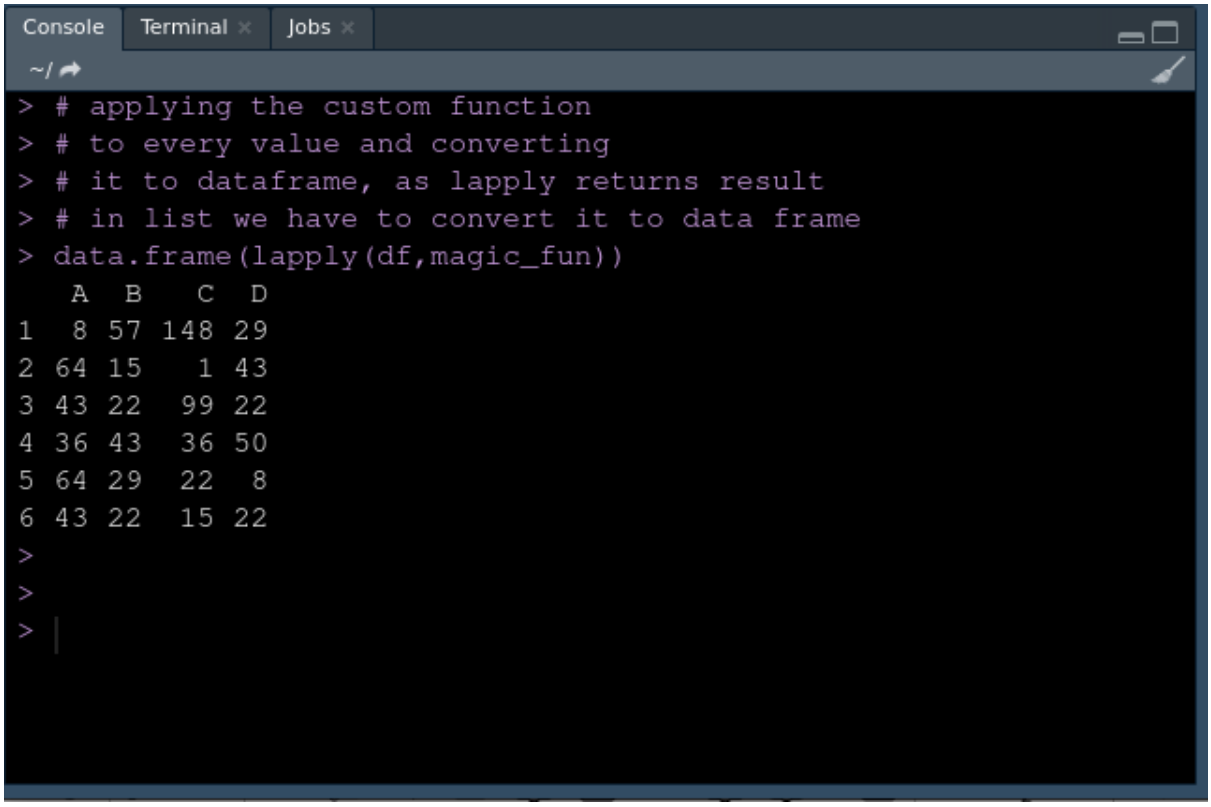
	A	B	C	D
1.	8	57	148	29

	A	B	C	D
2.	64	15	1	43
3.	43	22	99	22
4.	36	43	36	50
5.	64	29	22	8
6.	43	22	15	22

Method 1 : Using lapply function:
lapply is a function from apply family. By using lapply, we can avoid for loop as for loop is slower than lapply. lapply works faster than a normal loop because it doesn't mess with the environment you work in. It returns output as a list. 'l' in lapply indicates list.
Syntax:
lapply(X, FUN, ...)
Here, X can be a vector list or data frame. And FUN takes a function that you wish to apply to the data frame as an argument.
Approach:
Create a dummy dataset.
Create a custom function that you want to apply to every value in the data frame.
Apply this custom function to every value in the data frame with the help of lapply.
Display result
Example
 # Apply function to every value in dataframe

```
# Creating dataset
m <- c(1,9,6,5,9,6)
n <- c(8,2,3,6,4,3)
o <- c(21,0,14,5,3,2)
p <- c(4,6,3,7,1,3)
# creating dataframe
df <- data.frame(A=m,B=n,C=o,D=p)
# creating function
# that will multiply
# each value by 7 and then add 1
magic_fun <- function(x){
  return (x*7+1)}
# applying the custom function to every value and converting
# it to dataframe, as lapply returns result in list
# we have to convert it to data frame
data.frame(lapply(df,magic_fun))
```

Output :



Using lapply

Method 2 : Using paste and apply function:

paste() takes an R object as an argument and converts it to characters then paste it back with another string, i.e.it converts the argument to the character string and concatenates them.

Syntax:

paste (..., sep = " ")

Our R object which is to be converted to string goes in place of *"...", sep=" "* represents a character string to separate the terms.

Approach:

Create a dummy dataset.

Apply the custom function which will print *"Hello,"* then value in the data frame value

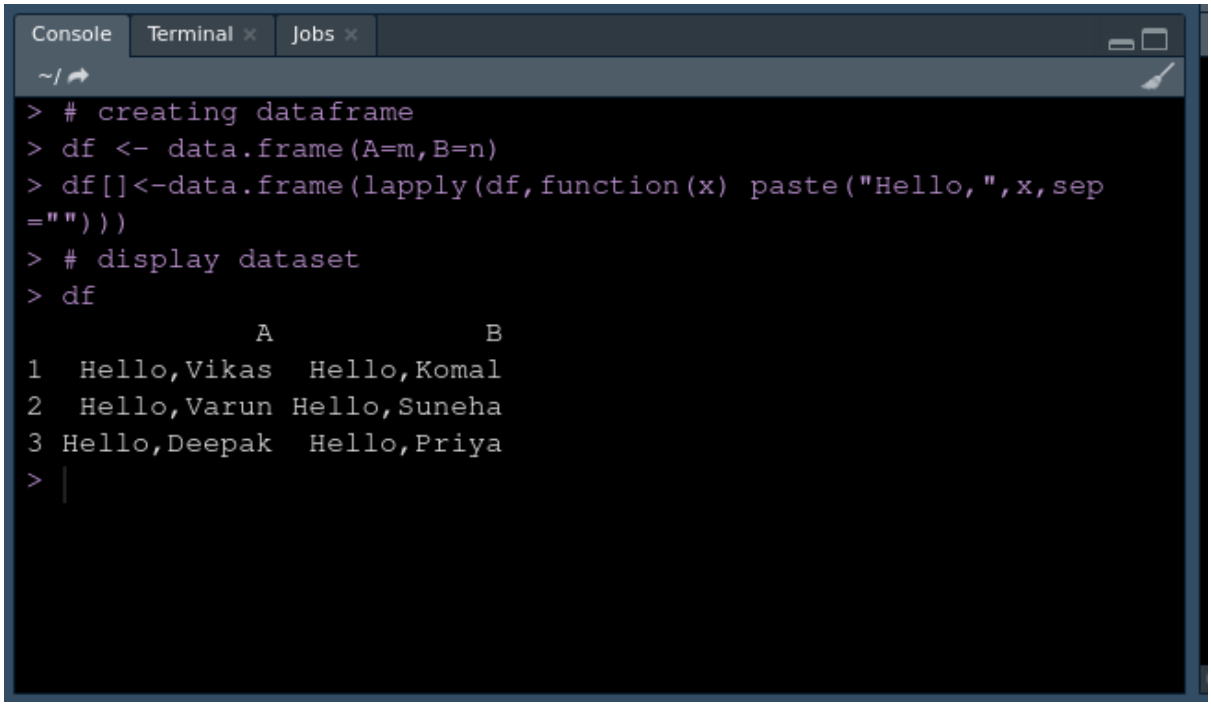
Display result

Example:

```
# Apply function to every value in dataframe
# Creating dataset
m <- c("Vikas","Varun","Deepak")

n <- c("Komal","Suneha","Priya")
# creating dataframe
df <- data.frame(A=m,B=n)
# Applying custom function to every element in dataframe
df[]<-data.frame(lapply(df,function(x) paste("Hello,",x,sep="")))
# display dataset
df
```

Output:



Using paste and apply

Method 3: Using purrr

purrr is a functional programming toolkit. Which comes with many useful functions such as a map. The map() function iterates across all entries of the vector and returns the output as a list. It allows us to replace for loop with in the code and makes it easier to read.

Syntax :

map(.x, .f) returns a list

map_df(.x, .f) returns a data frame

map_dbl(.x, .f) returns a numeric (double) vector

map_chr(.x, .f) returns a character vector

map_lgl(.x, .f) returns a logical vector

Here, .x is input and .f is a function that you want to be applied. Input to map function can be a list, a vector, or a data frame.

Note: You need to install purrr package explicitly using the following command.

install.packages("purrr")

Approach :

Create a vector, a list and a data frame

Create a custom function which you want to be applied.

Use map() to apply custom function on vector, list and data frame.

Display result

Factors in R

Factors in R are data structures used to represent categorical data. They are essentially integer vectors where each integer is assigned a label, known as a "level." This allows for efficient storage and manipulation of categorical variables.

Key characteristics of factors:

- Categorical Data:

Factors are designed for variables with a limited and predefined set of possible values (eg, "Male," "Female" for gender, or "Red," "Green," "Blue" for color).

- Levels:

The distinct values a factor can take are called its levels. By default, R sorts these levels alphabetically.

- Storage:

Although they appear as character strings, factors are stored internally as integers, which optimizes memory usage, especially with repeating values.

- Statistical Modeling:

Factors are crucial in statistical modeling functions like lm()and glm(), as they correctly represent categorical variables for analysis.

- Ordered vs. Unordered:

Factors can be either ordered (eg, "Low," "Medium," "High") or unordered (eg, "North," "South," "East," "West").

Creating a factor:

Code

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

Factors

Factors in R are data structures used to represent categorical variables. They are essentially integer vectors with associated labels (levels) that correspond to the unique categories.

- Purpose: Factors are crucial for statistical analysis and modeling functions (e.g., lm(), glm()) as they correctly handle categorical variables, ensuring appropriate statistical tests and model interpretations.
- Creation: Factors are created using the factor() function.

```
# Create a character vector
gender_vector <- c("male", "female", "male", "female", "male")

# Convert to a factor
gender_factor <- factor(gender_vector)
print(gender_factor)
```

- Levels:

Factors have predefined levels, which are the distinct categories. By default, levels are sorted alphabetically. The levels() function can be used to view or modify the order of levels.

- Ordered Factors:

Factors can be ordered, meaning there's an inherent order among the categories (e.g., "low", "medium", "high"). This is specified using the ordered = TRUE argument in the factor() function.

Tables

Tables in R, typically created using the table() function, are used to generate frequency distributions of one or more categorical variables (factors).

- Purpose:

Tables provide a concise summary of how many observations fall into each category or combination of categories. They are essential for understanding data distributions and exploring relationships between categorical variables.

- Creation:

The table() function takes one or more factors (or variables that can be coerced to factors) as arguments.

```
# Create a factor
color_factor <- factor(c("red", "blue", "red", "green", "blue"))
```

```
# Create a frequency table
color_table <- table(color_factor)
print(color_table)

# Create a two-way contingency table
gender_color_table <- table(gender_factor, color_factor)
print(gender_color_table)
```

- Contingency Tables:

When table() is used with multiple factors, it creates a multi-dimensional contingency table, showing the joint frequencies of the categories.

- Analysis:

Tables are often the starting point for further statistical analysis, such as chi-squared tests for independence using chisq.test().

Tables in R

Tables in R, often created using the table()function, are used to generate frequency distributions or cross-tabulations of categorical variables, including factors. They provide a summary of the counts of occurrences for each unique value or combination of values.
Key characteristics of tables:

- Frequency Distributions:

A single factor can be used to create a simple frequency table showing the count of each level.

- Cross-Tabulations (Contingency Tables):

Multiple factors can be used to create a two-way (or multi-way) cross-tabulation, displaying the joint frequencies of combinations of levels.

- Output:

The table()function returns a "table" object, which is a specialized form of vector or matrix designed for displaying frequencies.

Creating a table:

Code

```
# Using the factor created above
gender_table <- table(gender)

# Creating a cross-tabulation with another factor (e.g., 'education_level')
education_level <- factor(c("High School", "College", "High School", "College", "Graduate"))
cross_table <- table(gender, education_level)
```

Factors and levels

In R programming, factors are data structures used to represent categorical data. They are essentially integer vectors where each integer is associated with a label, known as a level. Factors are particularly useful for statistical modeling and plotting, as R's statistical functions often treat factors differently than other data types like character or numeric vectors.
Here's a breakdown of factors and levels:
Factors:

- Purpose:

Factors are designed to handle categorical data, which are variables that can take on a limited number of distinct values or categories. Examples include gender (Male, Female), education level (High School, Bachelor's, Master's), or product quality (Good, Average, Bad).

- Storage:

While they display as labels (e.g., "Male", "Female"), factors are internally stored as integers, with each integer mapping to a specific level. This makes them memory-efficient compared to storing character strings directly.

- Types:

Factors can be ordered or unordered.

  - Unordered factors: The levels have no inherent order (e.g., "Red", "Green", "Blue").
  - Ordered factors: The levels have a meaningful order (e.g., "Low" < "Medium" < "High"). This is important for analyses that depend on the order of categories.

- Creation:

Factors are created using the factor() function.

Levels:

- Definition:

Levels are the distinct, predefined categories or values that a factor can take. They are the labels associated with the internal integer representation of the factor.

- Default Order:

By default, R sorts factor levels alphabetically when a factor is created without explicitly specifying the levels argument in the factor() function.

- Custom Order:

You can specify the order of levels when creating a factor using the levels argument in the factor() function. This is crucial for ordered factors or when you want a specific display order.

- Accessing Levels:

The levels of a factor can be accessed and manipulated using the levels() function.

Example:

కోడ్

```
# Create a character vector
colors_vector <- c("Red", "Blue", "Green", "Red", "Blue")

# Convert to an unordered factor
colors_factor_unordered <- factor(colors_vector)
print(colors_factor_unordered)
# Output: [1] Red   Blue  Green Red   Blue
# Levels: Blue Green Red (default alphabetical order)

# Convert to an ordered factor with custom levels
satisfaction_vector <- c("Medium", "High", "Low", "Medium")
satisfaction_factor_ordered <- factor(satisfaction_vector,
                                      levels = c("Low", "Medium", "High"),
                                      ordered = TRUE)
print(satisfaction_factor_ordered)
# Output: [1] Medium High   Low    Medium
# Levels: Low < Medium < High (custom order)
```

Level Ordering of Factors in R Programming

Last Updated : 12 Jul, 2025

Level ordering controls how categorical values are stored, displayed, and interpreted in analyses and plots. By default, R orders factor levels alphabetically. In this article, we will see the level ordering of factors in the R Programming Language.

What Are Factors in R?

Factors are data objects used to categorize data and store it as levels. They can store a string as well as an integer. They represent columns as they have a limited number of unique values. Factors in R can be created using the factor() function. It takes a vector as input. c() function is used to create a vector with explicitly provided values.

Example:

In this example, x is a vector with 8 elements. To convert it to a factor the function factor() is used. Here, there are 8 factors and 3 levels. Levels are the unique elements in the data. It can be found using the levels() function

```
x <- c("Pen", "Pencil", "Brush", "Pen",
       "Brush", "Brush", "Pencil", "Pencil")

print(x)
print(is.factor(x))

# Apply the factor function.
factor_x = factor(x)
levels(factor_x)
```

Output :

```
[1] "Pen"    "Pencil" "Brush"  "Pen"    "Brush"  "Brush"  "Pencil" "Pencil"

[1] FALSE

[1] "Brush"  "Pen"    "Pencil"
```

Ordering Factor Levels

Ordered factors levels are an extension of factors. It arranges the levels in increasing order. We use two functions: factor() and argument ordered().

1. Using the factor() Function

You can rearrange factor levels by actually stating the order in the levels argument to the factor() function. This is perfect if you have an established, predetermined order for your factor levels.

Syntax:

factor(data, levels =c(""), ordered =TRUE)

Parameter:

- data: input vector with explicitly defined values.
- levels(): Mention the list of levels in c function.
- ordered: It is set true for enabling ordering.

Example: In this example, the size vector is created using the c function. Then it is converted to a factor. For the ordering factor, the factor() function is used along with the arguments described as "Orderd=TRUE" . Thus the sizes are arranged in order.

```
size = c("small", "large", "large", "small",
         "medium", "large", "medium", "medium")

size_factor <- factor(size)
```

```
print(size_factor)

ordered.size <- factor(size, levels = c(
  "small", "medium", "large"), ordered = TRUE)
print(ordered.size)
```
Output:
```
[1] small  large  large  small  medium large  medium medium
Levels: large medium small

[1] small  large  large  small  medium large  medium medium
Levels: small < medium < large
```
2. Using the ordered() Function

The same can be done using the ordered function.

Example:
```
sizes <- factor(c("small", "large", "large",
                  "small", "medium"))

# ordering the levels
sizes <- ordered(sizes, levels = c("small", "medium", "large"))
print(sizes)
```
Output:
```
[1] small  large  large  small  medium
Levels: small < medium < large
```
Level ordering visualization in R

To visualise the level ordering , we have a dataset of student grades, and we want to create a boxplot to compare the distribution of grades for different class levels (freshman, sophomore, junior, and senior). We can create a factor variable to represent the class levels and specify the level ordering so that the boxplot is ordered by class level.

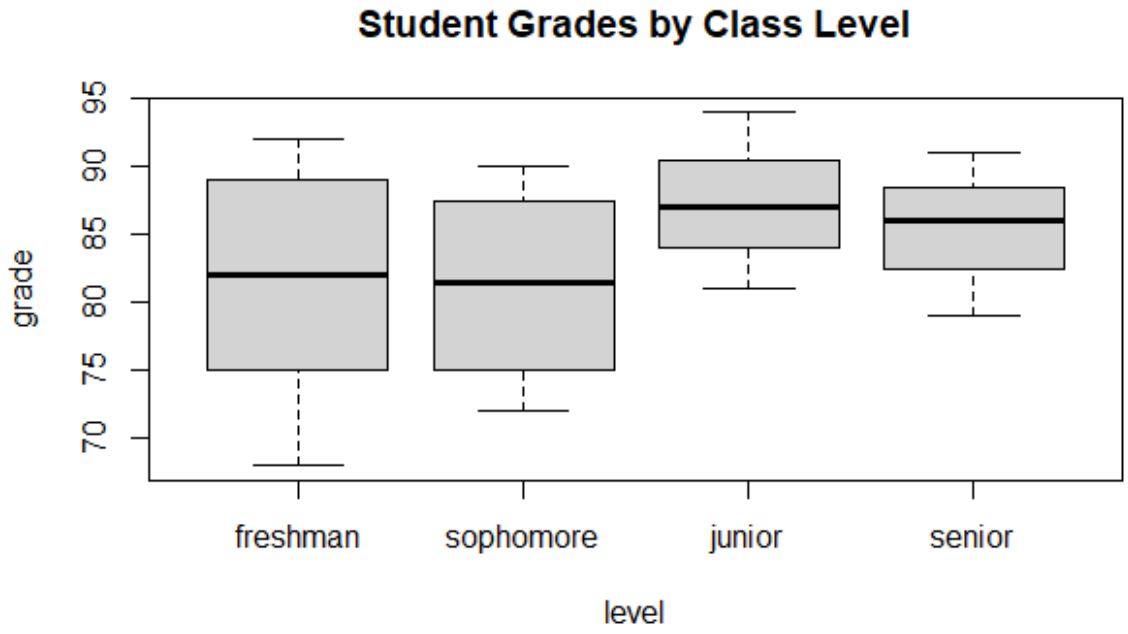
Example:

In this example, we create a sample dataset of student grades with a grade column and a level column representing the class level of each student. We then create a factor variable level from the level column and specify the level ordering as "freshman", "sophomore", "junior", and "senior" using the factor() function.

Finally, we create a boxplot of grades by class level using the boxplot() function. The grade column represents the response variable, and the level column represents the explanatory variable. We also specify a title for the plot.
```
grades <- data.frame(
  grade = c(75, 82, 68, 92, 89, 78, 85, 90, 72, 81, 94, 87, 79, 86, 91),
  level = factor(c(rep("freshman", 5), rep("sophomore", 4), rep("junior", 3), rep("senior", 3)))
)

grades$level <- factor(grades$level, levels = c("freshman", "sophomore", "junior", "senior"))

boxplot(grade ~ level, data = grades, main = "Student Grades by Class Level")
```
Output:



Level Ordering of Factors in R Programming

Examples

Factor Example

```
# Factor for fruits
fruits <- factor(c("Apple", "Mango", "Apple", "Banana", "Mango", "Apple"))

print(fruits)         # prints factor values
print(levels(fruits))  # shows unique categories
```

Output:

```
[1] Apple  Mango  Apple  Banana Mango  Apple
Levels: Apple Banana Mango
[1] "Apple"  "Banana" "Mango"
```

◈ 2. Table Example (One-way)

```
# Frequency of fruits
table(fruits)
```

Output:

```
fruits
 Apple Banana  Mango
     3      1      2
```

◈ 3. Two-way Table

```
# Factors for gender and preference
gender <- factor(c("Male", "Female", "Male", "Female", "Female", "Male"))
drink  <- factor(c("Tea", "Coffee", "Coffee", "Tea", "Tea", "Coffee"))

# Cross tabulation
table(gender, drink)
```

Output:

```
        drink
```

```
gender   Coffee Tea
  Female      1   2
  Male        2   1
```

◈ 4. Using prop.table() (Proportions)

```
# Proportion instead of counts
prop.table(table(gender, drink))
```

Output (as fractions of total 6):

```
        drink
gender     Coffee       Tea
  Female 0.1666667 0.3333333
  Male   0.3333333 0.1666667
```

◈ 5. Three-way Table

```
age_group <- factor(c("Young", "Young", "Old", "Young", "Old", "Old"))

# 3D table (gender x drink x age)
table(gender, drink, age_group)
```

Output (simplified view):

```
, , age_group = Old

        drink
gender   Coffee Tea
  Female      0   1
  Male        2   0
age_group = Young
drink
gender   Coffee Tea
  Female      1   1
  Male        0   1
```

- Factors = categories (Apple, Mango, Banana, …)
- Tables = count how often each category (or combination) appears

Common functions used with factors in R include:

Creation and Identification:

- factor(): Converts a vector into a factor.
- as.factor(): Converts an object to a factor.
- is.factor(): Checks if an object is a factor, returning TRUE or FALSE.
- ordered(): Creates an ordered factor, where levels have a specific order.
- is.ordered(): Checks if a factor is ordered.
- gl(): Generates factor levels, useful for creating factors with a specific number of levels and replications.

Accessing and Modifying Properties:

- levels(): Retrieves or sets the levels of a factor.
- nlevels(): Returns the number of levels in a factor.
- length(): Returns the number of elements in a factor.

Data Manipulation and Analysis:

- table(): Creates a frequency table of factor levels, showing the count of each level.

- summary(): Provides a summary of the factor, including frequency counts for each level.
- as.character(): Converts a factor back to a character vector.
- as.numeric(): Converts a factor to its underlying numeric codes.
  forcats Package (for advanced factor manipulation):
- fct_reorder(): Reorders factor levels based on another variable.
- fct_recode(): Recodes factor levels, changing their names.
- fct_collapse(): Collapses multiple factor levels into a single level.

Working with tables in R involves various operations, including creating, inspecting, manipulating, and presenting tabular data.

1. Creating Tables:

- Frequency Tables: The table() function is used to create frequency tables for categorical variables.

కోడ్

```
vec <- c(2, 4, 3, 1, 2, 3, 2, 1, 4, 2)
table(vec)
```

- Contingency Tables (Cross-tabulations): table() can also be used with multiple variables to create contingency tables, showing the relationship between categories.

కోడ్

```
df <- data.frame(
  "Name" = c("abc", "cde", "def"),
  "Gender" = c("Male", "Female", "Male")
)
table(df$Name, df$Gender)
```

- From Scratch: You can create a table from a matrix using as.table().

కోడ్

```
mat <- matrix(c(10, 20, 30, 40), nrow = 2, byrow = TRUE)
my_table <- as.table(mat)
```

2. Inspecting Tables:

- head(): Displays the first few rows of a table or data frame.

కోడ్

```
head(my_data_frame)
```

- str(): Provides a summary of the structure of the data, including data types and dimensions.

కోడ్

```
str(my_data_frame)
```

3. Manipulating Tables (often using dplyr or base R):

- Filtering Rows: Select rows based on conditions.

కోడ్

```
# Using dplyr
library(dplyr)
filtered_data <- my_data_frame %>% filter(column_name > value)
# Using base R
filtered_data_base <- my_data_frame[my_data_frame$column_name > value, ]
```

- Selecting Columns: Choose specific columns of interest.

కోడ్

```
# Using dplyr
selected_columns <- my_data_frame %>% select(col1, col2)
```

```
# Using base R
selected_columns_base <- my_data_frame[, c("col1", "col2")]
```

- Summarizing Data: Calculate summaries like mean, median, sum, etc.

కోడ్

```
# Using dplyr
summary_data <- my_data_frame %>% summarize(mean_col = mean(column_name))
```

- Grouping Data: Perform operations within groups defined by one or more categorical variables.

కోడ్

```
# Using dplyr
grouped_summary <- my_data_frame %>% group_by(category_col) %>%
summarize(mean_col = mean(value_col))
```

- Adding New Columns: Create new columns based on existing ones.

కోడ్

```
# Using dplyr
new_column_data <- my_data_frame %>% mutate(new_col = col1 * col2)
# Using base R
my_data_frame$new_col <- my_data_frame$col1 * my_data_frame$col2
```

4. Presenting Tables:

- Packages like gt, kableExtra, reactable, DT: These packages offer advanced functionalities for creating aesthetically pleasing and interactive tables for reports, web applications, or presentations. They allow for custom styling, formatting, and interactive features.

Working with Factors

- Creating Factors:

Use factor() to create a factor from a vector or to define a specific order for the factor's levels using ordered=TRUE or the ordered() function.

- Inspecting Factors:
  - levels(my_factor): Displays the different categories or levels of the factor.
  - summary(my_factor): Shows the frequency (count) of each level.
  - is.factor(my_variable): Checks if a variable is a factor.
- Converting Factors:
  - as.numeric(my_factor): Converts a factor to a numeric vector representing the underlying integer codes of the levels.
  - as.character(my_factor): Converts a factor to a character vector, which preserves the level names.

2. Working with Tables (Frequency Tables)

- Creating Tables:
  - table(my_factor): Creates a frequency table of factor levels, showing the counts for each category.
  - table(factor1, factor2, factor3): Creates a multi-dimensional contingency table for three or more categorical variables.
  - ftable(my_multi_dim_table): Prints multi-dimensional tables in a more compact, "flat" format for better readability.
- Modifying Tables:
  - prop.table(my_table): Converts a frequency table into a table of proportions.
  - margin.table(my_table): Calculates the marginal frequencies (total counts) for rows or columns of a table.
- Statistical Tests with Tables:

    - chisq.test(my_table): Performs a Chi-Square test for independence on a two-dimensional table.

3. Other Related Functions

- Applying Functions to Groups:
    - tapply(x, f, FUN): Applies a function (FUN) to subsets of a vector (x) defined by a factor (f).
    - aggregate(x ~ f, data=my_data, FUN=mean): A more general way to summarize data in a data frame by grouping with factors, often used for calculating means, sums, etc., within each group.
- Visualizing Tables:
    - barplot(table(my_factor)): Creates a bar plot from the frequency table of a factor, visualizing the counts of each category.

other factors and tables related functions

When designing or managing any table (document, spreadsheet, or database):

1. Data Integrity
    - Use validation rules or constraints to avoid duplicates and invalid data.
2. Normalization / Structure
    - Especially in databases: split repeated info into separate tables with relationships.
3. Performance & Size
    - For large data sets, plan indexes (SQL) or filters (Excel/Pandas) for faster queries.
4. Security & Access
    - Control who can edit or view sensitive rows/columns.
5. Consistency
    - Stick to a single data type per column (e.g., don't mix text and numbers).
6. Documentation & Metadata
    - Add captions, comments, or a data dictionary so others know what each field means.

🛠 Table-Related Functions by Environment

📊 Spreadsheet (Excel / Google Sheets)

Task	Common Functions
Math	`SUM()`, `AVERAGE()`, `COUNT()`, `ROUND()`
Conditional	`IF()`, `IFS()`, `COUNTIF()`, `SUMIF()`
Lookup	`VLOOKUP()`, `HLOOKUP()`, `XLOOKUP()`, `INDEX()` + `MATCH()`
Table Ops	Insert → Table, structured refs like `=Table1[Column]`, `SORT()`, `FILTER()`
Text	`CONCAT()`, `TEXT()`, `LEFT()`, `RIGHT()`, `TRIM()`

🗄 SQL Databases

Action	Key Statements / Functions
Create/Modify	`CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`
Insert/Update	`INSERT INTO`, `UPDATE`, `DELETE`
Retrieve	`SELECT`, `WHERE`, `ORDER BY`, `JOIN`
Aggregate	`COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN()`
Constraints	`PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `CHECK`

🌐 HTML/CSS

- Tags: `<table>`, `<thead>`, `<tbody>`, `<tr>`, `<th>`, `<td>`, `<caption>`.
- CSS helpers: `border-collapse`, `padding`, `nth-child(even)` for zebra stripes.

Control statements

Control statements in R programming are used to manage the flow of execution within a program, allowing for decision-making and repetitive actions based on specific conditions.

1. Conditional Statements:

- if statement: Executes a block of code only if a specified condition is TRUE.

కోడ్

```
x <- 10
if (x > 5) {
  print("x is greater than 5")
}
```

- if-else statement: Executes one block of code if the condition is TRUE and another if it's FALSE.

కోడ్

```
x <- 3
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

- if-else if-else statement: Tests multiple conditions sequentially, executing the block corresponding to the first TRUE condition.

కోడ్

```
grade <- 85
if (grade >= 90) {
  print("A")
} else if (grade >= 80) {
  print("B")
} else {
  print("C")
}
```

- switch statement: Evaluates an expression and executes code based on its value, providing a cleaner alternative to nested if-else if for multiple choices.

కోడ్
```
day <- "Monday"
switch(day,
       "Monday" = print("Start of the week"),
       "Friday" = print("End of the week"),
       print("Mid-week")
)
```
2. Looping Statements:
   - for loop: Iterates over a sequence (e.g., a vector, list, or range of numbers) and executes a block of code for each element.

కోడ్
```
for (i in 1:5) {
  print(i)
}
```
   - while loop: Continuously executes a block of code as long as a specified condition remains TRUE.

కోడ్
```
count <- 1
while (count <= 3) {
  print(count)
  count <- count + 1
}
```
   - repeat loop: Executes a block of code indefinitely until explicitly terminated by a break statement.

కోడ్
```
x <- 1
repeat {
  print(x)
  x <- x + 1
  if (x > 3) {
    break
  }
}
```
3. Jump Statements (within loops):
   - break statement: Terminates the current loop and transfers control to the statement immediately following the loop.
   - next statement: Skips the current iteration of a loop and proceeds to the next iteration.

Arithmetic and boolean operators and values in r programming

R programming language utilizes various operators for arithmetic calculations and logical (Boolean) operations.

Arithmetic Operators:

These operators perform mathematical computations:

- +: Addition
- -: Subtraction
- *: Multiplication
- /: Division
- ^ or **: Exponentiation (raises the first operand to the power of the second)
- %%: Modulus (returns the remainder of a division)
- %/%: Integer Division (returns the quotient of a division, discarding the remainder)

Boolean Operators and Values:

Boolean operators perform logical comparisons and
return TRUE or FALSE values. These TRUE and FALSE are the Boolean values in R.
Relational Operators (return Boolean values):

- <: Less than
- <=: Less than or equal to
- >: Greater than
- >=: Greater than or equal to
- ==: Equal to
- !=: Not equal to

Logical Operators (work with Boolean values):

- &: Element-wise Logical AND. Returns TRUE if both corresponding elements are TRUE.
- |: Element-wise Logical OR. Returns TRUE if at least one of the corresponding elements is TRUE.
- !: Logical NOT. Inverts the Boolean value (e.g., !TRUE is FALSE).
- &&: Logical AND (scalar). Evaluates only the first element of each operand, returning a single TRUE or FALSE.
- ||: Logical OR (scalar). Evaluates only the first element of each operand, returning a single TRUE or FALSE.

Examples:

```
# Arithmetic Operators
a <- 10
b <- 3
print(a + b)  # Output: 13
print(a - b)  # Output: 7
print(a * b)  # Output: 30
print(a / b)  # Output: 3.333333
print(a ^ b)  # Output: 1000
print(a %% b) # Output: 1 (remainder of 10/3)
print(a %/% b) # Output: 3 (integer part of 10/3)

# Boolean Operators and Values
x <- 5
y <- 7
print(x < y)   # Output: TRUE
print(x == y)  # Output: FALSE
print(x != y)  # Output: TRUE

vec1 <- c(TRUE, FALSE, TRUE)
vec2 <- c(FALSE, TRUE, TRUE)
print(vec1 & vec2) # Output: FALSE TRUE TRUE (element-wise AND)
print(vec1 | vec2) # Output: TRUE TRUE TRUE (element-wise OR)
print(!vec1)       # Output: FALSE TRUE FALSE (logical NOT)

print(TRUE && FALSE) # Output: FALSE (scalar AND)
print(TRUE || FALSE) # Output: TRUE (scalar OR)
```

Default values of arguments

Default arguments provide predefined values for function parameters that are automatically assigned if a user doesn't provide a value when calling the

function. This allows for flexible function calls, as you can omit certain arguments and still have the function work correctly using the default values. Default arguments are typically defined with a specific syntax (like = in Python and C++) and are placed after any non-default arguments in the function definition.

How Default Arguments Work

1. Function Definition:

When defining a function, you assign a default value to a parameter using syntax like parameter_name = default_value.

2. Function Call:
    - If you provide a value for that parameter in the function call, the default value is overridden.
    - If you do not provide a value for the parameter, the function automatically uses the assigned default value.

Example

Consider a Python function create_user(name, country="USA"):

- create_user("John") will result in a user named "John" from the "USA", as "USA" is the default value for country.
- create_user("Jane", "Canada") will result in a user named "Jane" from "Canada", as the explicitly provided "Canada" overrides the default.

Benefits

- Flexibility: Allows functions to be called with fewer arguments, making them more versatile.
- Simplicity: Reduces the need for separate functions with slightly different parameter lists.
- Readability: Clearly indicates optional values that are often used.

Key Considerations

- Order:

Default arguments must come after any non-default arguments in the function definition.

- Scope:

In languages like JavaScript, default parameters are evaluated at the time of the function call, not when the function is defined.

Returning Boolean values

To return a Boolean value, a function uses conditional logic (like if statements) or comparison operators to evaluate a condition and returns True or False (or 1 or 0 in some languages like C) based on the result. In many programming languages, functions that return a Boolean are often called "predicate" functions and are good practice to name starting with "is", such as isEven() or isValid().

1. Using Conditional Statements

You can use if/else statements to check conditions and explicitly return True or False. Python Example.

Python

```
def is_positive(number):
    if number > 0:
        return True
    else:
        return False
```

This function checks if a number is positive, returning True if it is, and False otherwise. JavaScript Example.

JavaScript

```
function isPositive(number) {
    if (number > 0) {
        return true;
    } else {
        return false;
    }
}
```

2. Using Comparison Operators

You can directly return the result of a comparison, as comparison operators inherently produce a Boolean value. Python Example.

Python

```
def is_positive(number):
    return number > 0 # Directly returns the result of the comparison
```

JavaScript Example.

JavaScript

```
function isPositive(number) {
    return number > 0; // The '>' operator returns a boolean directly
}
```

3. Best Practices

- Naming Conventions:

Name your Boolean-returning functions with a prefix like "is", "has", or "can" to clearly indicate that they ask a question and answer with a True/False value.

  - def is_valid_user(username)
  - function has_permission(user)

- Predicates:

Functions that return a Boolean value are often called "predicate functions".

4. Contexts where Boolean values are used

- Conditional Statements: To control the flow of your program, like if and else statements.
- Logical Operators: To combine or negate conditions.
- Checking for Success or Failure: To indicate if an operation was successful or not, such as checking if a file was found or a database record exists.

  Functions and objects

In programming, an object is a fundamental concept representing a collection of data (properties) and behaviors (methods) that can interact with each other. A function is a block of reusable code that performs a specific task, and in many languages like JavaScript, functions themselves are a special type of object, often called "function objects" or "first-class objects". This means they can be treated as values, assigned to variables, passed to other functions, and have properties and methods like other objects, but with the unique ability to be "called" or executed.

Objects

- Definition: A real-world or abstract entity modeled in a program.
- Structure: A collection of properties (key-value pairs), where the value can be any data type, including another object or a function (which then becomes a method).
- Examples: A "car" object might have properties like color and model, and a method like startEngine().

Functions

- Definition: A self-contained block of code designed to perform a specific task.
- Purpose: To provide reusable logic, allowing you to avoid repeating code and organize your program into logical, manageable units.
- Key Distinction: Unlike regular objects, functions are designed to be invoked or "called" to execute their defined actions.

The Relationship: Functions are Objects (in JavaScript)

- First-Class Objects:

In languages like JavaScript, functions are "first-class citizens" or "first-class objects". This means they possess all the characteristics of other objects:

- They can be assigned to variables.
- They can be stored in data structures like arrays.
- They can be passed as arguments to other functions (callback functions).
- They can be returned from other functions.

- Function-Specific Properties:

As special objects, function objects have built-in properties and methods that regular objects don't. Examples include name, length, arguments, call, apply, and bind.

- Methods vs. Functions:

When a function is associated with an object and performs an action on or for that object, it is often called a method. In JavaScript, all methods are functions, but the term highlights their role within an object's structure.

Tools for composing functions code

Composing function code in R involves creating and structuring functions to perform specific tasks, often combining them to build more complex operations. Several tools and concepts facilitate this process:

1. Integrated Development Environments (IDEs):
   - RStudio: The most widely used IDE for R, providing a comprehensive environment with a code editor, console, workspace viewer, plot viewer, and integrated tools for package development, version control (Git), and R Markdown reporting.
2. Core R Language Features:
   - function() keyword: Used to define functions, specifying parameters and the function body.

కోడ్

```
my_function <- function(arg1, arg2) {
  # Function body
  result <- arg1 + arg2
  return(result)
}
```

- Arguments: Inputs passed to functions, allowing for flexible and reusable code.
- Return values: Functions can return results using the return() statement or by implicitly returning the last evaluated expression.
- Scoping: Understanding how variables are accessed within functions (lexical scoping) is crucial for correct function composition.

3. Functional Programming Concepts:

- Anonymous functions:

Functions defined without a name, often used for concise operations within other functions or functionals.

- Functionals:

Functions that take other functions as arguments
(e.g., lapply, sapply, purrr::map).

- Closures:

Functions created by other functions, retaining access to variables from their enclosing environment.

- Function operators:

Functions that take functions as input and return modified functions as output.

4. Packages for Enhanced Function Composition:

- magrittr (and the base R pipe |>): Provides the pipe operator (%>% or |>) for chaining function calls, improving readability and reducing nested function calls.

కోడ్

```
library(magrittr)
data %>%
  filter(condition) %>%
  mutate(new_variable = old_variable * 2) %>%
  summarise(mean_value = mean(new_variable))
```

- dplyr:

A key package for data manipulation, offering functions that are easily composable with the pipe operator for common data transformation tasks.

- purrr:

Part of the Tidyverse, it provides a consistent and powerful set of tools for working with functions and iterations, including mapping functions over lists and vectors.

- devtools and roxygen2:

Essential for developing and documenting R packages, which often involve composing multiple functions into a cohesive unit.

5. Version Control:
   - Git and GitHub: Crucial for managing function code, tracking changes, collaborating with others, and ensuring reproducibility. RStudio offers excellent integration with Git.

These tools and concepts collectively empower R users to write modular, reusable, and maintainable function code, facilitating complex data analysis and application development.

Environment and scope issues

Environment and scope issues include broad environmental problems such as climate change, biodiversity loss, and pollution, alongside specific concerns like waste management, resource depletion, and land degradation. The "scope" refers to the broad extent and nature of these issues, which are often interconnected and global in nature, impacting everyone from local ecosystems to the entire planet.

Major Environmental Issues

- Climate Change:

Caused by global warming from fossil fuels, this includes rising sea levels and extreme weather events.

- Pollution:

This encompasses air pollution from industrial and vehicular sources, water pollution from various contaminants, plastic pollution, and soil degradation.

- Biodiversity Loss:

The loss of plant and animal species and the destruction of natural habitats due to deforestation and urbanization.

- Waste Management:

Issues like poor management of solid waste, plastic, and hazardous materials contribute to pollution and environmental degradation.

- Resource Depletion:

Growing demand from a rising global population leads to the depletion of land, water, and forest resources.

Scope of Environmental Issues

- Global Impact:

Environmental problems are not confined to specific regions but affect the entire world, such as climate change and plastic pollution.

- Interconnectedness:

Issues like deforestation, biodiversity loss, and climate change are deeply linked, with actions in one area affecting others.

- Regional Variations:

While global in scale, specific issues like water scarcity, land degradation, and pollution have varying impacts and challenges across different regions.

- Socio-Economic Factors:

Environmental challenges are often driven by development and human activities, putting immense pressure on natural resources.

Environmental Studies & Management

- Multidisciplinary Approach:

Environmental studies integrate various fields, including biology, chemistry, geology, engineering, sociology, and economics, to understand and address these complex issues.

- Public Awareness:

Educating the public about environmental problems fosters sensitivity, encourages resource conservation, and promotes active participation in solutions.

- Prevention and Management:

Environmental studies and management focus on preventing pollution, conserving biodiversity, and managing natural resources sustainably.

Writing Upstairs

Writing upstairs" in the context of R programming refers to the practice of creating and managing R scripts and functions in a structured manner, often in separate files, rather than directly typing and executing code in the R console. This approach promotes organization, reusability, and easier debugging.

Here's how to implement this in R: Create a New R Script.

In RStudio, navigate to File > New File > R Script to open a new, blank script file. In other environments, you can use a text editor to create a new file and save it with a .R extension (e.g., my_script.R). Write Your R Code.

Populate the R script with your code, including:

- Functions: Define custom functions to encapsulate specific tasks.
- Variable Assignments: Store data or results in variables.
- Data Manipulation: Perform operations on your data.
- Comments: Use the # symbol to add comments that explain your code and its purpose, improving readability.

కోడ్

```
# This is a sample R script for data analysis

# Define a function to calculate the mean
calculate_mean <- function(data_vector) {
  mean_value <- mean(data_vector)
  return(mean_value)
}

# Create a sample data vector
my_data <- c(10, 15, 20, 25, 30)

# Calculate the mean using the defined function
result_mean <- calculate_mean(my_data)

# Print the result
print(paste("The mean of the data is:", result_mean))
```

Save the R Script.

Save the script file to a designated location on your computer. This allows you to easily access and reuse your code in the future.

- Execute the Script:
    - In RStudio: You can execute individual lines or selected blocks of code by placing the cursor on the line/block and pressing Ctrl + Enter (or Cmd + Enter on macOS). To run the entire script, use the "Source" button or Ctrl + Shift + S.
    - From the R Console: Use the source() function to execute the entire script file. For example:

కోడ్

```
source("path/to/your/my_script.R")
```

This method of "writing upstairs" promotes good programming practices in R by separating code logic from interactive console commands, facilitating project organization and collaboration.

Recursion

Recursion in R, as in other programming languages, is a technique where a function calls itself to solve a problem. This approach breaks down a complex problem into smaller, self-similar subproblems until a simple base case is reached, which can be solved directly without further recursion.

Key Components of a Recursive Function in R:

- Base Case:

This is the condition that stops the recursion. Without a proper base case, a recursive function would call itself indefinitely, leading to a stack overflow error.

- Recursive Step:

This is the part of the function where it calls itself with a modified input, typically moving closer to the base case.

Example: Factorial Calculation using Recursion in R

The factorial of a non-negative integer n is the product of all positive integers less than or equal to n. It can be defined recursively as:

- 0! = 1 (Base case)
- n! = n * (n-1)! for n > 0 (Recursive step)

Code

```
factorial_recursive <- function(n) {
  # Base case
  if (n == 0 || n == 1) {
    return(1)
  } else {
    # Recursive step
    return(n * factorial_recursive(n - 1))
  }
}

# Example usage
print(factorial_recursive(5))
# Output: 120
print(factorial_recursive(0))
# Output: 1
```

Advantages of Recursion:

- Elegance and Readability:

For problems that naturally have a recursive structure (like tree traversals or certain mathematical sequences), recursive solutions can be more concise and easier to understand than iterative ones.

- Problem Decomposition:

Recursion excels at breaking down complex problems into smaller, manageable subproblems.

Disadvantages and Considerations:

- Performance:

Recursive calls involve overhead due to function call stack management, which can make them slower than iterative solutions for some problems, especially in R where function calls can be relatively expensive.

- Memory Usage:

Each recursive call adds a new frame to the call stack. Deep recursion can lead to excessive memory consumption and potentially stack overflow errors.

- Debugging:

Tracing the execution flow of a recursive function can be more challenging than with iterative loops.

Replacement functions:

In R, replacement functions are used to modify elements within vectors, lists, or data frames. The primary function for general replacement is replace(), but other functions like gsub() and direct indexing can also be used for specific replacement tasks.

1. **The replace() function:**

The replace() function is a versatile tool for replacing values at specified indices or based on conditions.

Code

```
replace(x, list, values)
```

- x: The vector or object in which values are to be replaced.
- list: An index vector specifying the positions of elements to be replaced. This can be a numeric vector of indices or a logical vector.
- values: The replacement values. These values are recycled if list has more elements than values.

Example:

Code

```
# Replacing elements by index
my_vector <- c("apple", "orange", "grape", "banana")
new_vector <- replace(my_vector, 2, "blueberry")
print(new_vector)
# Output: [1] "apple"     "blueberry" "grape"     "banana"

# Replacing elements based on a condition
numbers <- c(1, 2, 4, 4, 5, 7)
updated_numbers <- replace(numbers, numbers > 4, 50)
print(updated_numbers)
# Output: [1] 1 2 4 4 50 50
```

2. **Direct Indexing for Replacement:**

You can directly assign new values to specific elements or subsets of an object using indexing.

Code

```
# Replacing a single element
my_vector[2] <- "kiwi"
print(my_vector)
# Output: [1] "apple" "kiwi"  "grape" "banana"

# Replacing multiple elements using a logical condition
numbers[numbers == 4] <- 99
print(numbers)
# Output: [1] 1 2 99 99 50 50
```

3**. gsub() and sub() for String Replacement:**

For replacing patterns within character strings, gsub() (global substitution) and sub() (single substitution) are used.

Code

```
# Global substitution
text <- "This is a test string. This test is repeated."
new_text_gsub <- gsub("test", "sample", text)
print(new_text_gsub)
# Output: [1] "This is a sample string. This sample is repeated."

# Single substitution
new_text_sub <- sub("test", "sample", text)
print(new_text_sub)
# Output: [1] "This is a sample string. This test is repeated."
```

Tools for string manipulation functions in r

R provides several tools for string manipulation, primarily through base R functions and dedicated packages.

1. Base R Functions:
   - paste() and paste0(): Concatenate strings. paste0() is a shortcut for paste(..., sep = "").

Code

```
paste("Hello", "World") # "Hello World"
paste0("Hello", "World") # "HelloWorld"
```

   - nchar(): Count the number of characters in a string.

Code

```
nchar("R Programming") # 13
```

   - toupper() and tolower(): Convert strings to uppercase or lowercase.

Code

```
toupper("hello") # "HELLO"
tolower("WORLD") # "world"
```

   - substr(): Extract a substring from a specified start and end position.

Code

```
substr("R Programming", 1, 3) # "R P"
```

   - strsplit(): Split a string into a vector of substrings based on a delimiter.

Code

```
strsplit("apple,banana,orange", ",") # list("apple", "banana",
"orange")
```

   - grep(), grepl(), sub(), gsub(): Used for pattern matching and replacement, often with regular expressions.
     - grep(): Returns the indices of elements matching a pattern.
     - grepl(): Returns a logical vector indicating matches.

- sub(): Replaces the first occurrence of a pattern.
- gsub(): Replaces all occurrences of a pattern.

Code

```
grep("a", c("apple", "banana")) # 1 2
gsub("a", "X", "banana") # "bXnXnX"
```

2. stringr Package:

The stringr package, part of the Tidyverse, offers a consistent and user-friendly interface for string manipulation. Its functions typically start with str_.

- str_c(): Concatenate strings (similar to paste0()).
- str_length(): Get the length of strings (similar to nchar()).
- str_to_upper(), str_to_lower(), str_to_title(), str_to_sentence(): Case conversion.
- str_sub(): Extract substrings (similar to substr()).
- str_split(): Split strings (similar to strsplit()).
- str_detect(), str_subset(), str_count(), str_extract(), str_replace(): Powerful functions for pattern matching, extraction, and replacement using regular expressions.
- str_trim(): Remove leading/trailing whitespace.
- str_pad(): Pad strings to a certain length.

3. stringi Package:

The stringi package provides highly optimized and comprehensive string manipulation functionalities, often with better performance than base R functions for complex tasks. It's a powerful alternative to stringr for advanced use cases.

Choosing the right tool:

- For simple, one-off tasks, base R functions are often sufficient.
- For consistent and readable code within the Tidyverse ecosystem, stringr is the preferred choice.
- For performance-critical applications or advanced Unicode handling, stringi offers robust solutions.

math and simulation in r

R is a versatile programming language used extensively for mathematical computations and running simulations in various fields, from statistics to engineering. You can perform basic arithmetic and use built-in mathematical functions for complex calculations, as well as simulate random processes

by generating data from different probability distributions or for specific equations. R's ability to create sophisticated graphs and handle computationally intensive tasks makes it a powerful tool for exploring "what-if" scenarios, testing hypotheses, and making data-driven decisions.

Mathematical Computations in R

- Arithmetic Operators:

R uses standard operators (+, -, *, /, etc.) for performing basic mathematical operations.

- Built-in Functions:

R provides numerous built-in functions for mathematical tasks, such as:

    - sqrt() for square roots
    - abs() for absolute values
    - ceiling() for rounding up to the nearest integer
    - floor() for rounding down to the nearest integer
    - max() and min() for finding the highest and lowest values in a set

- Integration:

R can perform numerical integration using
the integrate() function by specifying the function and the integration bounds.

- Set Operations:

R offers functions for set operations
like union(), intersect(), setdiff(), and the membership operator %in%.

Simulation in R

- Generating Random Data:

R can generate random numbers from various distributions using functions like rnorm() for normal distributions.

- Running Multiple Simulations:

You can use loops (e.g., for loops) or functions like sapply() to repeat simulations, allowing you to perform statistical analysis on the simulated results.

- Simulating Equations:

R can simulate equations by running them for numerous parameter values to understand how they behave under different conditions.

- Setting a Seed:

To ensure the reproducibility of your simulations, it is important to set the random seed outside of any loops or sapply statements.

- Packages:

Various R packages offer specialized functions for more complex simulation tasks.

Applications of Math and Simulation in R

- **Data Analysis & Statistics:**

R is widely used for statistical computing and analysis, with simulation being a powerful tool for understanding complex systems and random processes.

- **Mathematical Modeling**:

Students and researchers can use R to create mathematical models, solve real-world problems, and visualize model behavior.

- **Hypothesis Testing:**

Simulations allow for testing hypotheses by analyzing system behavior under various simulated circumstances.

- **Parameter Estimation:**

By changing variables and running simulations, you can estimate the impact of these changes on outcomes, leading to more accurate parameter estimations.

- **Forecasting:**

You can leverage simulated data to predict future trends and behaviors, supporting informed decision-making.

UNIT IV

S3 class in R Programming

S3 classes represent the most fundamental and widely used object-oriented programming system in R. They are characterized by their simplicity and lack of formal class definitions, making them highly flexible.

Key characteristics of S3 classes:

- Informal Definition:

S3 classes do not have a formal class definition like S4 or Reference Classes. An object becomes an S3 object of a particular class simply by assigning a character vector to its class attribute.

- List-based Structure:

S3 objects are typically built upon lists, where the named components of the list serve as the object's "member variables" or attributes.

- Generic Functions and Method Dispatch:

S3 relies on generic functions (e.g., print(), summary(), plot()) and a mechanism called method dispatch. When a generic function is called with an S3 object, R looks for a specific method function named generic.classname() (e.g., print.myclass()) that corresponds to the object's class.

- Polymorphism:

This system allows for polymorphism, where the same generic function can behave differently depending on the class of the object it operates on.

Creating an S3 Class and Object:

- Create a list: Define a list containing the desired components for your object.

CODE

```
my_data <- list(name = "Alice", age = 30, city = "New York")
```

- Assign a class attribute: Set the class attribute of the list to a character string representing the class name.

CODE

```
class(my_data) <- "Person"
```

Now, my_data is an S3 object of class "Person".

Creating S3 Methods:

To make generic functions work specifically with your S3 class, you define methods using the generic.classname() naming convention.

CODE

```
print.Person <- function(x, ...) {
  cat("Name:", x$name, "\n")
  cat("Age:", x$age, "\n")
  cat("City:", x$city, "\n")
}
```

Now, when you call print(my_data), it will use the custom print.Person method.

Advantages:

- Simplicity and Ease of Use:

S3 is straightforward to learn and implement, contributing to its widespread use in R's base and many packages.

- Flexibility:

The informal nature allows for easy modification and extension of object structures.

Disadvantages:

- Lack of Formal Definition:

The absence of formal class definitions can lead to less rigorous object structures and potential inconsistencies if not carefully managed.

- Single Dispatch:

Method dispatch is primarily based on the class of the first argument to the generic function, limiting multiple dispatch scenarios found in other OOP systems.

Objects have attributes and the most common attribute related to an object is class. The command class is used to define a class of an object or learn about the classes of an object. Class is a vector and this property allows two things:

- Objects are allowed to inherit from numerous classes
- Order of inheritance can be specified for complex classes

Example: Checking the class of an object

```
# Creating a vector x consisting of type of genders

x<-c("female", "male", "male", "female")

# Using the command <code>class()</code>

# to check the class of the vector
```

class(x)

Output:
[1] "character"

S4 class in r programming

S4 classes in R provide a formal and structured system for object-oriented programming, offering a more rigorous approach compared to S3 classes. They are particularly useful for developing complex data structures and R packages.

Key Features of S4 Classes:

- **Formal Definition:** S4 classes are explicitly defined using the setClass() function, specifying the class name and its slots (attributes) along with their respective data types. This formal definition ensures type safety and consistency.

CODE

```
setClass(
  "Person",
  slots = list(
    name = "character",
    age = "numeric",
    occupation = "character"
  ),
  prototype = list(
    name = "Unknown",
    age = NA_real_,
    occupation = "Unemployed"
  )
)
```

- **Object Creation:** Objects of an S4 class are created using the new() function, providing the class name and initial values for the defined slots.

CODE

```
john <- new("Person", name = "John Doe", age = 30, occupation = "Engineer")
```

- **Slot Access and Modification:** Slots within an S4 object are accessed and modified using the @ operator.

CODE

```
john@name # Access the 'name' slot
john@age <- 31 # Modify the 'age' slot
```

- **Generics and Methods:** S4 classes leverage generic functions and methods for polymorphism. Generic functions (like show(), print()) define a general operation,

while methods provide specific implementations of these generics for particular S4 classes. This allows for customized behavior based on the object's class.

CODE

```
setMethod("show", "Person", function(object) {
  cat("Name:", object@name, "\n")
  cat("Age:", object@age, "years old\n")
  cat("Occupation:", object@occupation, "\n")
})

john # Calling show(john) implicitly
```

- **Inheritance:** S4 classes support inheritance, allowing new classes to extend existing ones, inheriting their slots and methods. This promotes code reuse and hierarchical organization.

Advantages of S4 Classes:

- **Structure and Formalism:** Enforces a clear structure and data types, reducing errors.
- **Multiple Dispatch:** Can handle methods that depend on the classes of multiple arguments.
- **Package Development:** Well-suited for building robust and maintainable R packages.
- **Readability and Maintainability:** Promotes organized and understandable code.

While S4 classes offer significant advantages for complex object-oriented programming, they can be more verbose than S3 classes. The choice between S3 and S4 depends on the specific needs and complexity of the project.

S3 and S4 classes in r

R provides two primary object-oriented programming systems for defining classes and methods: S3 and S4. They differ in their level of formality and structure.

S3 Classes:

- **Informal Definition:**

S3 classes do not have a formal class definition. An object becomes an S3 object simply by assigning a class attribute to it, typically a character vector.

- **Method Dispatch:**

Method dispatch in S3 relies on generic functions and the naming convention generic.class. When a generic function (e.g., print()) is called on an S3 object, R looks for a method specifically named print.myclass() if the object's class is myclass. If no specific method is found, a default method (e.g., print.default()) is called.

- **Flexibility:**

S3 is known for its flexibility and ease of use, as it requires less boilerplate code.

- **Example:**

CODE

```
# Create an S3 object
my_s3_object <- list(name = "Alice", age = 30)
class(my_s3_object) <- "Person"

# Define an S3 method
print.Person <- function(x, ...) {
  cat("Name:", x$name, "\n")
  cat("Age:", x$age, "\n")
}

# Call the method
print(my_s3_object)
```

S4 Classes:

- **Formal Definition:**

S4 classes have a formal definition using the setClass() function, which defines the class's slots (attributes) and their types.

- **Method Dispatch:**

Method dispatch in S4 uses the setMethod() function to define methods for generic functions based on the class of the arguments. This provides stronger type checking and more robust method dispatch.

- **Structure and Validation:**

S4 offers a more structured and rigorous approach to object-oriented programming, including features like inheritance and validation of slot types.

- **Example:**

CODE

```
# Define an S4 class
setClass("Employee",
         slots = c(name = "character", employee_id = "numeric"))

# Create an S4 object
my_s4_object <- new("Employee", name = "Bob", employee_id = 12345)
```

```
# Define an S4 method for the 'show' generic
setMethod("show", "Employee",
          function(object) {
            cat("Employee Name:", object@name, "\n")
            cat("Employee ID:", object@employee_id, "\n")
          })

# Call the method (implicitly by printing the object)
my_s4_object
```

Key Differences:

- **Formal Definition:** S4 has formal class definitions with slots, while S3 relies on attributes.
- **Method Definition:** S4 uses setMethod() for defining methods, while S3 uses the generic.class naming convention.
- **Type Checking:** S4 provides stronger type checking and validation of object slots.
- **Inheritance:** S4 supports formal inheritance mechanisms.
- **Attribute Access:** S3 objects use $ to access attributes, while S4 objects use

Managing objects in R programming

1. Creating and Assigning Objects:

Objects are created by assigning values to a variable name using the assignment operator <-.

code

```
my_vector <- c(1, 2, 3, 4, 5)
my_dataframe <- data.frame(Name = c("Alice", "Bob"), Age = c(25, 30))
```

2. Listing and Inspecting Objects:

- Use ls() to list all objects in the current environment.
- Use ls(pattern = "my_") to list objects matching a specific pattern.
- Use exists("object_name") to check if an object exists.
- Use class(object) to determine the class of an object (e.g., "numeric", "data.frame").
- Use str(object) to get a compact summary of an object's structure.
- Use summary(object) to get a statistical summary of an object.

3. Removing Objects:

- Use rm(object_name) to remove a specific object.

- Use rm(list = ls()) to remove all objects from the current environment.

4. Saving and Loading Objects:

- Use save(object1, object2, file = "my_objects.RData") to save selected objects to a file.
- Use save.image(file = "my_workspace.RData") to save the entire workspace (all objects) to a file.
- Use load("my_objects.RData") to load objects from a saved file back into the environment**.**

5. Understanding Object-Oriented Programming (OOP) in R:

R supports OOP concepts, primarily through S3 and S4 class systems:

- S3 Classes:

Informal and flexible, relying on generic functions that dispatch methods based on the object's class.

- S4 Classes:

More formal and structured, offering better encapsulation and method dispatch based on argument types.

6. Memory Management:

- R uses a garbage collector to automatically release memory used by objects that are no longer referenced.
- Be mindful of creating large, unnecessary objects, as this can lead to increased memory consumption.
- Understanding how R handles object replacement and modification can help avoid memory leaks.

Input/output in r

In R, input and output operations involve reading data into the R environment and displaying or saving results.

Input:

- From the console:
    - readline(): Reads a single line of text input from the user. The input is always treated as a character string and may require conversion to other data types (e.g., as.integer(), as.numeric()).

code

```
name<-readline("Enteryourname:")
age <- as.integer(readline("Enter your age: "))
```

- scan(): Reads multiple values from the console or a file, into a vector or list. It can handle different data types and is useful for reading structured data.

code

```
numbers <- scan(what = numeric()) # Reads numeric values
```

- From files:
    - read.csv(), read.table(), read_csv() (from readr package): Functions for reading tabular data from CSV files and other delimited text files.
    - readRDS(), saveRDS(): Functions for saving and loading R objects in a binary format, preserving their structure and attributes.
    - Specialized functions: R also has packages and functions for reading various other data formats like Excel files, databases, JSON, XML, etc.

Output:

- To the console:
    - print(): Displays the value of an R object.
    - cat(): Concatenates and prints its arguments to the console, often used for displaying messages or formatted output.

code

```
cat("Hello,", name, "! You are", age, "years old.\n")
```

- sprintf(): Formats strings and variables similar to the C printf() function, returning a character vector.

code

```
message <- sprintf("Your name is %s and your age is %d.", name, age) print(message)
```

- To files:
    - write.csv(), write.table(), write_csv() (from readr package): Functions for writing tabular data to CSV files and other delimited text files.
    - save(), save.image(): Functions for saving R objects or the entire workspace to a file.
    - sink(): Redirects R output (console output, messages, warnings) to a specified file.

Accessing keyboard and monitor in r

In R, interaction with the keyboard (for input) and the monitor (for output) is handled through a set of built-in functions.

Accessing the Keyboard (Input):

- readline(): This function reads a single line of input from the console as a character string. It can optionally display a prompt message to the user.

code

```
name <- readline("Enter your name: ")
print(paste("Hello,", name))
```

- scan(): This function reads data into a vector or list from the console or a file. It can be used for reading numeric or character data. To terminate input from the console, press Enter twice.

code

```
# Read numeric input
numbers <- scan(what = numeric())
print(numbers)

# Read character input
words <- scan(what = character())
print(words)
```

Accessing the Monitor (Output):

- print(): This function displays the value of an R object to the console. It is commonly used to show the contents of variables, data structures, or the results of computations.

code

```
x <- 10
print(x)
```

- cat(): This function concatenates its arguments and displays them to the console. It is useful for printing formatted output or combining multiple pieces of information.
- code

```
cat("The value of x is:", x, "\n")
```

These functions provide the fundamental tools for creating interactive R programs that can receive input from the user and display results on the screen.

Reading and writing files in r programming

R offers various functions for reading and writing data to and from files, supporting different file formats like CSV, text files, and R's own binary formats (RDA/RDS).

1. **Reading Files:**

- **CSV Files:** Use read.csv() to read comma-separated value files.

code

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

- **Text Files:** Use read.table() for general tabular data in text files, specifying the delimiter if it's not a space. readLines() can read a file line by line.

code

```
my_data <- read.table("path/to/your/file.txt", header = TRUE, sep = "\t")
lines <- readLines("path/to/your/text_file.txt")
```

- **R Binary Files (RDA/RDS):**
    - load() is used to load objects saved in .RData or .rda files (which can contain multiple R objects).
    - readRDS() is used to read a single R object saved in an .rds file.

code

```
load("path/to/your/data.RData") # Loads objects directly into the environment
my_object <- readRDS("path/to/your/object.rds")
```

2. **Writing Files:**

- **CSV Files:** Use write.csv() to write data frames to CSV files.

code

```
write.csv(my_dataframe, "path/to/output/file.csv", row.names = FALSE)
```

(Setting row.names = FALSE prevents writing the R row names as a column in the CSV.)

- **Text Files:** Use write.table() to write data frames to text files, specifying the delimiter.c

  code

```
write.table(my_dataframe, "path/to/output/file.txt", sep = "\t", row.names = FALSE)
```

- **R Binary Files (RDA/RDS):**
    - save() is used to save one or more R objects to an .RData or .rda file.
    - saveRDS() is used to save a single R object to an .rds file.

code

```
save(object1, object2, file = "path/to/output/data.RData")
saveRDS(my_object, "path/to/output/object.rds")
```

3. **Working Directory**:

- Get Current Working Directory: getwd()
- Set Working Directory: setwd("path/to/your/directory")

Accessing the internet in r

You can access the internet in R using libraries like rvest for web scraping, httr for interacting with APIs, or through base R's socket facilities for lower-level network connections. For web scraping, install the rvest package and use functions like read_html() to fetch a webpage's content, and then use html_nodes() or html_text() to extract the data. For other tasks, you can use functions like download.file() to download files directly from a URL.

Common tasks and packages

- **Web scraping:**

Use the rvest package to parse HTML and extract data from websites.

  - Install and load the package: install.packages("rvest"), library(rvest).
  - Read the webpage: webpage <- read_html("URL").
  - Extract data: Use functions like html_nodes() and html_text() to select and pull data from the HTML elements.

- **API interaction:**

The httr package is designed to work with APIs.

  - Install and load: install.packages("httr"), library(httr).
  - Use functions like GET() to make requests to APIs.

- **Downloading files:**

Use the download.file() function from base R to download files from a URL and save them to your local machine.

- **JSON data:**

The jsonlite package is useful for working with JSON data from APIs or other sources.

- **Low-level access:**

For more advanced network programming, R's socket facilities allow for direct interaction with the Internet's TCP/IP protocol.

Troubleshooting common issues

- **Antivirus/firewall:**

Antivirus software can sometimes block R's internet access. If you are on a work computer, you may need to contact your IT department to "whitelist" R's executable or internet-related files.

- **Proxy settings**:

If you are behind a proxy, you may need to configure R's settings or use a package like RCurl or httr to handle the proxy connection.

- **Connection check:**

To check if you have an active internet connection from within R, you can use functions like curl::nslookup() which is often more reliable than curl::has_internet().

string manipulation

R provides a comprehensive set of functions for manipulating strings, which are sequences of characters used to store and represent textual data.

Common String Manipulation Functions in R:

- **Concatenation:**
    - paste(): Combines strings, optionally with a separator (sep) and collapsing elements of a vector (collapse).

code

```
str1 <- "Hello"
str2 <- "World"
combined_str <- paste(str1, str2) # "Hello World"
vector_str <- paste(c("a", "b"), c("1", "2"), sep = "-") # "a-1" "b-2"
collapsed_str <- paste(c("apple", "banana"), collapse = ", ") # "apple, banana"
```

- **Case Conversion:**
    - toupper(): Converts a string to uppercase.
    - tolower(): Converts a string to lowercase.

code

```
my_string <- "R Programming"
uppercase_str <- toupper(my_string) # "R PROGRAMMING"
lowercase_str <- tolower(my_string) # "r programming"
```

- **Substring Extraction:**
    - substr(x, start, stop): Extracts a portion of a string from a specified start to end position.

code

```
text <- "R is fun"
sub_text <- substr(text, 1, 3) # "R i"
```

- **String Length:**
    - nchar(): Counts the number of characters in a string.

code

```
word <- "programming"
len <- nchar(word) # 11
```

- **Splitting Strings:**
    - strsplit(x, split): Splits a string into a list of substrings based on a delimiter (regular expression).

code

```
data <- "apple,banana,orange"
split_data <- strsplit(data, ",") # list("apple", "banana", "orange")
```

- **Searching and Replacing:**
    - grep(pattern, x, value = FALSE): Finds matches of a regular expression pattern within a character vector x. If value = TRUE, returns the matching elements; otherwise, returns the indices of matching elements.
    - sub(pattern, replacement, x): Replaces the first occurrence of a pattern with a replacement string.
    - gsub(pattern, replacement, x): Replaces all occurrences of a pattern with a replacement string.

code

```
sentence <- "The quick brown fox jumps over the lazy dog."
found_index <- grep("fox", sentence) # 1
new_sentence_sub <- sub("fox", "cat", sentence) # "The quick brown cat jumps over the lazy dog."
new_sentence_gsub <- gsub("the", "a", sentence) # "a quick brown fox jumps over a lazy dog."
```

Overview of string manipulation function

String manipulation functions are a set of operations used in programming to process, modify, and analyze textual data. These functions are fundamental across various

programming languages and databases for tasks involving strings, which are sequences of characters.

Here is an overview of common string manipulation functions and their purposes:

1. Length and Size:

- Functions like length(), size(), or strlen() return the number of characters in a string.

2. Concatenation:

- Functions like concat(), append(), or the + operator combine two or more strings into a single string.

3. Substring Extraction:

- Functions like substr(), slice(), or substring() extract a portion of a string based on starting and ending positions or length.

4. Searching and Replacing:

- Functions like find(), indexOf(), search(), replace(), sub(), or gsub() locate specific patterns or substrings within a string and/or replace them with other strings.

5. Case Conversion:

- Functions like upper(), lower(), toUpperCase(), toLowerCase(), or initcap() convert the case of characters within a string (e.g., all uppercase, all lowercase, or title case).

6. Trimming and Padding:

- Functions like trim(), ltrim(), rtrim(), or pad() remove leading/trailing whitespace or other specified characters, or add characters to the beginning or end of a string to reach a certain length.

7. Splitting and Joining:

- Functions like split() or explode() divide a string into an array of substrings based on a delimiter.
- Functions like join() or implode() combine an array of strings into a single string using a specified delimiter.

8. Character-level Operations:

- Functions from libraries like ctype.h in C provide functionalities to check character types (e.g., islower(), isdigit()) or convert individual characters.

These functions are crucial for tasks such as data cleaning, text processing, formatting output, parsing user input, and working with databases. The specific names and syntax of these functions may vary depending on the programming language (e.g., Python, Java, C++, JavaScript, R, SQL) or environment being used.

Regular expressions

In the context of regular expressions (regex), the character \r represents a carriage return.

A carriage return is a control character that, in older systems and some modern contexts, signals the cursor to return to the beginning of the current line without advancing to the next line. Its behavior can vary slightly depending on the operating system and the environment where the regex is being applied.

Key points about \r in regex:

- Meaning: It specifically matches the carriage return character.
- Platform Differences:
    - Mac OS 9 and earlier: \r was used as the sole new line character.
    - Unix and Mac OS X/macOS: \n (line feed) is the standard new line character.
    - Windows: \r\n (carriage return followed by line feed) is the standard new line sequence.
- Usage in R: When using regular expressions in R, special characters like \r need to be escaped with an additional backslash, so you would write \\r in your R code to match a carriage return.

Example:

If you have a string containing a carriage return, you could use \\r in an R regex function like str_detect() (from the stringr package) to detect its presence:

code

```
library(stringr)
text_with_cr <- "Line 1\rLine 2"
str_detect(text_with_cr, "\\r") # Returns TRUE
```

use of string utilities in the edtdbg debugging tool

The edtdbg debugging tool in R, particularly as described in "The Art of R Programming," heavily utilizes string utilities for its core functionality, which involves sending remote commands to a text editor like Vim.

A primary example of this usage is the dbgsendeditcmd() function, which constructs and sends commands to Vim. This function leverages R's string manipulation capabilities to build the shell command that interacts with Vim.

Consider the following simplified example:

code

```
dbgsendeditcmd<-function(cmd){
  vimserver <- "R_DEBUG_SERVER" # Assuming a pre-defined Vim server name
  syscmd <- paste("vim --remote-send ", cmd, " --servername ", vimserver, sep="")
  system(syscmd)
}
```

In this function:

- String Concatenation (paste()):

The paste() function is crucial for combining various string components into a single, executable shell command. It takes individual strings (like "vim --remote-send ", the cmd argument, " --servername ", and the vimserver variable) and concatenates them to form the complete command string. The sep="" argument ensures no spaces are inserted between the concatenated parts, creating a clean command.

- Command Construction:

The resulting syscmd string represents a command that can be executed in the system's shell. For instance, if cmd is "12G", and vimserver is "168", the syscmd would become "vim --remote-send 12G --servername 168".

- System Execution (system()):

The system() function then takes this constructed string and executes it as a shell command, effectively sending the desired instruction (e.g., moving the cursor to line 12) to the specified Vim instance.

In essence, edtdbg relies on string utilities to dynamically generate and execute commands that control the external text editor, allowing for interactive debugging sessions where actions within the R debugger can be reflected in the editor