Introduction to Data Science Lab
V Semester - R22

Teacher: Dr. Srilatha P
Email: pulipatisrilatha_aids@cbit.ac.in

Week -1 & 2

Introduction to R:

R is a powerful **programming language** and software environment used extensively for **statistical computing**, **data analysis**, and **graphical representation**.

It was created by **Ross Ihaka and Robert Gentleman** at the University of **Auckland**, New Zealand, and is now developed by the R Development Core Team

Features of R:

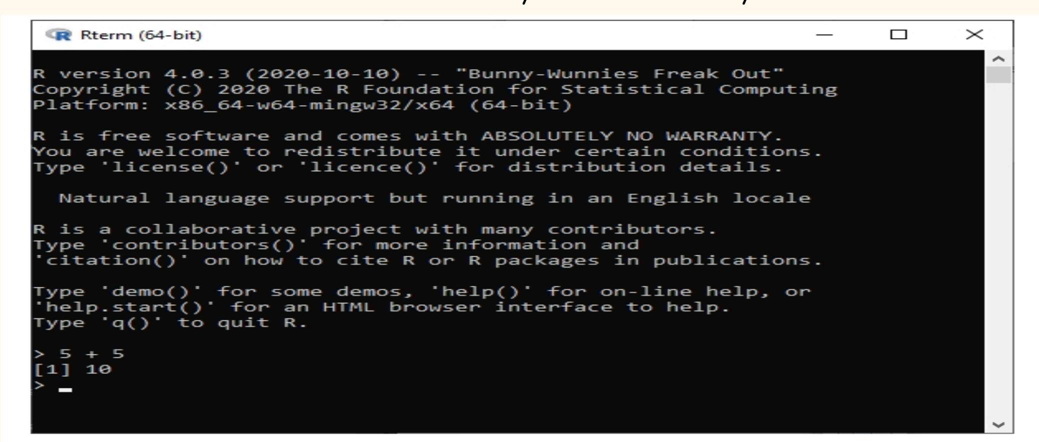
- **Statistical Analysis:** R provides a wide range of statistical techniques such as linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, and clustering.
- **Graphical Capabilities**: R is renowned for its advanced graphical capabilities, enabling users to produce high-quality plots and visualizations.
- **Open Source:** R is open-source software, meaning it is free to use, and its source code is available for anyone to inspect, modify, and enhance.
- **Extensible:** Users can extend R's capabilities through packages. The Comprehensive R Archive Network (CRAN) hosts thousands of packages developed by the R community.
- **Data Handling**: R provides robust tools for data manipulation, making it easier to clean, preprocess, and transform data.

Installation of R:

- To install R, go to **https://cloud.r-project.org/** and download the latest version of R for Windows, Mac or Linux.

- When you have downloaded and installed R, you can run R on your computer.

- The screenshot below shows how it may look like when you run R on a Windows PC:



R Comments:
Comments starts with a #.

R Variables:
Variables are used to store data values. In R, variables can hold different types of data, such as numbers, strings, or logical values.

Assignment Operator:
The assignment operator in R is <-. You can also use =, but <- is more common in practice

```
x <- 10     # Assigning the value 10 to variable x
y = 5       # Assigning the value 5 to variable y
name <- "John"  # Assigning a string value
is_active <- TRUE  # Assigning a logical value
```

Naming Conventions

- Variable names can contain letters, numbers, periods (`.`), and underscores (`_`).
- Variable names cannot start with a number.
- R is case-sensitive, so `Var` and `var` are different variables.

R Data Types
Basic data types in R can be divided into the following types:

- numeric - (10.5, 55, 787)
- integer - (1L, 55L, 100L, where the letter "L" declares this as an integer)
- complex - (9 + 3i, where "i" is the imaginary part)
- character (a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")
- logical (a.k.a. boolean) - (TRUE or FALSE)

We can use the class() function to check the data type of a variable:

Type Conversion
You can convert from one type to another with the following functions:

- as.numeric()
- as.integer()
- as.complex()

Data structures:
1. Vectors
Vectors are the most basic data structure in R. They hold elements of the same type: numeric, character, or logical.To combine the list of items to a vector, use the c() function and separate the items by a comma.

- **numeric_vector <- c(1, 2, 3, 4, 5)**
- **character_vector <- c("A", "B", "C")**
- **logical_vector <- c(TRUE, FALSE, TRUE)**

To create a vector with numerical values in a sequence, use the : operator:
# Vector with numerical values in a sequence
numbers <- 1:10

Vector Length
To find out how many items a vector has, use the length() function:

Accessing Elements:
numeric_vector[1]   # Access the first element
numeric_vector[2:4] # Access a range of elements

Operations on Vectors:
numeric_vector * 2   # Element-wise multiplication
sum(numeric_vector)   # Sum of elements
mean(numeric_vector)   # Mean of elements

2. Matrices

Matrices are two-dimensional arrays that hold elements of the same type.

A matrix is a two dimensional data set with columns and rows.

A column is a vertical representation of data, while a row is a horizontal representation of data.

A matrix can be created with the matrix() function. Specify the nrow and ncol parameters to get the amount of rows and columns:

Creating Matrices:
matrix_data <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)

Accessing Elements:

You can access the items by using [ ] brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

- **matrix_data[1, 2]   # Element at first row, second column**
- **matrix_data[1,]   # First row**
- **matrix_data[, 2]   # Second column**

Operations on Matrices:
matrix_data * 2   # Element-wise multiplication
matrix_data %*% t(matrix_data) # Matrix multiplication

3. Arrays

Arrays are multi-dimensional, generalizations of matrices.

Compared to matrices, arrays can have more than two dimensions.

We can use the array() function to create an array, and the dim parameter to specify the dimensions:

The syntax is as follow: array[row position, column position, matrix level]

Creating Arrays:

```
array_data <- array(1:12, dim = c(2, 3, 2))
```

Accessing Elements:

```
array_data[1, 2, 1]     # Element at first row, second column, first matrix
array_data[,, 1]        # First matrix
thisarray <- c(1:24)

# Access all the items from the first row from matrix one
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray[c(1),,1]

# Access all the items from the first column from matrix one
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray[,c(1),1]
```

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

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

A comma (,) before c() means that we want to access the column.

A comma (,) after c() means that we want to access the row.

4. Lists

Lists can hold elements of different types and structures, including vectors, matrices, data frames, and other lists.

Creating Lists:

```
my_list <- list(
  Name = "John",
  Age = 25,
  Scores = c(85, 90, 88)
)
```

Accessing Elements:

```
my_list$Name         # Access by name
my_list[["Name"]]     # Access by name
my_list[[1]]         # Access by index
```

Modifying Lists:

```
my_list$Name <- "Jane"   # Modify an element
my_list$Address <- "123 Street"  # Add a new element
```

5. Data Frames

Data frames are used to store tabular data. Each column in a data frame can be of a different type.

Data Frames are data displayed in a format as a table.

Data Frames can have different types of data inside it. While the first column can be character, the second and third can be numeric or logical. However, each column should have the same type of data.

Use the data.frame() function to create a data frame:

Creating Data Frames:

```
data_frame <- data.frame(
  Name = c("John", "Doe", "Jane"),
  Age = c(25, 30, 22),
  Score = c(85, 90, 88)
)
```

Accessing Data:

We can use single brackets [ ], double brackets [[ ]] or $ to access columns from a data frame:

```
data_frame$Name           # Access a column by name
data_frame[1, ]           # Access a row by index
data_frame[1:2, 1:2]      # Access a subset of the data frame
```

Adding Rows and Columns:

```
data_frame$Gender <- c("M", "M", "F")  # Adding a column
new_row <- data.frame(Name = "Alice", Age = 28, Score = 92, Gender = "F")
data_frame <- rbind(data_frame, new_row)  # Adding a row
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame[1]

Data_Frame[["Training"]]

Data_Frame$Training

Training
1 Strength
2  Stamina
3    Other
[1] Strength Stamina  Other
Levels: Other Stamina Strength
[1] Strength Stamina  Other
Levels: Other Stamina Strength
```

6. Factors

Factors are used to store categorical data and can be ordered or unordered.

Factors are used to categorize data. Examples of factors are:

Demography: Male/Female

Music: Rock, Pop, Classic, Jazz

Training: Strength, Stamina

To create a factor, use the factor() function and add a vector as argument:

 # Create a factor

music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"))

 # Print the factor

Music_genre

To only print the levels, use the levels() function:

music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"))

levels(music_genre)

```
[1] "Classic" "Jazz" "Pop" "Rock"
```

You can also set the levels, by adding the levels argument inside the factor() function:

music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"), levels = c("Classic", "Jazz", "Pop", "Rock", "Other"))

levels(music_genre)

Creating Factors:
factor_data <- factor(c("High", "Medium", "Low", "High", "Low"))

Ordered Factors:
ordered_factor <- factor(c("Low", "Medium", "High"), ordered = TRUE)

Accessing Levels:
levels(factor_data)   # Get the levels of the factor

Changing Levels:
levels(factor_data) <- c("L", "M", "H") # Renaming levels

Summary

- Vectors: Basic building blocks, holding elements of the same type.
- Matrices: Two-dimensional arrays with homogeneous elements.
- Arrays: Multi-dimensional generalizations of matrices.
- Lists: Can hold elements of different types and structures.
- Data Frames: Tabular data structures with heterogeneous columns.
- Factors: Used for categorical data, can be ordered or unordered.

Exercises:

1. **Implementation of R program to create a list containing numbers, vectors, and logical values., Strings**

R code:

```
  # Creating a list in R
my_list <- list(
  # Adding a single number
  single_number = 42,

  # Adding a numeric vector
  numeric_vector = c(1, 2, 3, 4, 5),

  # Adding a logical vector
  logical_vector = c(TRUE, FALSE, TRUE),

  # Adding a character string
  character_string = "Hello, R!"
)

# Displaying the list
print(my_list)

# Accessing elements of the list
print(my_list$single_number)       # Access by name
print(my_list[["numeric_vector"]]) # Access by name
print(my_list[[3]])                # Access by index
print(my_list$character_string)    # Access by name

# Modifying elements of the list
my_list$single_number <- 99
my_list$logical_vector <- c(FALSE, FALSE, TRUE)
my_list$character_string <- "Updated string"

# Adding new elements to the list
my_list$new_element <- "New Element"

# Displaying the updated list
print(my_list)
```

2. **Demonstrate the usage of R data structures. (List, Tuples, Sets, Dictionaries, Strings, Factors)**

R Code:

1. Lists

```
my_list <- list(  Name = "John",  Age = 25,  Scores = c(85, 90, 88),  Address = list(    Street = "123
Main St",    City = "New York",    Zip = "10001"  ))
```

Accessing List Elements:

```
my_list$Name        # Access by name
my_list[["Age"]]    # Access by name
my_list[[3]]        # Access by index
```

```
my_list$Address$City  # Access nested list element
```

Modifying List Elements:

```
my_list$Name <- "Jane"
my_list$Phone <- "555-1234"  # Add new element
```

2. Tuples (Simulated using Lists)

```
my_tuple <- list(1, "A", TRUE)
```

Accessing Tuple-like List Elements:

```
my_tuple[[1]]  # Access by index
My_tuple[[2]]
```

3. Sets (Using unique and union functions)

R does not have a built-in set data structure, but we can use vectors with functions like unique to simulate sets.

Creating a Set:

```
set_a <- unique(c(1, 2, 3, 4, 5, 5, 4))  # Removes duplicates
set_b <- unique(c(3, 4, 5, 6, 7))
```

Set Operations:

```
union_set <- union(set_a, set_b)         # Union
intersect_set <- intersect(set_a, set_b)  # Intersection
diff_set <- setdiff(set_a, set_b)        # Difference
```

4. Dictionaries (Using Named Lists)

R does not have a dictionary data structure like Python, but named lists can serve a similar purpose.

Creating a Dictionary-like List:

```
my_dict <- list(  Name = "John",  Age = 25,  Address = "123 Main St")
```

Accessing Dictionary-like List Elements:

```
my_dict$Name       # Access by name
my_dict[["Age"]]   # Access by name
```

Modifying Dictionary-like List Elements:

```
my_dict$Name <- "Jane"
my_dict$Phone <- "555-1234"  # Add new element
```

5. Strings

Strings in R are character data and can be manipulated using various functions.

Creating Strings:
my_string <- "Hello, R!"

String Operations:

```
nchar(my_string)            # Number of characters
substr(my_string, 1, 5)       # Substring
paste(my_string, "How are you?") # Concatenation
toupper(my_string)           # Convert to uppercase
```

6. Factors

Factors are used for categorical data and can be ordered or unordered.

Creating Factors:

```
factor_data <- factor(c("High", "Medium", "Low", "High", "Low"))
```

Accessing and Modifying Factors:

```
levels(factor_data)         # Get levels
table(factor_data)          # Frequency table
levels(factor_data) <- c("L", "M", "H")  # Renaming levels
```

Ordered Factors:

```
ordered_factor <- factor(c("Low", "Medium", "High"), ordered = TRUE)
```

Summary

Lists: Flexible containers for elements of different types.
Tuples: Simulated using lists.
Sets: Simulated using vectors with unique, union, intersect, and setdiff functions.
Dictionaries: Simulated using named lists.
Strings: Character data manipulated using string functions.
Factors: Categorical data structures with levels.

Create Vectors:

- **V=c(1:20,19:1)**
- **V1=c(rep(2,4),1,2,3,8,1,9)**
- **V2=rep(c(4,6,3),times=10)**
- **V3=rep(c(4,6,3),times=31) where there are 11 occurrences of 4, 10 occurrences of 6, and 10 occurrences of 3.**
- **v = paste("label", 1:30)**
- **v = paste("n", 1:30, sep=".")**

Execute the following lines which create two vectors of random integers which are chosen with replacement from the integers 0, 1, ..., 999. Both vectors have length 250

```
xVec <- sample(0:999, 250, replace=T)
yVec <- sample(0:999, 250, replace=T)
```

Suppose $x = (x_1, x_2, \ldots, x_n)$ denotes the vector xVec and
$y = (y_1, y_2, \ldots, y_n)$ denotes the vector yVec.

- Create a vector $(y_2 - x_1, y_3 - x_2, \ldots, y_n - x_{n-1})$
  > z = c(y[2:250] - x[1:249])

- Create a vector $(\sin(y_1) / \cos(x_2), \ldots, \sin(y_{n-1}) / \cos(x_n))$
  > z = c(sin(y[1:249]) / cos(x[2:250]))

- Create a vector $(x_1 + 2x_2 - x_3, x_2 + 2x_3 - x_4, \ldots, x_{n-2} + 2x_{n-1} - x_n)$
  > z = c(x[1:249] + 2*x[2:250] - x[3:251])

Use the vectors xVec and yVec created in the previous question and the functions sort, order, mean, sqrt, sum, and abs.

(a) Pick out the values in yVec which are > 600

y[y>600]

(b) What are the index positions in yVec of the values which are > 600?

order(y[y>600])

(c) How many numbers in x are divisible by 2

sum(x%2==0)

(d) Pick out the elements in yVec at index positions 1, 4, 7, 10, 13, …

y[seq(1,250,by=3)]

Creating Matrices

1. Create a matrix A with the numbers 1 to 9 arranged in 3 rows and 3 columns.
2. Create a matrix B with the numbers 1 to 12 arranged in 3 rows and 4 columns.
3. Create a matrix C with the numbers 1 to 16 arranged in 4 rows and 4 columns by filling the matrix by row.

A <- matrix(1:9, nrow = 3, ncol = 3)
B <- matrix(1:12, nrow = 3, ncol = 4)
C <- matrix(1:16, nrow = 4, ncol = 4, byrow = TRUE)

Basic Matrix Operations

1. Add two matrices A and C.
2. Multiply matrix A by a scalar value 2.
3. Perform matrix multiplication of A and the transpose of A.

A_plus_C <- A + C # This will throw an error since A and C have different dimensions
A_times_2 <- A * 2
A_mult_A_transpose <- A %*% t(A)

Accessing Elements and Slicing

1. Extract the element at the 2nd row and 3rd column of matrix A.
2. Extract the entire 2nd row of matrix A.
3. Extract the entire 3rd column of matrix B.
4. Create a submatrix of C that contains the elements from the 2nd and 3rd rows and the 2nd and 3rd columns.

element_A_2_3 <- A[2, 3]
row_A_2 <- A[2,]
col_B_3 <- B[, 3]
submatrix_C <- C[2:3, 2:3]

Matrix Functions

1. Find the row sums of matrix `A`.
2. Find the column means of matrix `B`.
3. Calculate the determinant of matrix `C`.
4. Find the inverse of matrix `C` (note: `C` must be square and non-singular).

row_sums_A <- rowSums(A)
col_means_B <- colMeans(B)
det_C <- det(C)
inv_C <- solve(C)

Adding Names to Rows and Columns

1. Create a matrix `D` with 3 rows and 3 columns and add row names as "row1", "row2", "row3" and column names as "col1", "col2", "col3".
2. Change the column names of matrix `A` to "a", "b", "c".

D <- matrix(1:9, nrow = 3, ncol = 3)
rownames(D) <- c("row1", "row2", "row3")
colnames(D) <- c("col1", "col2", "col3")
colnames(A) <- c("a", "b", "c")

Applying Functions to Matrices

1. Use the `apply` function to find the sum of each row in matrix `A`.
2. Use the `apply` function to find the product of each column in matrix `B`

row_sums_A_apply <- apply(A, 1, sum)
col_products_B_apply <- apply(B, 2, prod)

Logical Operations

1. Create a logical matrix by checking which elements in matrix `A` are greater than 5.
2. Count how many elements in matrix `B` are greater than 6.

logical_matrix_A <- A > 5
count_greater_than_6_B <- sum(B > 6)

Combining Matrices

1. Combine matrices `A` and `C` row-wise (use `rbind`).
2. Combine matrices `A` and `B` column-wise (use `cbind`).

combined_row <- rbind(A, C)
combined_col <- cbind(A, B) # This will throw an error since A and B have different row counts

Reshaping Matrices

1. Reshape matrix `C` into a 2x8 matrix.
2. Flatten matrix `B` into a single vector.

reshaped_C <- matrix(C, nrow = 2, ncol = 8)
flattened_B <- as.vector(B)

Advanced Matrix Operations

1. Perform element-wise multiplication of `A` and `C`.
2. Compute the eigenvalues and eigenvectors of matrix `C`.

element_wise_mult <- A * C # This will throw an error since A and C have different dimensions
eigen_C <- eigen(C)

Creating Lists

1. Create a list named `my_list` containing the following elements:
   a numeric vector `c(1, 2, 3)`,
   a character vector `c("a", "b", "c")`,
   and a logical vector `c(TRUE, FALSE, TRUE)`.
2. Create a nested list named `nested_list` that contains the elements of `my_list` and an additional element which is a matrix with numbers 1 to 4 arranged in 2 rows and 2 columns.

```
my_list <- list(numeric_vector = c(1, 2, 3),  char_vector = c("a", "b", "c"),   logical_vector =
    (TRUE, FALSE, TRUE))
nested_list <- list(my_list, matrix = matrix(1:4, nrow = 2, ncol = 2))
```

Accessing List Elements

1. Access the second element of `my_list`.
2. Access the character vector from `my_list` using the `$` operator.
3. Access the first element of the nested matrix in `nested_list`.

```
second_element <- my_list[[2]]
char_vector <- my_list$char_vector
first_element_matrix <- nested_list[[2]][1, 1]
```

Modifying Lists

1. Add a new element `c(4, 5, 6)` to `my_list` and name it `new_vector`.
2. Change the first element of the numeric vector in `my_list` to 10.

```
my_list$new_vector <- c(4, 5, 6)
my_list$numeric_vector[1] <- 10
```

List Operations

1. Create a list `list1` with elements `a = 1:3` and `b = 4:6`. Create another list `list2` with elements `c = 7:9` and `d = 10:12`. Combine these two lists into a single list.
2. Apply the `sum` function to each element of `list1` using `lapply`.

```
list1 <- list(a = 1:3, b = 4:6)
list2 <- list(c = 7:9, d = 10:12)
combined_list <- c(list1, list2)
summed_list <- lapply(list1, sum)
```

Naming List Elements

1. Create an unnamed list `unnamed_list` with elements `10, "R", TRUE`. Name the elements of this list as `num`, `char`, and `log`.

```
unnamed_list <- list(10, "R", TRUE)
names(unnamed_list) <- c("num", "char", "log")
```

List within List

1. Create a list `inner_list` with elements `x = 5` and `y = 6`. Create another list `outer_list` containing `inner_list` and an element `z = 7`.

```
inner_list <- list(x = 5, y = 6)
outer_list <- list(inner_list = inner_list, z = 7)
```

Extracting Subsets

1. Extract the numeric vector from `my_list` and store it in `num_vec`.
2. Extract the first two elements of the character vector from `my_list` and store them in `char_vec_subset`.

```
num_vec <- my_list$numeric_vector
char_vec_subset <- my_list$char_vector[1:2]
```

Combining Lists

1. Combine two lists `list3 = list(1, 2, 3)` and `list4 = list("a", "b", "c")` into a single list.
2. Flatten the combined list from the previous step into a vector.

```
list3 <- list(1, 2, 3)
list4 <- list("a", "b", "c")
combined_list2 <- c(list3, list4)
flattened_list <- unlist(combined_list2)
```

List Length and Structure

1. Find the length of `my_list`.
2. Print the structure of `nested_list` using the `str` function.

```
list_length <- length(my_list)
str(nested_list)
```

Converting List to Data Frame

1. Convert the following list `data_list` to a data frame:

```
data_list <- list(name = c("John", "Jane", "Doe"), age = c(30, 25, 35),
  gender = c("M", "F", "M"))
data_frame <- as.data.frame(data_list)
```

Creating Factors

1. Create a factor `gender` with levels "Male" and "Female" from the vector `c("Male", "Female", "Female", "Male", "Female")`.
2. Create an ordered factor `education` with levels "High School", "Bachelor", and "Master" from the vector `c("Master", "Bachelor", "High School", "Bachelor", "Master")`.

```
gender <- factor(c("Male", "Female", "Female", "Male", "Female"))
education <- factor(c("Master", "Bachelor", "High School", "Bachelor", "Master"),
                    levels = c("High School", "Bachelor", "Master"),
                    ordered = TRUE)
```

Factor Levels

1. Print the levels of the `gender` factor.
2. Change the levels of the `gender` factor to "M" and "F".

```
levels(gender)
levels(gender) <- c("M", "F")
```

Accessing and Modifying Factors

1. Access the second element of the `education` factor.
2. Change the third element of the `education` factor to "Master".

```
second_element_education <- education[2]
education[3] <- "Master"
```

Factor Frequency

1. Create a factor `colors` with levels "Red", "Green", and "Blue" from the vector `c("Red", "Green", "Blue", "Green", "Green", "Red")`.
2. Find the frequency of each level in the `colors` factor.

```
colors <- factor(c("Red", "Green", "Blue", "Green", "Green", "Red"))
colors_frequency <- table(colors)
```

Dropping Unused Levels

1. Create a factor `status` with levels "Single", "Married", and "Divorced" from the vector `c("Single", "Married", "Single", "Single")`.
2. Drop the unused level "Divorced" from the `status` factor.

```
status <- factor(c("Single", "Married", "Single", "Single"),
                 levels = c("Single", "Married", "Divorced"))
status <- droplevels(status)
```

Combining Factors

1. Create two factors: `factor1` with levels "A", "B", and `factor2` with levels "B", "C".
2. Combine `factor1` and `factor2` into a single factor.

```
factor1 <- factor(c("A", "B", "A", "B"))
factor2 <- factor(c("B", "C", "B", "C"))
combined_factor <- factor(c(as.character(factor1), as.character(factor2)))
```

Ordered Factors

1. Create an ordered factor `temperature` with levels "Low", "Medium", and "High" from the vector `c("High", "Low", "Medium", "High", "Low")`.
2. Check if the first element of `temperature` is greater than the second element.

```
temperature <- factor(c("High", "Low", "Medium", "High", "Low"),
                      levels = c("Low", "Medium", "High"),
                      ordered = TRUE)
comparison_result <- temperature[1] > temperature[2]
```

Renaming Factor Levels

1. Create a factor `months` with levels "Jan", "Feb", "Mar" from the vector `c("Jan", "Feb", "Mar", "Jan")`.
2. Rename the levels of `months` to "January", "February", "March".

```
months <- factor(c("Jan", "Feb", "Mar", "Jan"))
levels(months) <- c("January", "February", "March")
```

Converting Factors

1. Create a factor `grades` with levels "A", "B", "C" from the vector `c("A", "B", "A", "C")`.
2. Convert the `grades` factor to a character vector.
3. Convert the `grades` factor to a numeric vector with levels ordered as 1, 2, 3 for "A", "B", "C".

```
grades <- factor(c("A", "B", "A", "C"))
grades_char <- as.character(grades)
grades_numeric <- as.numeric(grades)
```

Applying Functions to Factors

1. Create a factor `responses` with levels "Yes", "No", "Maybe" from the vector `c("Yes", "No", "Maybe", "Yes", "No")`.
2. Use the `tapply` function to count the number of each response.

```
responses <- factor(c("Yes", "No", "Maybe", "Yes", "No"))
responses_count <- tapply(responses, responses, length)
```

Creating and Manipulating Strings

1. Create a string variable `greeting` with the value "Hello, world!".
2. Extract the substring "world" from `greeting`.
3. Convert the entire string `greeting` to uppercase.

```
greeting <- "Hello, world!"
substring_world <- substr(greeting, 8, 12)
uppercase_greeting <- toupper(greeting)
```

String Length

1. Create a string variable `my_string` with the value "OpenAI".
2. Find the length of the string `my_string`.

```
my_string <- "OpenAI"
string_length <- nchar(my_string)
```

String Concatenation

1. Create two string variables `first_name` with the value "John" and `last_name` with the value "Doe".
2. Concatenate `first_name` and `last_name` with a space in between to form the full name.

```
first_name <- "John"
last_name <- "Doe"
full_name <- paste(first_name, last_name)
```

Splitting Strings

1. Create a string variable `sentence` with the value "R is a powerful language".
2. Split the string `sentence` into individual words.

```
sentence <- "R is a powerful language"
words <- strsplit(sentence, " ")
```

Replacing Substrings

1. Create a string variable `text` with the value "I love cats".
2. Replace the word "cats" with "dogs" in the string `text`.

```
text <- "I love cats"
new_text <- sub("cats", "dogs", text)
```

Pattern Matching and Extraction

1. Create a string variable `email` with the value "contact@openai.com".
2. Check if the string `email` contains the pattern "openai".
3. Extract the domain part of the email (i.e., "openai.com").

```
email <- "contact@openai.com"
contains_pattern <- grepl("openai", email)
domain <- sub(".*@", "", email)
```

Formatting Strings

1. Create variables `age` with the value 30 and `name` with the value "Alice".
2. Create a formatted string "Alice is 30 years old." using the `sprintf` function.

```
age <- 30
name <- "Alice"
formatted_string <- sprintf("%s is %d years old.", name, age)
```

Collapsing Vectors into Strings

1. Create a character vector `colors` with the values "red", "green", and "blue".
2. Collapse the `colors` vector into a single string with comma separation.

```
colors <- c("red", "green", "blue")
collapsed_colors <- paste(colors, collapse = ", ")
```

Repeating Strings

1. Create a string variable `word` with the value "R".
2. Repeat the string `word` 5 times, separated by a hyphen.

```
word <- "R"
repeated_word <- paste(rep(word, 5), collapse = "-")
```

String Trimming

1. Create a string variable `padded_string` with the value " Hello, world! ".
2. Trim the leading and trailing white spaces from `padded_string`.

```
padded_string <- "  Hello, world!  "
trimmed_string <- trimws(padded_string)
```

Extracting Specific Characters

1. Create a string variable `code` with the value "R2024".
2. Extract the numeric part from the string `code`.

```
code <- "R2024"
numeric_part <- gsub("[^0-9]", "", code)
```

Checking for Digits

1. Create a string variable `mixed` with the value "R4Data".
2. Check if the string `mixed` contains any digits.

```
mixed <- "R4Data"
contains_digits <- grepl("[0-9]", mixed)
```

Substring Matching and Counting

1. Create a string variable `sentence2` with the value "R is an amazing programming language. R is widely used.".
2. Count how many times the letter "R" appears in `sentence2`.

```
sentence2 <- "R is an amazing programming language. R is widely used."
count_R <- gregexpr("R", sentence2)[[1]]
num_R <- length(count_R)
```

Case Conversion

1. Create a string variable `phrase` with the value "Learning R is FUN".
2. Convert the entire string `phrase` to lowercase.

```
phrase <- "Learning R is FUN"
lowercase_phrase <- tolower(phrase)
```

Extracting Email Usernames

1. Create a vector `emails` with the values "alice@example.com", "bob@example.com", and "carol@example.com".
2. Extract the usernames (i.e., the part before the "@") from each email.

```
emails <- c("alice@example.com", "bob@example.com", "carol@example.com")
usernames <- sub("@.*", "", emails)
```

Creating Data Frames

1. Create a data frame `df` with the following columns:
- `ID` with values 1, 2, 3, 4
- `Name` with values "Alice", "Bob", "Charlie", "David"
- `Age` with values 25, 30, 35, 40
- `Score` with values 85, 90, 95, 80

```
df <- data.frame(ID = 1:4,  Name = c("Alice", "Bob", "Charlie", "David"),
  Age = c(25, 30, 35, 40),  Score = c(85, 90, 95, 80))
```

Accessing Data Frame Elements

1. Access the `Age` column of the data frame `df`.
2. Access the second row of the data frame `df`.
3. Access the element in the third row and fourth column of the data frame `df`.

```
age_column <- df$Age
second_row <- df[2, ]
element_3_4 <- df[3, 4]
```

Modifying Data Frames

1. Add a new column `Gender` to `df` with values "F", "M", "M", "M".
2. Change the `Score` of the first row to 88.

```
df$Gender <- c("F", "M", "M", "M")
df$Score[1] <- 88
```

Filtering Data Frames

1. Filter the rows of `df` where `Age` is greater than 30.
2. Filter the rows of `df` where `Score` is greater than or equal to 90.

```
age_above_30 <- df[df$Age > 30, ]
score_above_90 <- df[df$Score >= 90, ]
```

Sorting Data Frames

1. Sort the data frame `df` by `Age` in ascending order.
2. Sort the data frame `df` by `Score` in descending order.

```
df_sorted_by_age <- df[order(df$Age), ]
df_sorted_by_score_desc <- df[order(-df$Score), ]
```

Summarizing Data Frames

1. Get the summary statistics of the `Score` column.
2. Calculate the mean and median of the `Age` column.

```
score_summary <- summary(df$Score)
mean_age <- mean(df$Age)
median_age <- median(df$Age)
```

Merging Data Frames

1. Create another data frame `df2` with the following columns:
   - `ID` with values 1, 2, 5
   - `Height` with values 160, 170, 180
2. Merge `df` and `df2` by the `ID` column.

```
df2 <- data.frame(  ID = c(1, 2, 5),  Height = c(160, 170, 180))
merged_df <- merge(df, df2, by = "ID", all = TRUE)
```

Applying Functions to Data Frames

1. Calculate the mean `Score` for each `Gender` using the `tapply` function.
2. Calculate the maximum `Age` for each `Gender` using the `aggregate` function.

```
mean_score_by_gender <- tapply(df$Score, df$Gender, mean)
max_age_by_gender <- aggregate(Age ~ Gender, data = df, max)
```

Subsetting Data Frames

1. Select only the `Name` and `Score` columns from `df`.
2. Select the rows where `Name` is "Alice" or "Bob".

```
name_score_df <- df[, c("Name", "Score")]
alice_bob_df <- df[df$Name %in% c("Alice", "Bob"), ]
```

Renaming Columns

1. Rename the `Score` column to `Exam_Score`.

```
names(df)[names(df) == "Score"] <- "Exam_Score"
```

Handling Missing Data

1. Add a new column `Attendance` with values 90, NA, 85, 95 to `df`.
2. Remove rows with missing values in the `Attendance` column.
3. Replace the missing value in the `Attendance` column with the mean of the `Attendance` column.

```
df$Attendance <- c(90, NA, 85, 95)
df_no_na <- df[!is.na(df$Attendance), ]
df$Attendance[is.na(df$Attendance)] <- mean(df$Attendance, na.rm = TRUE)
```

Reshaping Data Frames

1. Create a data frame `df3` with the following structure:
   - `ID` with values 1, 1, 2, 2
   - `Test` with values "Midterm", "Final", "Midterm", "Final"
   - `Score` with values 85, 88, 90, 92
2. Reshape `df3` to a wide format with `ID` as rows and `Test` as columns using the `spread` function from the `tidyr` package.

```
library(tidyr)

df3 <- data.frame(  ID = c(1, 1, 2, 2),  Test = c("Midterm", "Final",
   "Midterm", "Final"),  Score = c(85, 88, 90, 92))
df3_wide <- spread(df3, Test, Score)
```

Aggregating Data Frames

1. Aggregate the `Score` by `ID` and calculate the mean score for each `ID`.

```
mean_score_by_id <- aggregate(Score ~ ID, data = df3, mean)
```

Combining Data Frames by Rows

1. Create a new data frame `df4` with the same structure as `df`.
2. Combine `df` and `df4` by rows using the `rbind` function.

```
df4 <- data.frame(  ID = 5:8,  Name = c("Eve", "Frank", "Grace", "Hank"),
  Age = c(28, 33, 38, 43),  Exam_Score = c(88, 92, 84, 79),  Gender =
   ("F", "M", "F", "M"))
combined_df <- rbind(df, df4)
```