**SEMESTER-II**

**INTRODUCTION TO DATA SCIENCE AND R PROGRAMMING**

**LAB RECORD**

**Experiment 1. Installing R and R studio, with proper notes on version management, cosmetic settings and different libraries.**

**Procedure:**

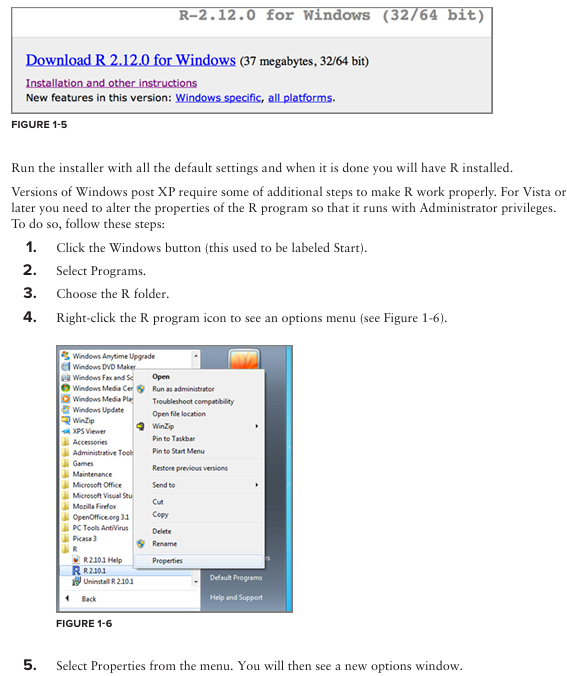
R can deal with a huge variety of mathematical and statistical tasks.

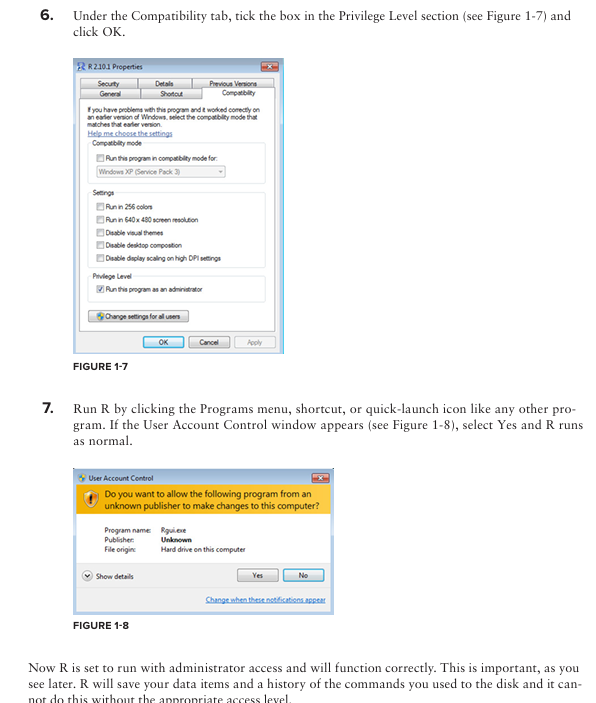
R is open source, which means that it is continually being reviewed and improved.

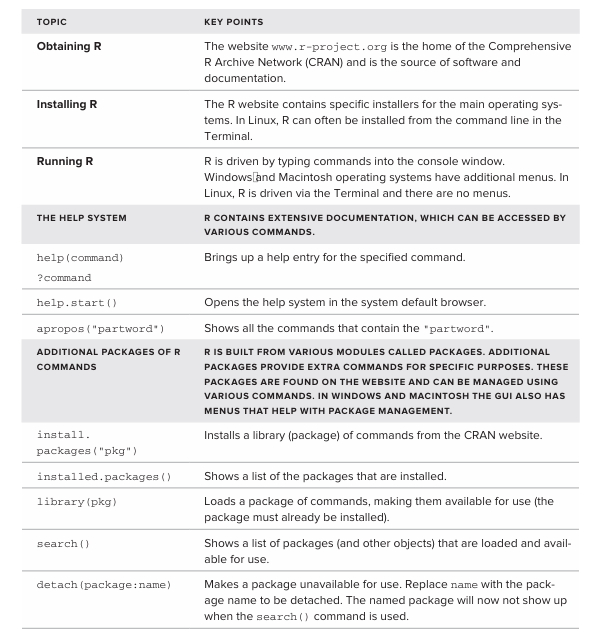
R runs on most computers—installations are available for Windows, Macintosh, and Linux. It also has good interoperability, so if you work on one computer and switch to another you can take your work with you.

The Comprehensive R Archive Network (CRAN) is a network of websites that host the R program

CRAN is the R website and holds downloads (including old versions of software) and documentation (e.g. manuals, FAQs).





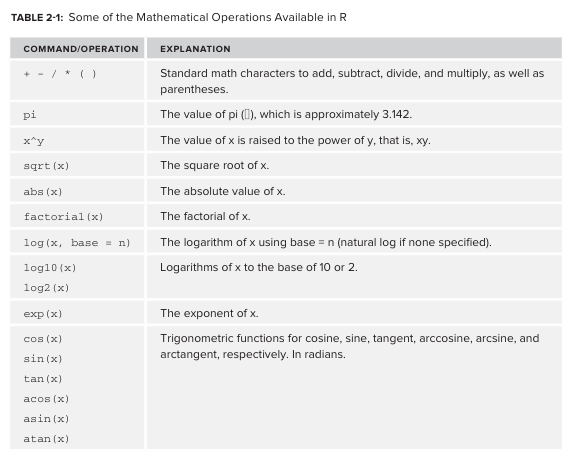


**Experiment 2: Basic operations in r with arithmetic and statistics.**

Use R like a calculator. You can think of R as a big calculator; it will perform many complicated calculations but generally, these are made up of smaller elements. To see how you can use R in this way, start by typing in some simple math:

> 3 + 9 + 12 -7

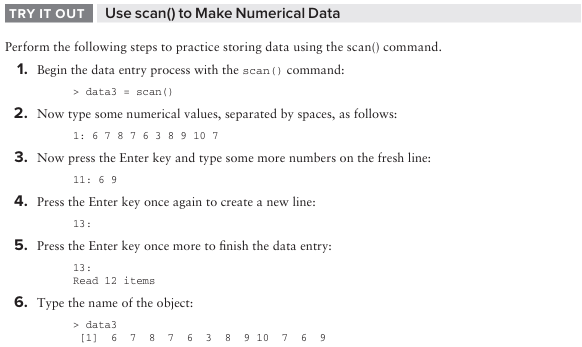
[1] 17



**Experiment 3: Getting data into R, Basic data manipulation, Loading Data into R**

**Procedure:**

**Entering Numbers** as Data: You can enter text using the scan() command as follows.



**Entering Text** as Data You can enter text using the scan() command as follows:

day2 = scan(what = 'character')

Once you are finished, enter a blank line to complete data entry

**Using the Clipboard to Make Data**

1. copy the data to the clipboard.
2. Return to R and paste the data from the clipboard into R.
3. As usual, R waits until a blank line is entered before ending the data entry so you can continue to copy and paste more data as required.
4. Once you are finished, enter a blank line to complete data entry

### **Reading a File of Data from a Disk into R**

### **Using the scan() Command:**

### **To read a file, use the command: scan(file = 'filename')**

### **Example: data6 = scan(file = 'test data.txt') print(data6)**

### **This reads data from the file named 'test data.txt'. The data should be in plain text and separated by spaces.**

### **2. Getting the Working Directory:**

### **To find out where R is looking for files by default, use the getwd() command: getwd()**

### **This will show the current working directory where R looks for files.**

### **3. Reading Files from a Different Location:**

### **If your file is in a different directory, type the full path of the file. Example: data6 = scan(file = 'Desktop/test data.txt')**

**4. Setting the Working Directory:**

* You can change the working directory using the setwd() command: Example setwd('Desktop')  
  getwd()

**5. Listing Files in a Directory:**

* To see files in the current directory, use:  
  dir()  
  list.files()
* To see files in a specific directory, type the path in quotes. Example: dir('Desktop')

**6. Using** file.choose()**:**

* In Windows and Macintosh, you can use the file.choose() command to select a file via a browser window:

data7 = scan(file.choose())

data7

**7. Reading Different Data Types:**

* If the data is separated by different characters, use the what = and sep = instructions. Example for comma-separated text:

data8 = scan(file.choose(), what = 'char', sep = ',')

print(data8)

This reads a CSV file where data items are separated by commas

### **Reading Bigger Data Files into R**.

1. **The** read.csv() **Command:**
   * Reads data from a CSV file into R.
   * Basic usage: read.csv(file.choose(), sep = ',', header = TRUE, row.names)
   * file.choose(): Opens a window to select the file.
   * sep = ',': Sets the separator to a comma.
   * header = TRUE: Indicates the first row contains column names.
   * row.names: Optionally set the row names from a specific column.

### **Alternative Commands for Reading Data in R**

1. read.table()**:**
   * A more generalized command for reading various text formats.
   * Example for space-separated data: read.table(file.choose(), header = TRUE, sep = ' ')
2. read.delim()**:**
   * Reads tab-separated values.
   * Example: read.delim(file.choose(), header = TRUE)
3. read.csv2()**:**
   * For data separated by semicolons and using commas as decimal points.
   * Example: read.csv2(file.choose())

### **Summary of Commands:**

* **CSV Files:**

read.csv(file.choose())

or

read.csv(file.choose(), sep = ',', header = TRUE, row.names)

* Space-Separated Files:

read.table(file.choose(), header = TRUE, sep = ' ')

* Tab-Separated Files:

read.delim(file.choose())

or

read.csv(file.choose(), sep = '\t')

or

read.table(file.choose(), header = TRUE, sep = '\t')

* Semicolon-Separated Files:

read.csv2(file.choose())

**Experiment No: 4: Basic Plotting**

**Line Graph**

A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. Line charts are usually used in identifying the trends in data.

The plot() function in R is used to create the line graph.

Syntax

The basic syntax to create a line chart in R is –

plot(v,type,col,xlab,ylab)

Following is the description of the parameters used −

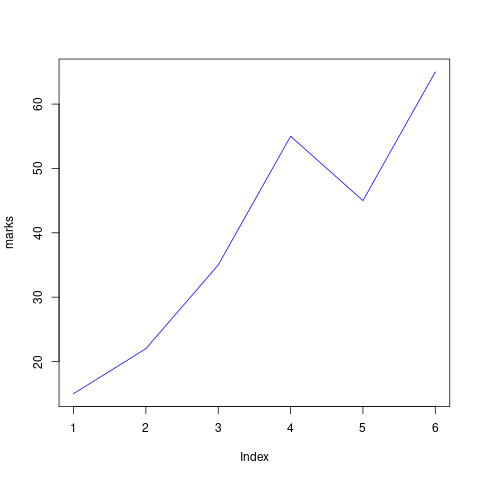
* v is a vector containing the numeric values.
* type takes the value "p" to draw only the points, "l" to draw only the lines and "o" to draw both points and lines.
* xlab is the label for x axis.
* ylab is the label for y axis.
* main is the Title of the chart.
* col is used to give colors to both the points and lines.

**Program:**

marks=c(15,22,35,55,45,65)

plot(marks, type="l", col="Blue")

**Output:**

****

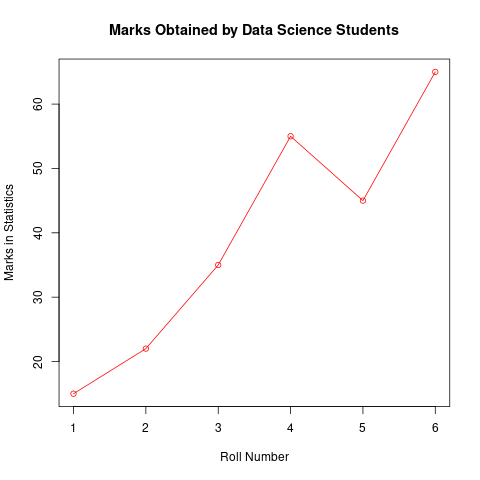
**Program2:**

marks=c(15,22,35,55,45,65)

plot(marks,type = "o", col = "red", xlab = "Roll Number", ylab = "Marks in Statistics",

main = "Marks Obtained by Data Science Students")

**Output**



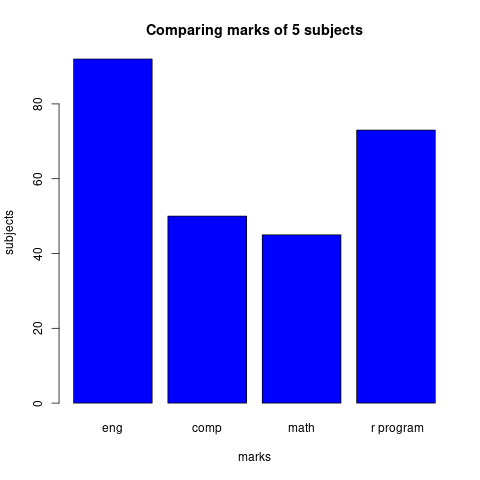
**Bar Chart**

**Program:**

marks=c(92,50,45,73)

barplot(marks, main="Comparing marks of 5 subjects", xlab="marks", ylab="subjects", names.arg = c("eng","comp","math"," r program"), col="blue",horiz=FALSE)

**Output:**



**Pie Chart**

Syntax:

The basic syntax for creating a pie-chart using the R is −

pie(x, labels, radius, main, col, clockwise)

Following is the description of the parameters used −

x is a vector containing the numeric values used in the pie chart.

labels is used to give description to the slices.

radius indicates the radius of the circle of the pie chart.(value between −1 and +1).

main indicates the title of the chart.

col indicates the color palette.

clockwise is a logical value indicating if the slices are drawn clockwise or anti clockwise.

**Program**

vtr=c(43,31,64,40,29)

names=c("london","paris","los angeles", "mexico","new york")

pie(vtr,labels=names,main="Favourable Places", col= rainbow(length(vtr)))

**Output:**



**Experiment No: 5: Loops and Functions**

There are three types of loop in R programming:

1. for
2. while
3. repeat

**1. for loop**

Syntax:

for (value in sequence)

{

statement

}

Flow Chart:

Diagram

Description automatically generated

Program to print first five natural numbers:

for (i in 1:5)

{

    # statement

    print(i)

}

Output:

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

Program to display days of week using for loop

week = c('Sunday', 'Monday','Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday')

for (day in week)

{

print(day)

}

Output:

[1] "Sunday"

[1] "Monday"

[1] "Tuesday"

[1] "Wednesday"

[1] "Thursday"

[1] "Friday"

[1] "Saturday"

**While loop**

Syntax:

while( condition)

{

statement

}

Flow chart:

Diagram

Description automatically generated

Program to calculate factorial of 5.

n=5

factorial = 1

i = 1

while (i <= n)

{

factorial = factorial \* i

i = i + 1

}

print(factorial)

Output:

[1] 120

**Repeat Loop**

Repeat loop does not have any condition to terminate the loop, a programmer must specifically place a condition within the loop’s body and use the declaration of a break statement to terminate this loop. If no condition is present in the body of the repeat loop then it will iterate infinitely.

Syntax:

repeat

{

statement

if( condition )

{

break

}

}

Flow chart:

Diagram

Description automatically generated

Program:

Program to display numbers from 1 to 5 using repeat loop in R.

val = 1

repeat

{

print(val)

val = val + 1

if(val > 5)

{

break

}

}

Output:

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

**Functions in R**

Function Components

The different parts of a function are −

1. Function Name − This is the actual name of the function. It is stored in R environment as an object with this name.
2. Arguments − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
3. Function Body − The function body contains a collection of statements that defines what the function does.
4. Return Value − The return value of a function is the last expression in the function body to be evaluated.

R has many in-built functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as user defined functions.

Built-in Function

Simple examples of in-built functions are seq(), mean(), max(), sum(x) and paste(...) etc. They are directly called by user written programs. You can refer most widely used R functions.

# Create a sequence of numbers from 32 to 44.

print(seq(32,44))

# Find mean of numbers from 25 to 82.

print(mean(25:82))

# Find sum of numbers from 41 to 68.

print(sum(41:68))

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

Program to create a function to print squares of first n natural numbers

| new = function(n)  {  for(i in 1:n)  {  b =i^2  print(b)  }  }  new(5) | Output:  [1] 1  [1] 4  [1] 9  [1] 16  [1] 25 |
| --- | --- |

**Experiment 6:** Create Vectors, Lists, Arrays, Matrices, Data frames and operations on them.

### **Vectors**

**Creating Vectors** You can create a vector using the c() function to combine elements:

a <- c(1, 2)

print(a)

Output:

1 2

**Adding Elements** Use the c() function to add new elements to an existing vector:

b <- c(a, 3)

print(b)

Output:

1 2 3

**Deleting Elements** Use negative indexing within square brackets [] to exclude elements:

c <- b[-2] # Delete the second element

print(c)

Output:

1 3

**Vector Slicing** Extract a subset of elements using a range of indices:

nums <- c(5, 10, 15)

slice <- nums[2:3] # Select elements from the second to the third

print(slice)

Output:

10 15

**Vectorised Operations** Arithmetic operations on vectors are performed element-wise:

vec1 <- c(2, 3)

vec2 <- c(4, 5)

addition <- vec1 + vec2

print(paste("Addition:", addition))

multiplication <- vec1 \* vec2

print(paste("Multiplication:", multiplication))

Output:

"Addition: 6 8"

"Multiplication: 8 15"

### **Lists**

**Creating Lists** Lists can contain elements of different classes:

my\_list <- list(val1 = 1, val2 = "a")

print(my\_list)

Output:

$val1

1

$val2

"a"

**Adding Elements** You can add a new element to a list by assigning a value to a new named index or an unnamed index:

my\_list$val3 <- TRUE

print(my\_list)

my\_list[] <- c(1, 2) # Adding without a name

print(my\_list)

Output:

$val1

1

$val2

"a"

$val3

TRUE

$val1

1

$val2

"a"

$val3

TRUE

[]

1 2

**Deleting Elements** You can remove elements by setting them to NULL:

my\_list$val3 <- NULL

print(my\_list)

Output:

$val1

1

$val2

"a"

[]

1 2

**Subsetting Lists** Use single square brackets [] to return a sublist, and double square brackets [[]] or the $ operator to access individual elements by index or name:

sub\_list <- my\_list[1:2]

print(sub\_list)

element1 <- my\_list[]

print(element1)

element2 <- my\_list$val2

print(element2)

Output:

$val1

1

$val2

"a"

1

"a"

### **Matrices**

**Creating Matrices** Matrices are vectors with a dimension attribute (nrow and ncol):

mat1 <- matrix(1:4, nrow = 2, ncol = 2)

print(mat1)

Output:

[,1] [,2]

1 3

2 4

**Matrix Operations** Basic arithmetic operations on matrices are also vectorized but operate element-wise:

mat\_a <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)

mat\_b <- matrix(c(5, 6, 7, 8), nrow = 2, ncol = 2)

mat\_add <- mat\_a + mat\_b

print(paste("Matrix Addition:"))

print(mat\_add)

mat\_mult <- mat\_a \* mat\_b # Element-wise multiplication

print(paste("Element-wise Matrix Multiplication:"))

print(mat\_mult)

Output:

"Matrix Addition:"

[,1] [,2]

6 9

8 12

"Element-wise Matrix Multiplication:"

[,1] [,2]

5 21

12 32

### **Data Frames**

**Creating Data Frames** Data frames are used to store tabular data:

df <- data.frame(col1 = c(1, 2), col2 = c("a", "b"))

print(df)

Output:

col1 col2

1 1 a

2 2 b

**Adding Columns** Assign a new vector to a new column name using the $ operator:

df$col3 <- c(TRUE, FALSE)

print(df)

Output:

col1 col2 col3

1 1 a TRUE

2 2 b FALSE

**Deleting Columns** Set a column to NULL to remove it, or use negative indexing with column indices:

df$col3 <- NULL

print(df)

df\_no\_col2 <- df[,-2] # Remove the second column

print(df\_no\_col2)

Output:

col1 col2

1 1 a

2 2 b

col1

1 1

2 2

**Adding Rows** Use the rbind() function to combine data frames with the same structure:

df1 <- data.frame(col1 = 3, col2 = "c")

combined\_df <- rbind(df, df1)

print(combined\_df)

Output:

col1 col2

1 1 a

2 2 b

3 3 c

**Deleting Rows** Use negative indexing with row numbers:

df\_no\_row2 <- combined\_df[-2,] # Remove the second row

print(df\_no\_row2)

Output:

col1 col2

1 1 a

3 3 c

**Subsetting Data Frames** Use square brackets with row and column indices or logical conditions:

subset\_cols <- df[, c("col1")]

print(subset\_cols)

subset\_rows <- df[df$col1 > 1, ]

print(subset\_rows)

Output:

col1

1 1

2 2

col1 col2

2 2 b

**Experiment 7:** Demonstrate the visualization and graphics using visualization packages like ggplot2.

ggplot2 is a powerful R package for creating stunning and informative visualizations.

**1. Install and Load ggplot2**

If you haven’t already installed ggplot2, do so using:

install.packages("ggplot2")

Then, load the package:

library(ggplot2)

### **2. Load Sample Data**

ggplot2 works well with data frames. You can use built-in datasets like mtcars or load your own data:

data(mtcars)

### **3. Create a Basic Plot**

Let’s plot a scatter plot of mpg (miles per gallon) vs hp (horsepower):

ggplot(mtcars, aes(x = hp, y = mpg)) +

geom\_point()

### **4. Customize the Plot**

* **Add Titles and Labels**

ggplot(mtcars, aes(x = hp, y = mpg)) +

geom\_point() +

labs(title = "MPG vs Horsepower",

x = "Horsepower",

y = "Miles per Gallon")

* **Change Colors**

ggplot(mtcars, aes(x = hp, y = mpg)) +

geom\_point(color = "blue")

### **5. Use Other Visualizations**

* **Histogram**

ggplot(mtcars, aes(x = mpg)) +

geom\_histogram(binwidth = 2, fill = "skyblue", color = "black")

* **Box Plot**

ggplot(mtcars, aes(x = as.factor(cyl), y = mpg)) +

geom\_boxplot(fill = "lightgreen")

* **Line Graph**

ggplot(mtcars, aes(x = hp, y = mpg)) +

geom\_line(color = "red")

ggplot2 allows endless customizations to refine your graphics.

**Experiment 8:** Implement Loop functions with lapply(), sapply(), tapply(), apply(), mapply().

### **1. lapply() (List Apply)**

lapply() **applies a function to each element of a list and returns a list containing the results**. It always returns a list, regardless of the input's class.

**Example:**

# Using a list of vectors

my\_list <- list(a = 1:3, b = 4:6, c = 7:9)

print(my\_list)

# Calculate the sum of each vector in the list

sums <- lapply(my\_list, sum)

print(sums)

# Calculate the mean of each vector in the list

means <- lapply(my\_list, mean)

print(means)

# Using an anonymous function to square each element

squared\_list <- lapply(my\_list, function(x) x^2)

print(squared\_list)

Output (based on the logic in the sources):

$a

1 2 3

$b

4 5 6

$c

7 8 9

$a

6

$b

15

$c

24

$a

2

$b

5

$c

8

$a

1 4 9

$b

16 25 36

$c

49 64 81

As shown, lapply() iterates through the list my\_list and applies the specified function (sum, mean, or an anonymous function for squaring) to each element, returning a list where each element is the result of the function call.

### **2. sapply() (Simplified Apply)**

sapply() is similar to lapply(), but it **tries to simplify the output to a vector or matrix if possible**, instead of always returning a list.

**Example:**

# Using the same list

my\_list <- list(a = 1:3, b = 4:6, c = 7:9)

print(my\_list)

# Calculate the sum of each vector (sapply simplifies to a named vector)

sums\_simple <- sapply(my\_list, sum)

print(sums\_simple)

# Calculate the mean of each vector

means\_simple <- sapply(my\_list, mean)

print(means\_simple)

Output (based on the logic in the sources):

$a

1 2 3

$b

4 5 6

$c

7 8 9

a b c

6 15 24

a b c

2 5 8

Here, sapply() returns a named vector for both sum and mean because each function applied to the list elements returned a single value.

### **3. apply() (Apply Over Array Margins)**

apply() **applies a function to the rows or columns (or other margins) of a matrix or array**.

**Example:**

# Creating a matrix

my\_matrix <- matrix(1:9, nrow = 3, byrow = TRUE)

print(my\_matrix)

# Calculate the sum of each row (MARGIN = 1)

row\_sums <- apply(my\_matrix, 1, sum)

print(row\_sums)

# Calculate the mean of each column (MARGIN = 2)

col\_means <- apply(my\_matrix, 2, mean)

print(col\_means)

Output (based on the logic in the sources):

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

1 2 3

4 5 6

7 8 9

6 15 24

4 5 6

apply() takes the matrix my\_matrix, applies the sum function across rows (MARGIN = 1), and the mean function across columns (MARGIN = 2), returning the respective results as vectors.

### **4. tapply() (Table Apply)**

tapply() **applies a function to subsets of a vector defined by one or more factor variables**. It can be thought of as a combination of split() and sapply() for vectors only.

**Example:**

# Sample data

ages <- c(25, 30, 22, 35, 28, 32)

groups <- factor(c("A", "B", "A", "B", "A", "B"))

print(ages)

print(groups)

# Calculate the mean age for each group

mean\_ages <- tapply(ages, groups, mean)

print(mean\_ages)

# Apply a function that returns more than one value (range)

range\_ages <- tapply(ages, groups, range)

print(range\_ages)

Output (based on the logic in the sources):

25 30 22 35 28 32

A B A B A B

Levels: A B

A B

25.0 32.33333

$A

22 28

$B

30 35

tapply() uses the groups factor to divide the ages vector into subsets and then applies the mean and range functions to each subset.

### **5. mapply() (Multivariate Apply)**

mapply() is a **multivariate version of lapply() which applies a function in parallel over a set of arguments**. It iterates over multiple R objects in parallel.

**Example:**

# Calculate base raised to the power of exponent for each pair

bases <- c(2, 3, 4)

exponents <- c(3, 2, 1)

print(bases)

print(exponents)

results <- mapply(function(b, e) b ^ e, bases, exponents)

print(results)

# Simulate different numbers of random normals with different means

random\_data <- mapply(rnorm, n = 1:3, mean = 5:7)

print(random\_data)

Output (based on the logic in the sources):

2 3 4

3 2 1

8 9 4

[]

5.087897

[]

6.487273 6.815835

[]

6.178099 6.784239 6.849855

mapply() applies the power function to corresponding elements of the bases and exponents vectors. It also demonstrates generating different numbers of random normal values with different means simultaneously.

These examples illustrate the basic usage of the five key loop functions in R. They offer efficient ways to perform operations on different data structures without writing explicit for or while loops in many cases.

9. Explore data using Single Variables: Unimodal, Bimodal, Histograms, Density Plots,Barcharts

You're asking to explore data using single variables, focusing on unimodal and bimodal distributions, and the visualisation techniques of histograms, density plots, and barcharts. Based on the information in the sources, here's how we can approach this in R:

When exploring a single variable, we aim to understand its distribution, identify central tendencies, and observe its spread. The shape of the distribution can tell us a lot about the underlying data.

### **Unimodal and Bimodal Distributions**

* **Unimodal distributions** are distributions with a single peak, indicating one value or a narrow range of values that occur most frequently [thought: none of the sources explicitly define unimodal or bimodal].
* **Bimodal distributions** have two distinct peaks, suggesting the presence of two separate groups or modes within the data [thought: none of the sources explicitly define unimodal or bimodal].

Histograms and density plots are excellent tools for visually identifying whether a distribution is unimodal, bimodal, or has other characteristics.

### **Histograms**

A **histogram** visualises the distribution of a single numerical variable by dividing the data into bins and showing the frequency (or relative frequency) of values falling into each bin. The shape of the histogram can help us understand the modality of the distribution.

**Creating Histograms in R:**

The hist() function in R is used to create histograms.

# Example using data from source

x <- c(1, 2, 5)

hist(x, main = "Histogram of x", xlab = "Values of x")

# Example using data from source (simulated based on description)

word\_frequencies <- c(1, 1, 1, 1, 2, 2, 3, 4, 9)

hist(word\_frequencies, breaks = 0:max(word\_frequencies), main = "Histogram of Word Frequencies", xlab = "Frequency Count")

# Example using the CO2 dataset available in R (from source)

data("CO2")

hist(CO2$uptake, main = "Histogram of CO2 Uptake", xlab = "CO2 Uptake")

The hist() function takes a vector of numeric values as input. Key arguments include:

* v: A vector containing the data.
* main: The title of the histogram.
* xlab: Label for the x-axis.
* breaks: Specifies how the bins are divided (though often R chooses sensible defaults).
* col: Colour of the bars.
* border: Colour of the bar borders.

By examining the resulting histogram, you can visually assess if there is one peak (unimodal), two peaks (bimodal), or multiple peaks (multimodal), and also get an idea of the spread and skewness of the data.

### **Density Plots**

A **density plot** provides a smoothed representation of the distribution of a numerical variable. It estimates the probability density function of the variable. Density plots can also help in identifying the number of modes in a distribution.

**Creating Density Plots in R:**

The density() function is used to estimate the kernel density, and the plot() and lines() functions can be used to visualise it.

# Example using data from source

x <- c(1, 2, 5)

plot(density(x), main = "Density Plot of x", xlab = "Values of x")

y <- c(1, 3, 8)

lines(density(y), col = "red") # Adding another density plot for comparison

# Example using the CO2 dataset

plot(density(CO2$uptake), main = "Density Plot of CO2 Uptake", xlab = "CO2 Uptake")

The density() function computes the density estimate. The plot() function can directly plot the output of density(). You can overlay multiple density plots on the same graph using lines() to compare distributions.

### **Barcharts**

A **barchart** is used to visualise the distribution of a categorical variable. It displays rectangular bars with lengths proportional to the frequencies or relative frequencies of each category. While barcharts don't directly show unimodal or bimodal distributions in the same way as histograms and density plots for numerical data, they illustrate the mode(s) as the category(ies) with the tallest bars (highest frequency) [thought: bimodal usually applies to continuous distributions, but one could argue for categorical data having two dominant categories].

**Creating Barcharts in R:**

The barplot() function in R is used to create barcharts.

# Example using data frame from source

df <- data.frame(

"Name" = c("abc", "cde", "def"),

"Gender" = c("Male", "Female", "Male")

)

# Create a frequency table of the Gender variable

gender\_table <- table(df$Gender)

print(gender\_table)

# Create a barchart of the Gender distribution

barplot(gender\_table, main = "Distribution of Gender", xlab = "Gender", ylab = "Frequency")

# Example using H vector from source

H <- c(50, 40, 10)

barplot(H, names.arg = c("mpcs", "mscs", "dscs"), main = "Bar Chart of Categories", ylab = "Values")

The barplot() function takes a vector or a matrix of values as input. Key arguments include:

* H: A vector or matrix of values representing the heights of the bars.
* names.arg: A vector of names to be displayed below each bar.
* main: The title of the barchart.
* xlab: Label for the x-axis.
* ylab: Label for the y-axis.
* col: Colour(s) for the bars.

In the context of a single categorical variable, the "mode" would be the category with the highest bar. If two categories have similarly high and distinct frequencies compared to others, one might loosely describe it as having two modes.

In summary, to explore data using single variables and identify unimodal or bimodal characteristics:

1. For **numerical variables**, use **histograms** and **density plots** to visualise the shape of the distribution and look for the number of peaks.
2. For **categorical variables**, use **barcharts** to examine the frequencies of each category and identify the mode(s).

Remember that the interpretation of "bimodal" might be slightly different for categorical data compared to continuous data.

10. Explore data using two Variables: Line plots, Scatter Plots, smoothing cures, Bar charts

You're asking to explore data using two variables, focusing on the visualisation techniques of **line plots, scatter plots, smoothing curves, and bar charts**. Based on the information in the sources, here's how we can approach this in R:

When exploring the relationship between two variables, the choice of visualisation technique depends on the types of the variables (numerical or categorical).

### **Line Plots**

**Line plots** are primarily used to visualise the relationship between two numerical variables where one variable, typically on the x-axis, represents a continuous scale, such as time, and the other, on the y-axis, represents a value that changes over that scale. Line plots are excellent for identifying trends and patterns over a continuous variable.

**Creating Line Plots in R:**

The plot() function with type="l" or the lines() function can be used to create line plots.

# Example using data from source (simulated time series data)

time <- 1:5

stock\_price <- c(10, 12, 15, 13, 16)

plot(time, stock\_price, type="l", main="Stock Price Over Time", xlab="Time", ylab="Stock Price")

# Adding another line to an existing plot (from source)

x <- c(1, 2, 3)

y <- c(1, 3, 8)

plot(x, y, type="l", main="Two Lines", xlab="X", ylab="Y")

lines(c(1.5, 2.5), c(3, 3), type="b", col="red") # Adding a line segment with points

The plot() function can create a new graph, while lines() adds lines to an existing graph. The type argument controls how the data is displayed ("l" for lines, "p" for points, "b" for both).

### **Scatter Plots**

**Scatter plots** are used to visualise the relationship between two numerical variables. Each point on the plot represents a pair of values for the two variables, allowing you to see if there is any correlation or pattern between them.

**Creating Scatter Plots in R:**

The plot() function with default settings (or type="p") is used to create scatter plots.

# Example using data from source

heightx <- c(1, 2, 3)

weighty <- c(1, 3, 4)

plot(heightx, weighty, main="Scatter Plot of Weight vs Height", xlab="Height", ylab="Weight")

# Adding points to an existing plot (from source)

x1 <- c(1, 2, 3)

y1 <- c(1, 3, 8)

plot(x1, y1, pch ="+", type="b", main="Scatter and Points", xlab="X", ylab="Y")

x2 <- c(1.5, 2.5)

y2 <- c(3, 3)

points(x2, y2, pch=25, col="blue") # Adding blue filled squares

The pch argument controls the symbol used for the points.

### **Smoothing Curves**

**Smoothing curves** are used to highlight the underlying trend in a scatter plot, especially when the relationship between the two numerical variables is not strictly linear. They help to visualise the general pattern by fitting a smooth line through the data points.

**Creating Smoothing Curves in R:**

The lowess() and loess() functions can be used to fit smoothing curves, and the lines() function can add these curves to a scatter plot.

# Example using lowess smoothing (from source)

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

y <- c(1, 3, 4, 1, 3)

df <- data.frame(x, y)

plot(df, pch=8, main="Scatter Plot with Lowess Smoothing", xlab="X", ylab="Y")

lines(lowess(df), col="yellow")

# Example using curve for explicit functions (can also represent a smooth trend)

curve((x^2 + 1)^0.5, 0, 5, main="Curve Example", xlab="X", ylab="f(x)")

lowess() is typically used for univariate smoothing (one predictor), while loess() can handle multivariate data. The curve() function can plot mathematical functions, which can sometimes represent a smoothed trend.

### **Bar Charts**

**Bar charts** are commonly used to compare a numerical variable across different categories of a categorical variable. The height (or length) of each bar represents the value of the numerical variable for that specific category. While less direct for showing relationships between two numerical variables, they can illustrate how a numerical variable changes based on the levels of a categorical variable. Bar charts can also be adapted to show the relationship between two categorical variables using techniques like stacked or grouped bar charts (though the sources don't explicitly detail this).

**Creating Bar Charts in R:**

The barplot() function is used to create bar charts.

# Example comparing a numerical variable across categories (from source)

H <- c(50, 40, 10)

barplot(H, names.arg=c("mpcs", "mscs", "dscs"), main="Bar Chart of Counts by Category", xlab="Category", ylab="Count", col=c("red", "green", "blue"))

# Using a frequency table of a categorical variable (as discussed previously)

df <- data.frame(

"Category" = c("A", "B", "A", "C", "B", "A"),

"Value" = c(10, 15, 12, 8, 14, 11)

)

category\_summary <- aggregate(Value ~ Category, data=df, FUN=sum)

barplot(category\_summary$Value, names.arg=category\_summary$Category, main="Sum of Value by Category", xlab="Category", ylab="Sum of Value")

The barplot() function takes a vector or matrix of values. names.arg specifies the names for each bar, and col sets the bar colours.

In summary, when exploring data using two variables:

* Use **line plots** to visualise trends over a continuous variable.
* Use **scatter plots** to examine the relationship between two numerical variables.
* Use **smoothing curves** to highlight the underlying patterns in scatter plots.
* Use **bar charts** to compare a numerical variable across categories of a categorical variable.

The choice of plot depends on the nature of your two variables and the type of relationship you want to investigate.

**11. Explore and implement commands using dplyr package**

You're asking to **explore and implement commands using the dplyr package** in R, drawing on the information in the sources and our conversation history.

The dplyr package is a very popular and powerful R package designed for **managing and manipulating data frames** in a more intuitive and efficient way compared to base R functions. It provides a set of **"verbs"** that correspond to common data manipulation tasks.

### **Installing and Loading dplyr**

Before using dplyr, you need to **install it** if you haven't already. You can do this using the install.packages() function:

install.packages("dplyr")

You only need to install a package once. After installation, you need to **load the package** into your current R session using the library() function:

library(dplyr)

You might see some warning messages when loading dplyr because it has functions with the same names as functions in other packages (like stats and base). For now, you can generally ignore these warnings.

### **Key dplyr Verbs**

Here are some of the most important dplyr verbs, with explanations and examples drawn from your sources and our previous conversation:

**select()**: Used to **choose specific columns** from a data frame.  
  
 # Assuming you have a data frame called 'chicago' (as used in our previous conversation)

# Select the 'city' and 'tmpd' columns

selected\_data <- select(chicago, city, tmpd)

head(selected\_data)

# Select all columns from 'city' to 'dptp' (inclusive)

select(chicago, city:dptp)

# Select all columns except those from 'city' to 'dptp'

select(chicago, -(city:dptp))

# Select columns based on patterns (e.g., those ending with '2')

# select(chicago, ends\_with("2"))

**filter()**: Used to **select specific rows** from a data frame based on one or more conditions.  
  
 # Filter for rows where the temperature ('tmpd') is less than 20

cold\_days <- filter(chicago, tmpd < 20)

head(cold\_days)

# Filter for rows where city is 'chic' AND temperature is greater than 30

filter(chicago, city == "chic" & tmpd > 30)

# Filter for rows where city is either 'chic' OR temperature is less than 10

filter(chicago, city == "chic" | tmpd < 10)

**arrange()**: Used to **reorder the rows** of a data frame based on the values of one or more columns.  
  
 # Arrange the data frame by temperature ('tmpd') in ascending order

arranged\_by\_temp <- arrange(chicago, tmpd)

head(arranged\_by\_temp)

# Arrange by temperature in descending order using the desc() function

arranged\_by\_temp\_desc <- arrange(chicago, desc(tmpd))

head(arranged\_by\_temp\_desc)

# Arrange by 'city' first, then by 'tmpd'

arranged\_by\_city\_temp <- arrange(chicago, city, tmpd)

head(arranged\_by\_city\_temp)

**rename()**: Used to **change the names of columns** in a data frame.  
  
 # Rename the 'tmpd' column to 'temperature' and 'dptp' to 'dewpoint'

chicago\_renamed <- rename(chicago, temperature = tmpd, dewpoint = dptp)

head(chicago\_renamed)

* The syntax is new\_name = old\_name.

**mutate()**: Used to **add new columns** to a data frame or to **modify existing columns**.  
  
 # Add a new column 'temp\_fahrenheit' by converting 'tmpd' from Celsius to Fahrenheit

chicago\_with\_fahrenheit <- mutate(chicago, temp\_fahrenheit = tmpd \* 9/5 + 32)

head(chicago\_with\_fahrenheit)

# Create a combined column 'city\_temp' by pasting 'city' and 'tmpd'

chicago\_combined <- mutate(chicago, city\_temp = paste(city, tmpd, sep = "-"))

head(chicago\_combined)

# You can also use mutate to perform more complex transformations

# Example: detrending the 'pm25' variable by subtracting its mean

# chicago\_detrended <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))

# head(chicago\_detrended)

**summarise()** (or summarize()): Used to **calculate summary statistics** for a data frame. It often works in conjunction with group\_by().  
  
 # Calculate the average temperature ('tmpd') for the entire data frame

average\_temp <- summarise(chicago, average\_temperature = mean(tmpd, na.rm = TRUE))

print(average\_temp)

# Calculate the standard deviation of 'tmpd'

sd\_temp <- summarise(chicago, sd\_temperature = sd(tmpd, na.rm = TRUE))

print(sd\_temp)

**group\_by()**: Used to **group the rows** of a data frame based on the values of one or more columns. Subsequent summarise() operations will then be performed on each group.  
  
 # Group the 'chicago' data frame by 'city'

grouped\_by\_city <- group\_by(chicago, city)

# Calculate the average temperature for each city

average\_temp\_per\_city <- summarise(grouped\_by\_city, mean\_temperature = mean(tmpd, na.rm = TRUE))

print(average\_temp\_per\_city)

# Group by 'city' and then calculate both the mean and standard deviation of temperature

summary\_per\_city <- grouped\_by\_city %>%

summarise(mean\_temp = mean(tmpd, na.rm = TRUE),

sd\_temp = sd(tmpd, na.rm = TRUE),

n = n()) # n() counts the number of observations in each group

print(summary\_per\_city)

### **The Pipe Operator (%>%)**

The pipe operator %>%, which comes with dplyr (though technically from the magrittr package), is a very handy tool for **chaining multiple dplyr operations together** in a more readable and logical sequence. It takes the output of the previous operation and passes it as the first argument to the next function.

# Without the pipe:

result\_without\_pipe <- summarise(group\_by(filter(select(chicago, city, tmpd), tmpd > 20), city), average\_temp = mean(tmpd, na.rm = TRUE))

print(result\_without\_pipe)

# With the pipe:

result\_with\_pipe <- chicago %>%

select(city, tmpd) %>%

filter(tmpd > 20) %>%

group\_by(city) %>%

summarise(average\_temp = mean(tmpd, na.rm = TRUE))

print(result\_with\_pipe)

The version using the pipe is generally easier to read and understand because it reflects the sequence of operations you're performing on the data.

By combining these dplyr verbs with the pipe operator, you can perform complex data manipulation tasks in a clear and concise manner. The dplyr package significantly simplifies working with data frames in R.

**12. Download a dataset and work on basic data manipulation followed by inferential statistics.**

You're asking to **download a dataset and perform basic data manipulation followed by inferential statistics**, drawing on the information from the sources and our conversation history.

### **Downloading a Dataset in R**

The sources provide several examples of how to access datasets:

* **Built-in R datasets:** You can access datasets built into R using the data() command. For example, data("CO2") loads the CO2 uptake dataset. You can get more information about these datasets using help(dataset\_name).

**Reading from local files:** The read.table() and read.csv() functions can be used to read data from .txt and .csv files, respectively. The readRDS() function can read R objects saved in a specific format. You'll need to download the file to your local machine first. For example:  
 airqual <- read.table("C:/Desktop/airquality.txt") #

chicago <- readRDS("chicago.rds") #

* **Reading from URLs:** The url() function can open a connection to a webpage, allowing you to read data directly from the internet. This is mentioned as a good practice for reproducible analysis.
* **Using packages:** Many R packages provide access to specific datasets. For example, the sklearn.datasets in Python (mentioned in the context of data science) suggests that R might have similar packages, although specific examples aren't given in your sources for downloading via a package.

For this exercise, let's consider using a dataset that can be downloaded from a URL, aligning with the reproducible analysis suggestion. Source mentions a dataset available from a URL for dplyr examples: http://www.biostat.jhsph.edu/~rpeng/leanpub/rprog/chicago\_data.zip. You would first download and unzip this file, then read it into R.

# Download the data (you might need to set your working directory first using setwd())

download.file("http://www.biostat.jhsph.edu/~rpeng/leanpub/rprog/chicago\_data.zip", destfile = "chicago\_data.zip")

unzip("chicago\_data.zip")

# Read the RDS file

chicago <- readRDS("chicago.rds") #

### **Basic Data Manipulation in R**

Once you have loaded your data into R, you can perform basic data manipulation. The sources highlight several key functions and concepts:

* **Inspecting data:** Use functions like dim() to see the dimensions (rows and columns) and str() to see the structure of the data frame, including variable types.
* **Subsetting data:** You can extract subsets of data using square brackets [] with row and column indices or names. Logical conditions can also be used for filtering rows.
* **Using the dplyr package:** Source introduces the dplyr package, which provides a more intuitive grammar for data manipulation. Key verbs include:

select(): To choose specific columns.  
 library(dplyr) #

chicago\_subset <- select(chicago, city, tmpd) # Select 'city' and 'tmpd' columns

head(chicago\_subset)

filter(): To select rows based on conditions.  
 cold\_days <- filter(chicago, tmpd < 20) # Filter for days with temperature less than 20

head(cold\_days)

arrange(): To reorder rows.  
 arranged\_by\_temp <- arrange(chicago, tmpd) # Arrange by temperature

head(arranged\_by\_temp)

rename(): To change column names.  
 chicago\_renamed <- rename(chicago, temperature = tmpd) # Rename 'tmpd' to 'temperature'

head(chicago\_renamed)

mutate(): To add new columns or modify existing ones.  
 chicago\_with\_fahrenheit <- mutate(chicago, temp\_fahrenheit = tmpd \* 9/5 + 32) # Add a temperature in Fahrenheit

head(chicago\_with\_fahrenheit)

* + summarise(): To generate summary statistics.

### **Inferential Statistics in R**

The sources touch upon inferential statistics and related R functionalities:

* **Inferential statistics overview:** Source defines inferential statistics as drawing conclusions about a population based on a sample and mentions hypothesis testing, t-tests, z-tests, confidence intervals, and regression analysis as common methodologies.
* **T-tests:** Source specifically mentions the t-test as a method to test for a statistically significant difference in the means of two groups.
* **Linear Regression:** Sources mention linear regression as a technique for predicting continuous values and provide examples like predicting house prices or temperature. Source also mentions functions like lm() and predict() for implementing linear regression in R.
* **stats package:** Source indicates that the dplyr package can mask functions from the stats package, implying that the stats package contains functions relevant to statistical analysis.

Let's perform a simple inferential statistics task using the chicago dataset. Suppose we want to know if there's a statistically significant difference in temperature (tmpd) between the first and second half of the dataset (we'll treat this as two arbitrary groups).

# Create two groups (first half and second half of the data)

n <- nrow(chicago)

group1 <- chicago[1:(n/2), ]

group2 <- chicago[((n/2)+1):n, ]

# Perform an independent samples t-test to compare the means of 'tmpd'

t\_test\_result <- t.test(group1$tmpd, group2$tmpd)

print(t\_test\_result)

The output of the t.test() function will provide information about the t-statistic, degrees of freedom, p-value, and confidence interval for the difference in means. If the p-value is below a chosen significance level (e.g., 0.05), you might conclude that there is a statistically significant difference in the average temperature between the two groups.

As another example, let's say we want to see if there's a linear relationship between temperature (tmpd) and ozone levels (if available in the chicago dataset - if not, we'd need a dataset with both variables). Assuming a column named o3 exists:

# Fit a simple linear regression model

linear\_model <- lm(o3 ~ tmpd, data = chicago) # Formula: o3 is predicted by tmpd

# Get a summary of the model

summary(linear\_model)

The summary() output will provide information about the model coefficients, their statistical significance (p-values), and the overall fit of the model (R-squared).

These examples demonstrate basic data manipulation and the application of a t-test and linear regression in R, drawing upon the concepts and functions mentioned in your provided sources. Remember to adapt the dataset loading and the specific inferential tests based on the actual data you download and the research questions you want to address.