SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chennai-603 203

FACULTY OF MANAGEMENT

MBG24108L – PROGRAMMING LANGUAGES FOR BUSINESS DECISIONS

LAB MANUAL

Academic Year -2025-26

Name of the Student	
Register Number	
Name of The Programme	
Year & Semester	**I YEAR & I SEMESTER**
Section	

Prepared by

1. **Dr.V.M.Shenbagaraman**
2. **Dr.P.Saravanan**
3. **Dr.N.Arun Fred**
4. **Mr.R.Vijay Anand**
5. **Mrs.S.Saranya**

Faculty Coordinators

1. **Dr.V.M.Shenbagarama**
2. **Dr.P.Saravanan**
3. **Dr.S.K.Manivannan**
4. **Dr.J.Dinesh**
5. **Dr.G.Kumar**
6. **Dr.R.Srilalitha**
7. **Dr.N.Arun Fred**
8. **Dr.G.R.Kanmani**
9. **Dr.Saithu Mohammed**

SL. NO	DATE	TITLE OF THE EXERCISE
1		INTRODUCTION TO PYTHON & JUPYTER NOTEBOOK
2		INSTALLING PYTHON AND JUPYTER NOTEBOOK USING ANACONDA DISTRIBUTION
3		PRINTING OPTIONS IN PYTHON
4		VARIABLES IN PYTHON
5		DATA TYPES IN PYTHON
6		ARITHMETIC OPERATORS IN PYTHON
7		PROGRAM TO FIND AREA OF TRIANGLE
8		PROGRAM TO FIND SQUARE ROOT
9		LOGICAL OPERATORS IN PYTHON
10		SWAPPING OF TWO NUMBERS USING THIRD VARIABLE
11		PYTHON PROGRAM TO FIND THE LARGEST AMONG THREE NUMBERS
12		CONTROL STRUCTURES IN PYTHON
13		PYTHON PROGRAM DISPLAY THE MULTIPLICATION TABLE
14		FUNCTIONS IN PYTHON
15		PYTHON PROGRAM TO MAKE A SIMPLE CALCULATOR
16		STRING OPERATIONS IN PYTHON
17		PYTHON PROGRAM TO CHECK WHETHER A STRING IS PALINDROME OR NOT
18		PYTHON PROGRAM TO COMPUTE THE POWER OF A NUMBER
19		PYTHON PROGRAM TO COUNT THE NUMBER OF DIGITS PRESENT IN A NUMBER
20		INSTALLING & USING PACKAGES IN PYTHON
21		FILE HANDLING IN PYTHON
22		LIST IN PYTHON
23		TUPLE IN PYTHON
24		DICTIONARY IN PYTHON
25		SETS IN PYTHON
26		DATA SLICING IN PYTHON

27		IMPORTING CSV DATA IN PANDAS
28		IMPORTING EXCEL DATA IN PANDAS
29		INTRODUCTION TO R PROGRAMMING
30		HOW TO INSTALL R PROGRAMMING
31		UNDERSTANDING BASIC COMMANDS IN R
32		IMPORTING & EXPORTING DATA IN R
33		EXPLORING THE DATA USING R STUDIO
34		CREATING PIE CHART IN R
35		CREATING HISTOGRAM IN R
36		CREATING BARPLOT IN R
37		HANDLING GGPLOT PACKAGE IN R STUDIO
38		CREATING FREQUENCY DISTRIBUTION IN R
39		DESCRIPTIVE STATISTICS USING R
40		CORRELATION IN R

Exercise No: 1

Date:

INTRODUCTION TO PYTHON & JUPYTER NOTEBOOK

Python Introduction

Python was created 1991 with focus on code readability and express concepts in fewer lines of code.

- Simple and readable syntax makes it beginner-friendly.
- Runs seamlessly on Windows, macOS and Linux.
- Includes libraries for tasks like web development, data analysis and machine learning.
- Variable types are determined automatically at runtime, simplifying code writing.
- Supports multiple programming paradigms, including object-oriented, functional and procedural programming.
- Free to use, distribute and modify.

Understanding Hello World Program in Python

Hello, World! in python is the first python program which we learn when we start learning any program. It’s a simple program that displays the message “Hello, World!” on the screen.

Hello World Program

Here’s the “Hello World” program:

```
# This is a comment. It will not be executed.
print("Hello, World!")
```

Output

Hello, World!

How does this work:

- print() is a built-in Python function that tells the computer to show something on the screen.
- The message "Hello, World!" is a string, which means it's just text. In Python, strings are always written inside quotes (either single ' or double ").
- Anything after # in a line is a comment. Python ignores comments when running the code, but they help people understand what the code is doing.
- Comments are helpful for explaining code, making notes or skipping lines while testing.

We can also write multi-line comments using triple quotes:

"""

This is a multi-line comment.

It can be used to describe larger sections of code.

"""

*To understand comments in detail, refer to article: **Comments**.*

Indentation in Python

In Python, Indentation is used to define blocks of code. It tells the Python interpreter that a group of statements belongs to a specific block. All statements with the same level of indentation are considered part of the same block. Indentation is achieved using whitespace (spaces or tabs) at the beginning of each line. The most common convention is to use 4 spaces or a tab, per level of indentation.

Example:

```
print("I have no indentation")
  print("I have tab indentaion")
```

Output:

Hangup (SIGHUP)
File "/home/guest/sandbox/Solution.py", line 3
print("I have tab indentaion")
IndentationError: unexpected indent

Explanation:

- first **print** statement has no indentation, so it is correctly executed.
- second **print** statement has **tab indentation**, but it doesn't belong to a new block of code. Python expects the indentation level to be consistent within the same block. This inconsistency causes an **IndentationError**.

Famous Application Built using Python

- **YouTube:** World's largest video-sharing platform uses Python for features like video streaming and backend services.
- **Instagram:** This popular social media app relies on Python's simplicity for scaling and handling millions of users.
- **Spotify:** Python is used for backend services and machine learning to personalize music recommendations.
- **Dropbox:** The file hosting service uses Python for both its desktop client and server-side operations.
- **Netflix:** Python powers key components of Netflix's recommendation engine and content delivery systems (CDN).
- **Google:** Python is one of the key languages used in Google for web crawling, testing and data analysis.
- **Uber:** Python helps Uber handle dynamic pricing and route optimization using machine learning.
- **Pinterest:** Python is used to process and store huge amounts of image data efficiently.

What can we do with Python?

Python is used for:

- Web Development: Frameworks like Django, Flask.
- Data Science and Analysis: Libraries like Pandas, NumPy, Matplotlib.
- Machine Learning and AI: TensorFlow, PyTorch, Scikit-learn.
- Automation and Scripting: Automate repetitive tasks.
- Game Development: Libraries like Pygame.
- Web Scraping: Tools like BeautifulSoup, Scrapy.
- Desktop Applications: GUI frameworks like Tkinter, PyQt.
- Scientific Computing: SciPy, SymPy.
- Internet of Things (IoT): MicroPython, Raspberry Pi.
- DevOps and Cloud: Automation scripts and APIs.
- Cybersecurity: Penetration testing and ethical hacking tools.

What is Jupyter Notebook?

At its core, a notebook is a document that blends code and its output seamlessly. It allows you to run code, display the results, and add explanations, formulas, and charts all in one place. This makes your work more transparent, understandable, and reproducible.

Jupyter Notebook is an incredibly powerful tool for interactively developing and presenting data science projects. It combines code, visualizations, narrative text, and other rich media into a single document, creating a cohesive and expressive workflow.

Jupyter Notebooks have become an essential part of the data science workflow in companies and organizations worldwide. They enable data scientists to explore data, test hypotheses, and share insights efficiently.

As an open-source project, Jupyter Notebooks are completely free. You can download the software directly from the Project Jupyter website or as part of the Anaconda data science toolkit.

While Jupyter Notebooks support multiple programming languages, this article will focus on using Python, as it is the most common language used in data science. However, it's worth noting that other languages like R, Julia, and Scala are also supported.

If your goal is to work with data, using Jupyter Notebooks will streamline your workflow and make it easier to communicate and share your results.

In the next exercise, you can learn how to install python and jupyter notebook using Anaconda Distribution.

Exercise No: 2

Date:

INSTALLING PYTHON AND JUPYTER NOTEBOOK USING ANACONDA DISTRIBUTION

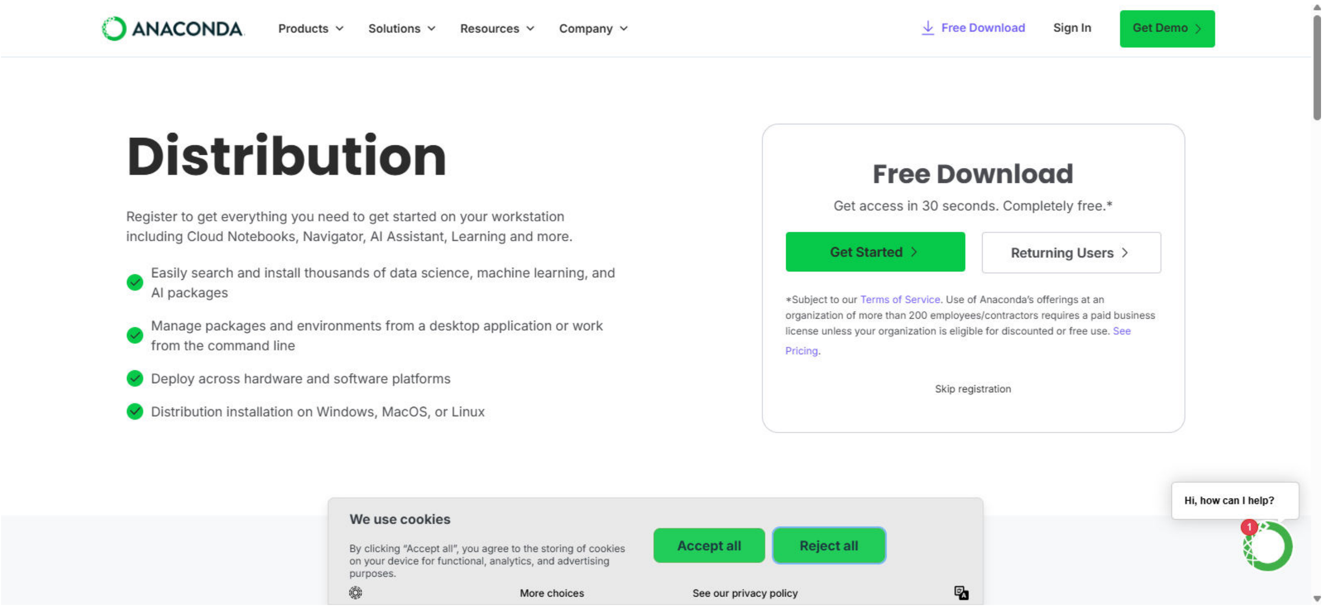
Aim:

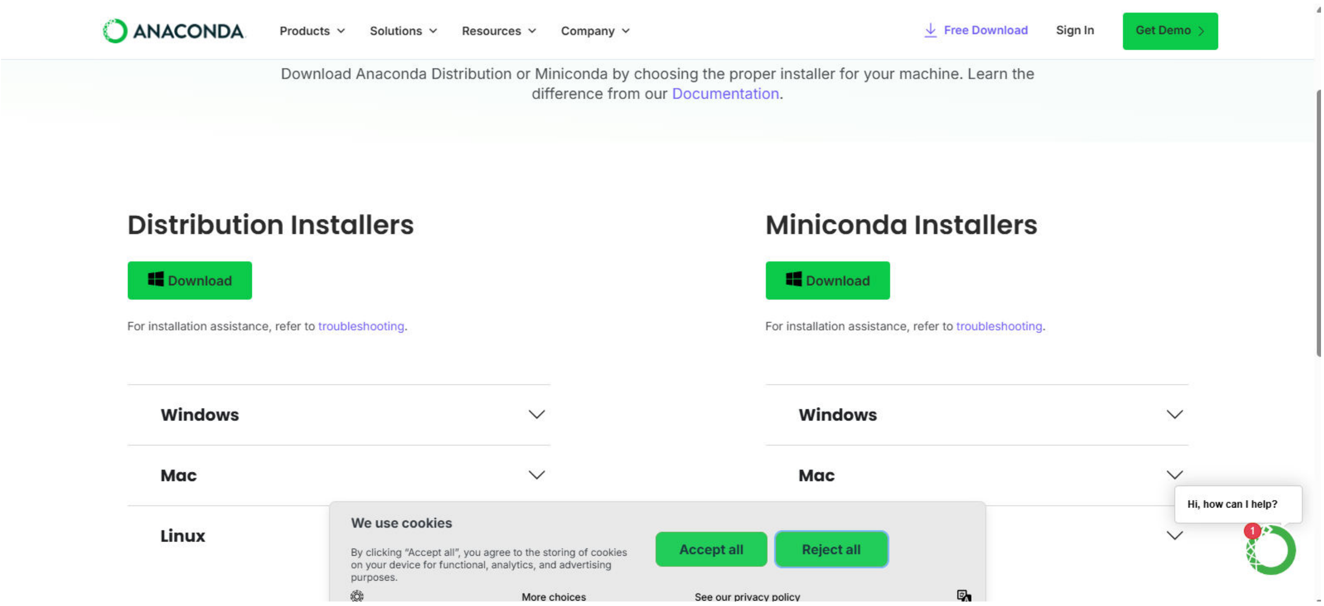
To learn how to download and install python using Anaconda Distribution.

Procedure:

Installing Anaconda on Windows 11

Step 1: Go to https://anaconda.com/download. Click Skip Registration.



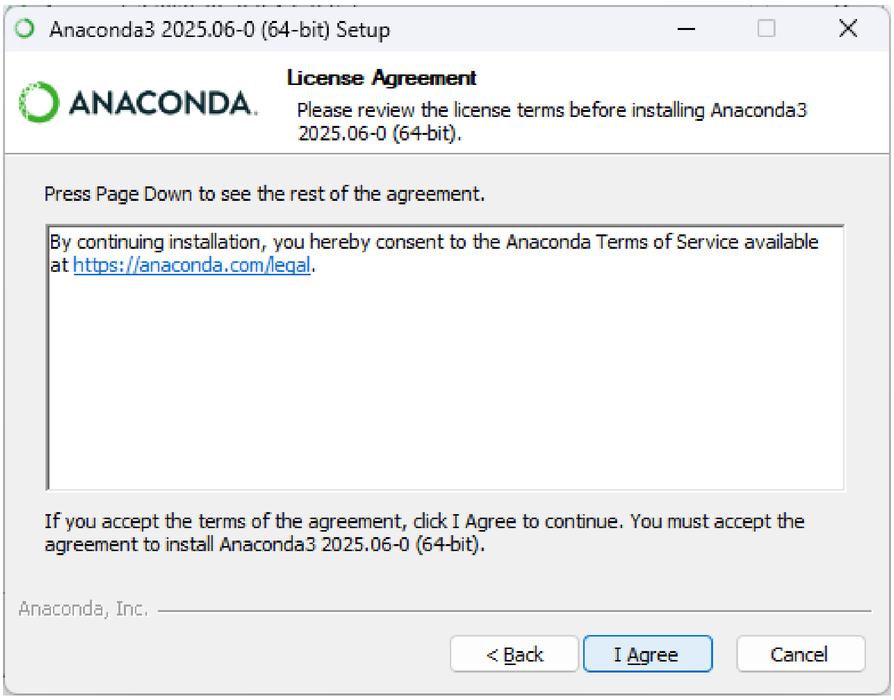


Click the "Download" button under the Distribution Installers to start downloading the latest Anaconda installer for Windows.

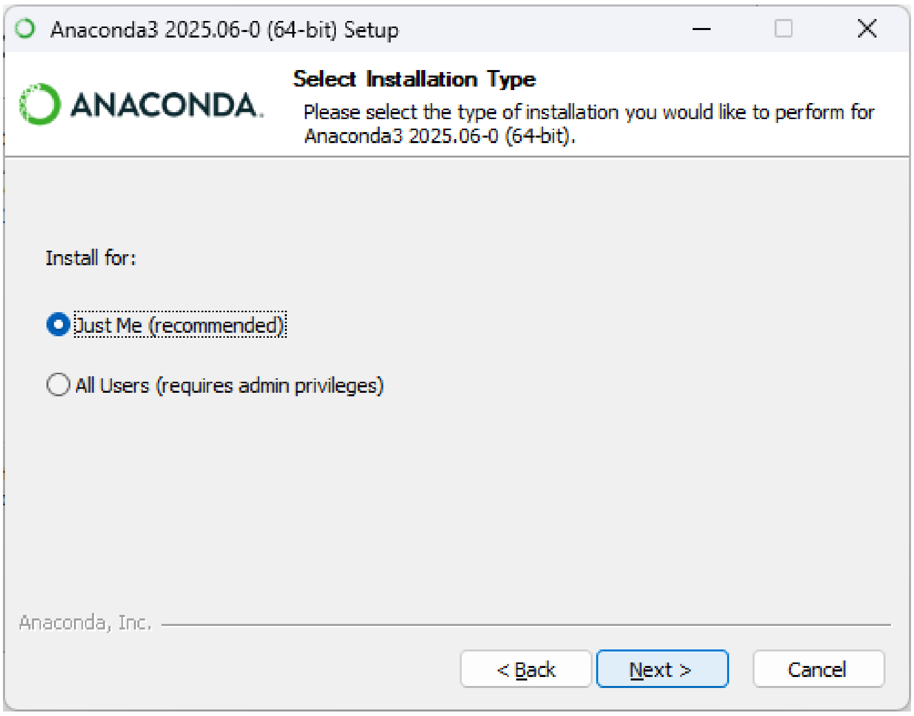
Step 2: Run the Anaconda Installer

Once the installer is downloaded, locate the executable file (usually named something like " “Anaconda3-2025.06-0-Windows-x86_64.exe" for 64-bit) and double-click it to run the installer. Windows may ask if you want to allow the app to make changes to your device. Click "Yes" to proceed. The Anaconda installation wizard will appear. Click "Next" to continue.

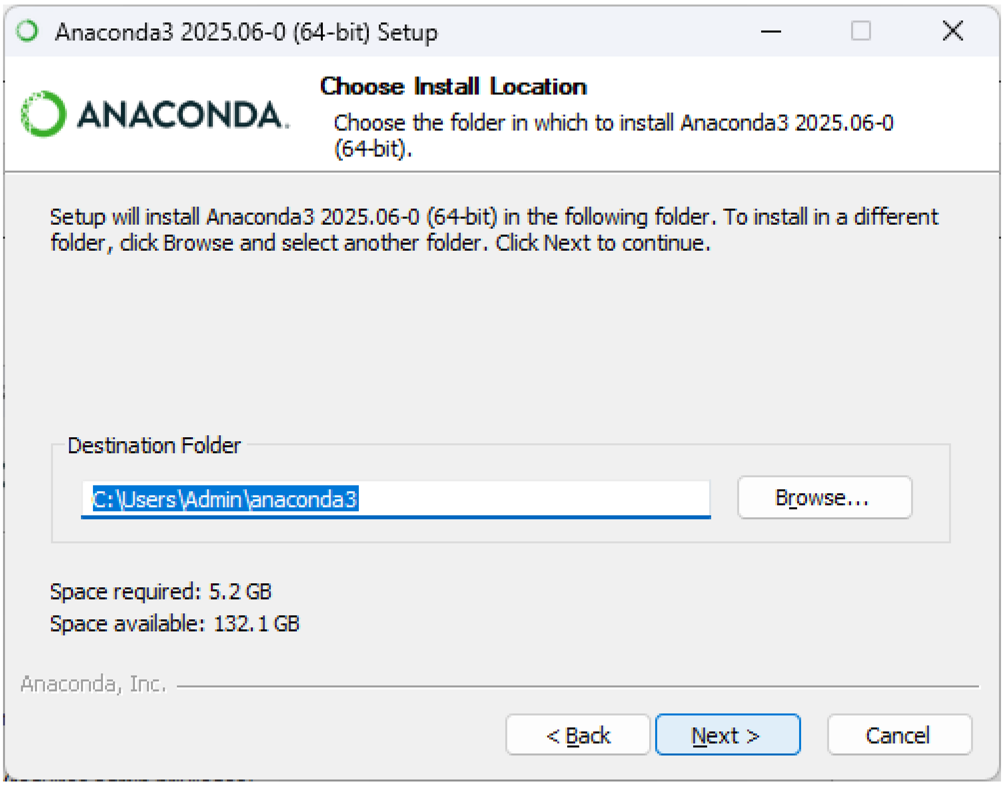
Step 3: Read the Anaconda Individual Edition License Agreement carefully. If you agree to the terms, select "I accept the terms in the License Agreement" and click "Next."



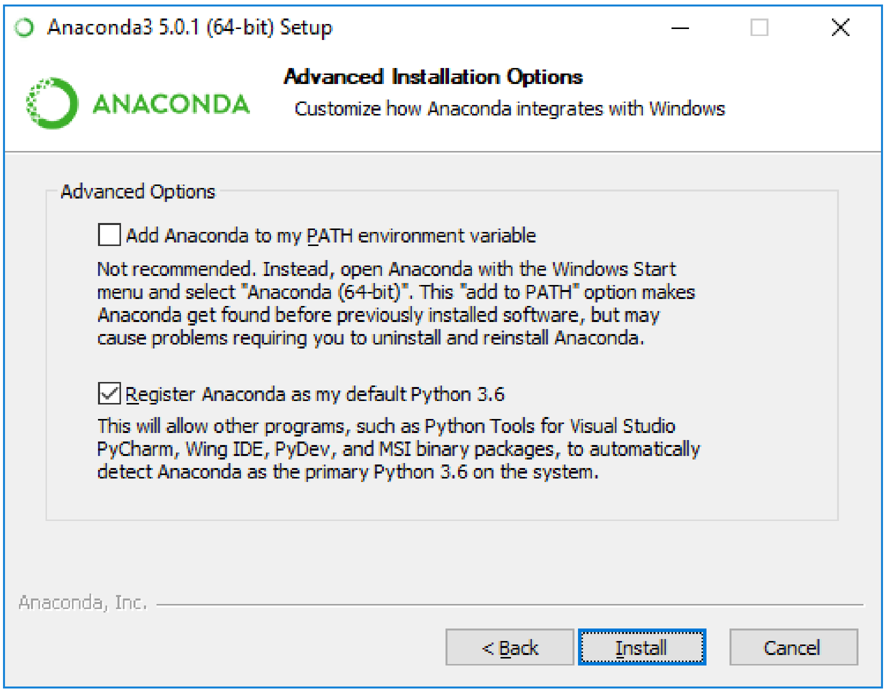
Step 4: Choose the installation type. For most users, the default option, "Just Me (recommended)," is suitable. Click "Next."



Step 5: Select the destination folder where Anaconda will be installed. You can use the default location or choose a different one by clicking "Browse." Once you've made your choice, click "Next."



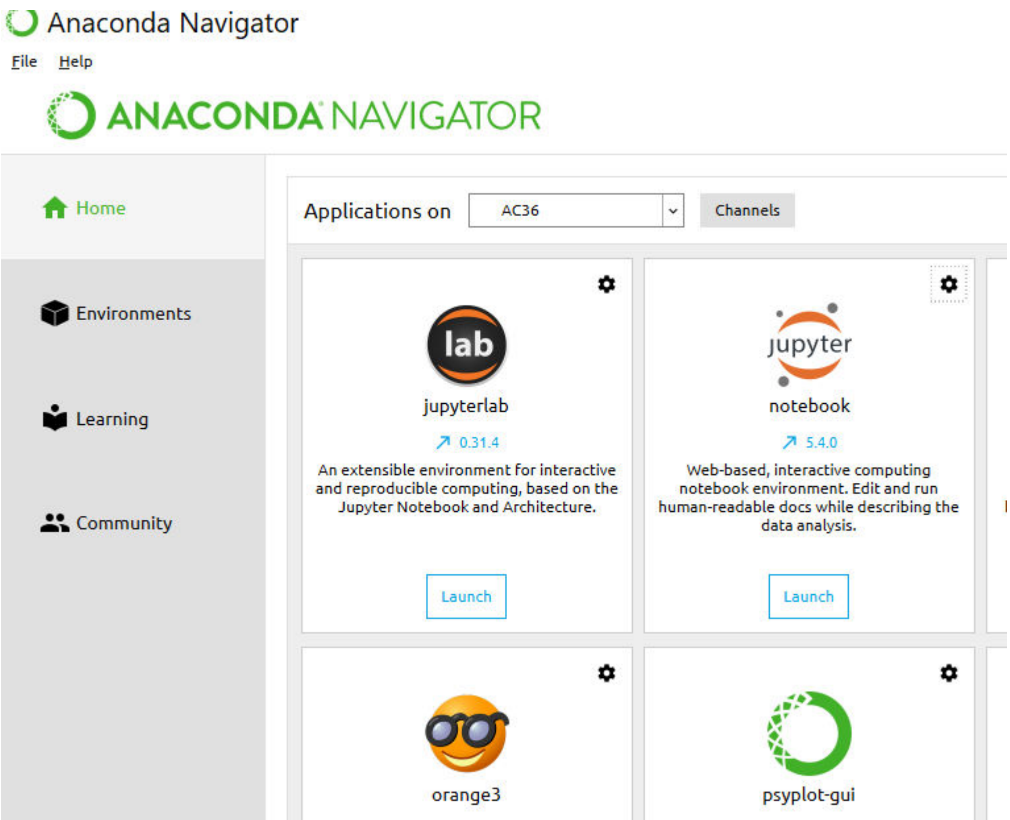
Step 6: At the Advanced Installation Options screen, I recommend that you do not check "Add Anaconda to my PATH environment variable"



The installation process will commence. This may take a few minutes as Anaconda installs Python, packages, and dependencies.

Running Jupyter

On Windows, you can run or launch Jupyter via the shortcut Anaconda adds to your start menu, which will open a new tab in your default web browser that should look something like the following screenshot:



This isn’t a notebook just yet, but don’t panic! There’s not much to it. This is the Notebook Dashboard, specifically designed for managing your Jupyter Notebooks. Think of it as the launchpad for exploring, editing and creating your notebooks.

Be aware that the dashboard will give you access only to the files and sub-folders contained within Jupyter’s start-up directory (i.e., where Jupyter or Anaconda is installed). However, the start-up directory can be changed.

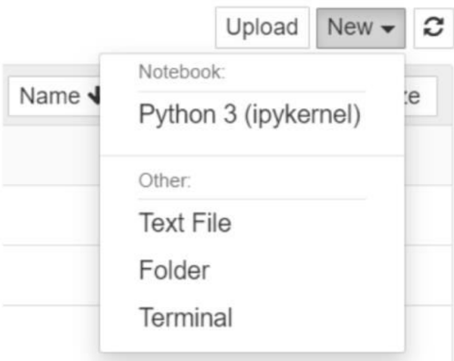
It is also possible to start the dashboard on any system via the command prompt (or terminal on Unix systems) by entering the command jupyter notebook; in this case, the current working directory will be the start-up directory.

With Jupyter Notebook open in your browser, you may have noticed that the URL for the dashboard is something like https://localhost:8888/tree. Localhost is not a website, but indicates that the content is being served from your local machine: your own computer.

Jupyter’s Notebooks and dashboard are web apps, and Jupyter starts up a local Python server to serve these apps to your web browser, making it essentially platform-independent and opening the door to easier sharing on the web.

(If you don't understand this yet, don't worry — the important point is just that although Jupyter Notebooks opens in your browser, it's being hosted and run on your local machine. Your notebooks aren't actually on the web until you decide to share them.)

The dashboard’s interface is mostly self-explanatory — though we will come back to it briefly later. So what are we waiting for? Browse to the folder in which you would like to create your first notebook, click the “New” drop-down button in the top-right and select “Python 3(ipykernel)”:



Hey presto, here we are! Your first Jupyter Notebook will open in new tab — each notebook uses its own tab because you can open multiple notebooks simultaneously.

If you switch back to the dashboard, you will see the new file Untitled.ipynb and you should see some green text that tells you your notebook is running.

What is an .ipynb File?

The short answer: each .ipynb file is one notebook, so each time you create a new notebook, a new .ipynb file will be created.

The longer answer: Each .ipynb file is an **Interactive PYthon NoteBook** text file that describes the contents of your notebook in a format called JSON. Each cell and its contents, including image attachments that have been converted into strings of text, is listed therein along with some metadata.

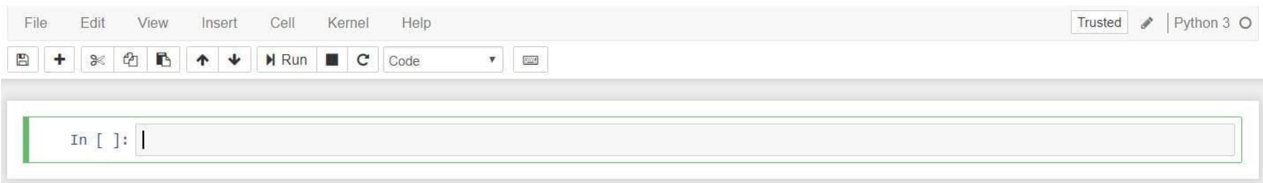
You can edit this yourself — if you know what you are doing! — by selecting "Edit > Edit Notebook Metadata" from the menu bar in the notebook. You can also view the contents of your notebook files by selecting "Edit" from the controls on the dashboard.

However, the key word there is *can*. In most cases, there's no reason you should ever need to edit your notebook metadata manually.

The Notebook Interface

Now that you have an open notebook in front of you, its interface will hopefully not look entirely alien. After all, Jupyter is essentially just an advanced word processor.

Why not take a look around? Check out the menus to get a feel for it, especially take a few moments to scroll down the list of commands in the command palette, which is the small button with the keyboard icon (or Ctrl + Shift + P).



There are two key terms that you should notice in the menu bar, which are probably new to you: **Cell** and **Kernel**. These are key terms for understanding how Jupyter works, and what makes it more than just a word processor. Here's a basic definition of each:

- The **kernel** in a Jupyter Notebook is like the brain of the notebook. It's the "computational engine" that runs your code. When you write code in a notebook and ask it to run, the kernel is what takes that code, processes it, and gives you the results. Each notebook is connected to a specific kernel that knows how to run code in a particular programming language, like Python.
- A **cell** in a Jupyter Notebook is like a block or a section where you write your code or text (notes). You can write a piece of code or some explanatory text in a cell, and when you run it, the code will be executed, or the text will be rendered (displayed). Cells help you organize your work in a notebook, making it easier to test small chunks of code and explain what's happening as you go along.

Cells

We'll return to kernels a little later, but first let's come to grips with cells. Cells form the body of a notebook. In the screenshot of a new notebook in the section above, that box with the green outline is an empty cell. There are two main cell types that we will cover:

- A **code cell** contains code to be executed in the kernel. When the code is run, the notebook displays the output below the code cell that generated it.
- A **Markdown cell** contains text formatted using [Markdown]() and displays its output in-place when the Markdown cell is run.

The first cell in a new notebook defaults to a code cell. Let's test it out with a classic "Hello World!" example.

Type print('Hello World!') into that first cell and click the Run button in the toolbar above or press Ctrl + Enter on your keyboard.

The result should look like this:

When we run the cell, its output is displayed directly below the code cell, and the label to its left will have changed from In [ ] to In [1].

Like the contents of a cell, the output of a code cell also becomes part of the document. You can always tell the difference between a code cell and a Markdown cell because code cells have that special In [ ] label on their left and Markdown cells do not.

The "In" part of the label is simply short for "Input," while the label number inside [ ] indicates when the cell was executed on the kernel — in this case the cell was executed first.

Run the cell again and the label will change to In [2] because now the cell was the second to be run on the kernel. Why this is so useful will become clearer later on when we take a closer look at kernels.

From the menu bar, click Insert and select Insert Cell Below to create a new code cell underneath your first one and try executing the code below to see what happens. Do you notice anything different compared to executing that first code cell?

import time

time.sleep(3)

This code doesn't produce any output, but it does take three seconds to execute. Notice how Jupyter signifies when the cell is currently running by changing its label to In [*].

In general, the output of a cell comes from any text data specifically printed during the cell's execution, as well as the value of the last line in the cell, be it a lone variable, a function call, or something else. For example, if we define a function that outputs text and then call it, like so:

```
def say_hello(recipient):
    return 'Hello, {}!'.format(recipient)
say_hello('Tim')
```

We will get the following output below the cell:

'Hello, Tim!'

You'll find yourself using this feature a lot in your own projects, and we'll see more of its usefulness later on.

Keyboard Shortcuts

One final thing you may have noticed when running your cells is that their border turns blue after it's been executed, whereas it was green while you were editing it. In a Jupyter Notebook, there is always one "active" cell highlighted with a border whose color denotes its current mode:

- **Green outline** — cell is in "edit mode"
- **Blue outline** — cell is in "command mode"

So what can we do to a cell when it's in command mode? So far, we have seen how to run a cell with Ctrl + Enter, but there are plenty of other commands we can use. The best way to use them is with keyboard shortcuts.

Keyboard shortcuts are a very popular aspect of the Jupyter environment because they facilitate a speedy cell-based workflow. Many of these are actions you can carry out on the active cell when it's in command mode.

Below, you'll find a list of some of Jupyter's keyboard shortcuts. You don't need to memorize them all immediately, but this list should give you a good idea of what's possible.

- Toggle between command mode (blue) and edit mode (green) with Esc and Enter, respectively.

- While in command mode:
    - Scroll up and down your cells with your Up and Down keys.
    - Press A or B to insert a new cell above or below the active cell.
    - M will transform the active cell to a Markdown cell.
    - Y will set the active cell to a code cell.
    - D + D (D twice) will delete the active cell.
    - Z will undo cell deletion.
    - Hold Shift and press Up or Down to select multiple cells at once. With multiple cells selected, Shift + M will merge your selection. You can also click and Shift + Click in the margin to the left of your cells to select a range of them.
- While in edit mode:
    - Ctrl + Enter to run the current cell.
    - Shift + Enter to run the current cell and move to the next cell (or create a new one if there isn't a next cell)
    - Alt + Enter to run the current cell and insert a new cell below.
    - Ctrl + Shift + - will split the active cell at the cursor.
    - Ctrl + Click to create multiple cursors within a cell.

Go ahead and try these out in your own notebook. Once you're ready, create a new Markdown cell and we'll learn how to format the text in our notebooks.

Result:
The Python Anaconda Distribution was successfully downloaded and installed. It provided an integrated environment with Python, Jupyter Notebook, Spyder IDE, and pre-installed scientific libraries (such as NumPy, Pandas, and Matplotlib), making the system ready for Python programming, data science, and machine learning applications.

Exercise No: 3

Date:

PRINTING OPTIONS IN PYTHON

AIM

To write a python program to demonstrate all the printing options.

PROCEDURE

Step 1: Use the following print option used in python.

Printing the simple message

Printing the message along with the assigned name and age.

Printing the assigned message.

Printing the content in the separate output file (Output.txt).

Step 2: Print the results.

SOURCE CODE

```
print('Hello', 'world', sep=', ')
print('This is the end.', end=' ')
print('My message.')
name = 'Alice'
age = 30
print(f"My name is {name} and I am {age} years old.")
name = 'Bob'
age = 25
message = "My name is {} and I am {} years old.".format(name, age)
print(message)
name = 'Charlie'
age = 35
message = "My name is " + name + " and I am " + str(age) + " years old."
print(message)
name = 'David'
age = 40
message = "My name is %s and I am %d years old." % (name, age)
print(message)
```

OUTPUT

Hello, world

This is the end. My message.

My name is Alice and I am 30 years old.

My name is Bob and I am 25 years old.

My name is Charlie and I am 35 years old.

My name is David and I am 40 years old.

RESULT

Thus all the printing option for python has been demonstrated successfully.

Exercise No: 4

Date:

VARIABLES IN PYTHON

Aim:

To learn how to create and use variables in Python to store and manipulate different types of data.

Procedure:

1. Open Python (IDLE/Jupyter Notebook/any IDE).
2. Create variables of different data types (integer, float, string, boolean).
3. Assign values to variables and perform simple operations.
4. Print the variables to observe the stored values.
5. Use the type() function to check the data type of each variable.
6. Run the program and verify the output.

Program (Code):

```
# Exercise: Variables in Python

# Creating variables
product_name = "Laptop"      # String variable
quantity = 5                 # Integer variable
price = 45000.75             # Float variable
available = True             # Boolean variable

# Performing operations
total_cost = quantity * price

# Displaying variable values
print("Product:", product_name)
print("Quantity:", quantity)
print("Price per unit:", price)
print("Available:", available)
```

```
print("Total Cost:", total_cost)

# Checking data types
print("Data type of product_name:", type(product_name))
print("Data type of quantity:", type(quantity))
print("Data type of price:", type(price))
print("Data type of available:", type(available))
print("Data type of total_cost:", type(total_cost))
```

Output:

Product: Laptop

Quantity: 5

Price per unit: 45000.75

Available: True

Total Cost: 225003.75

Data type of product_name: <class 'str'>

Data type of quantity: <class 'int'>

Data type of price: <class 'float'>

Data type of available: <class 'bool'>

Data type of total_cost: <class 'float'>

Result:

The program was executed successfully. Variables of different data types were created, values were assigned, operations were performed, and data types were verified. This demonstrated how Python handles variables in business-related scenarios.

Exercise No: 5

Date:

DATA TYPES IN PYTHON

Aim:

To understand how to create, assign, and use variables in Python for storing and manipulating data.

Procedure:

1. Open Python (IDLE, Jupyter Notebook, or any Python IDE).
2. Create variables of different data types such as integer, float, string, and boolean.
3. Assign values to the variables.
4. Perform simple arithmetic operations using variables.
5. Use the print() function to display the variable values.
6. Use the type() function to check the data type of each variable.
7. Execute the program and observe the output.

Program (Code):

```
# Exercise: Variables in Python

# Declaring variables of different data types
student_name = "John"      # String variable
age = 22                   # Integer variable
gpa = 8.5                  # Float variable
is_present = True          # Boolean variable

# Performing an operation
next_year_age = age + 1

# Displaying variable values
print("Student Name:", student_name)
print("Age:", age)
print("GPA:", gpa)
```

```
print("Is Present:", is_present)
print("Age Next Year:", next_year_age)

# Checking data types
print("Type of student_name:", type(student_name))
print("Type of age:", type(age))
print("Type of gpa:", type(gpa))
print("Type of is_present:", type(is_present))
print("Type of next_year_age:", type(next_year_age))
```

Output:

Student Name: John

Age: 22

GPA: 8.5

Is Present: True

Age Next Year: 23

Type of student_name: <class 'str'>

Type of age: <class 'int'>

Type of gpa: <class 'float'>

Type of is_present: <class 'bool'>

Type of next_year_age: <class 'int'>

Result:

The program was executed successfully. Variables of different data types were created, values were assigned, operations were performed, and their data types were verified. This confirms that Python can effectively handle variables in business-related computations.

Exercise No: 6

Date:

ARITHMETIC OPERATORS IN PYTHON

Aim:

To study and demonstrate the use of arithmetic operators in Python.

Procedure:

1. Open Python (IDLE, Jupyter Notebook, or any IDE).
2. Declare numeric variables and assign values.
3. Apply arithmetic operators (+, -, *, /, //, %, **) on the variables.
4. Use the print() function to display results for each operator.
5. Execute the program and verify the output.

Program (Code):

```
# Exercise: Arithmetic Operators in Python

# Declaring variables
a = 15
b = 4

# Applying arithmetic operators
print("a =", a, " b =", b)
print("Addition (a + b):", a + b)
print("Subtraction (a - b):", a - b)
print("Multiplication (a * b):", a * b)
print("Division (a / b):", a / b)
print("Floor Division (a // b):", a // b)
print("Modulus (a % b):", a % b)
print("Exponentiation (a ** b):", a ** b)
```

Output:

a = 15 b = 4

Addition (a + b): 19

Subtraction (a - b): 11

Multiplication (a * b): 60

Division (a / b): 3.75

Floor Division (a // b): 3

Modulus (a % b): 3

Exponentiation (a ** b): 50625

Result:

The program was executed successfully. Different arithmetic operators in Python were applied on numeric variables, and their results were displayed.

Exercise No: 7	**PROGRAM TO FIND AREA OF TRIANGLE**
Date:	

AIM

To write a python program to find the area of a triangle.

PROCEDURE

Step 1: Get the values of Breadth and Height from the user.

Step 2: Use the formula ½*b*h to find the area of triangle.

Step 3: Print the results.

SOURCE CODE

```
b=int(input('Enter breadth of a triangle: '))
h=int(input("Enter height of a triangle: "))
area=(b*h)/2
print('The area of triangle is',area)
```

OUTPUT

```
Enter breadth of a triangle: 4
Enter height of a triangle: 6
The area of triangle is 12.0
```

RESULT

Thus the area of a triangle has been found out successfully.

Exercise No: 8

Date:

PROGRAM TO FIND SQUARE ROOT

AIM

To write a python program to find the square root.

PROCEDURE

Step 1: Get an integer number.

Step 2: Find the square root of the number.

Step 3: Print the results.

SOURCE CODE

```
# To take the input from the user
num = float(input('Enter a number: '))
num_sqrt = num ** 0.5
print('The square root of %0.3f is %0.3f'%(num ,num_sqrt))
```

OUTPUT

Enter a number: 690

The square root of 690.000 is 26.268

RESULT

Thus the square root for the given number has been performed successfully.

Exercise No: 9

Date:

LOGICAL OPERATORS IN PYTHON

Aim:

To study and demonstrate the use of logical operators (and, or, not) in Python.

Procedure:

1. Open Python (IDLE, Jupyter Notebook, or any IDE).
2. Declare boolean variables with values True or False.
3. Apply logical operators and, or, and not on the variables.
4. Display the results using the print() function.
5. Execute the program and verify the output.

Program (Code):

```
# Exercise: Logical Operators in Python

# Declaring boolean variables
x = True
y = False

# Applying logical operators
print("x =", x, " y =", y)
print("x and y:", x and y)   # True only if both are True
print("x or y:", x or y)     # True if at least one is True
print("not x:", not x)       # Negates the value of x
print("not y:", not y)       # Negates the value of y
```

Output:

```
x = True  y = False
x and y: False
x or y: True
```

not x: False

not y: True

Result:

The program was executed successfully. Logical operators and, or, and not were demonstrated with boolean values, and their truth table behavior was verified.

Exercise No: 10

Date:

SWAPPING OF TWO NUMBERS USING THIRD VARIABLE

AIM

To write a python program to swap two numbers using third variable.

PROCEDURE

Step 1: Get the values of x and y.

Step 2: Swap the values of two variables using a third variable called temp**.**

Step 3: Print the results.

SOURCE CODE

```
x = 10
y = 50

# Swapping of two variables
# Using third variable
temp = x
x = y
y = temp

print("Value of x:", x)
print("Value of y:", y)
```

OUTPUT

```
Value of x: 50
Value of y: 10
```

RESULT

Thus the two variable has been swapped using a third variable successfully.

Exercise No: 11	PYTHON PROGRAM TO FIND THE LARGEST AMONG THREE NUMBERS
Date:	

AIM

To write a python program to find the largest among three numbers.

PROCEDURE

1. Get the value for num1,num2 and num3
2. Use the if elif and else statement to find largest number
3. Print the result

SOURCE CODE

```
# Python program to find the largest number among the three input numbers
# change the values of num1, num2 and num3
# for a different result
num1 = 10
num2 = 14
num3 = 12

if (num1 >= num2) and (num1 >= num3):
   largest = num1
elif (num2 >= num1) and (num2 >= num3):
   largest = num2
else:
   largest = num3

print("The largest number is", largest)
```

OUTPUT

The largest number is 14.0.

RESULT

Thus, the program is executed successfully

Exercise No: 12

Date:

CONTROL STRUCTURES IN PYTHON

Aim:

To study and implement control structures in Python, including **conditional statements** (if, if-else, if-elif-else) and **loops** (for, while).

Procedure:

1. Open Python (IDLE, Jupyter Notebook, or any IDE).
2. Write a program to demonstrate:
   - **Decision-making** using if, if-else, and if-elif-else.
   - **Iteration** using for loop and while loop.
3. Print appropriate outputs for each case.
4. Execute the program and verify the results.

Program (Code):

```
# Exercise: Control Structures in Python


# 1. Conditional Statements
num = 10


# if statement
if num > 0:
    print("Number is positive")


# if-else statement
if num % 2 == 0:
    print("Number is even")
else:
    print("Number is odd")
```

```
# if-elif-else statement
if num < 0:
    print("Number is negative")
elif num == 0:
    print("Number is zero")
else:
    print("Number is positive")

print("-------------------------")

# 2. Looping Statements

# for loop
print("For loop: Printing numbers from 1 to 5")
for i in range(1, 6):
    print(i)

print("-------------------------")

# while loop
print("While loop: Printing numbers from 1 to 5")
i = 1
while i <= 5:
    print(i)
    i += 1
```

Output:

Number is positive

Number is even

Number is positive

For loop: Printing numbers from 1 to 5

1

2

3

4

5

While loop: Printing numbers from 1 to 5

1

2

3

4

5

Result:

The program was executed successfully. Different control structures in Python (if, if-else, if-elif-else, for loop, and while loop) were demonstrated with correct outputs.

Exercise No: 13	**PYTHON PROGRAM DISPLAY THE MULTIPLICATION TABLE**
Date:	

AIM

To write a python program to display the multiplication table.

PROCEDURE

1. Get the value as 12
2. Use the loop to display the multiplication table
3. Print the result

SOURCE CODE

```
# Multiplication table (from 1 to 10) in Python
num = 12
# To take input from the user
# num = int(input("Display multiplication table of? "))
# Iterate 10 times from i = 1 to 10
for i in range(1, 11):
   print(num, 'x', i, '=', num*i)
```

OUTPUT

12 x 1 = 12, 12 x 2 = 24, 12 x 3 = 36, 12 x 4 = 48, 12 x 5 = 60, 12 x 6 = 72, 12 x 7 = 84, 12 x 8 = 96, 12 x 9 = 108, 12 x 10 = 120

RESULT

Thus, the program is executed successfully.

Exercise No: 14

Date:

FUNCTIONS IN PYTHON

Aim:
To understand the concept of functions in Python and learn how to define and call them for code reusability.

Procedure:

1. Open Anaconda Navigator and launch Jupyter Notebook (or any Python IDE).
2. Create a new Python file or notebook.
3. Define a function using the def keyword.
   - Example:
   - def greet(name):
   -     print("Hello,", name, "Welcome to Python Programming!")
4. Call the function with different arguments (values).
5. greet("Alice")
6. greet("Bob")
7. Execute the program and observe the output.

Code Example:

```
# Defining a function
def greet(name):
    """This function greets the person by name"""
    print("Hello,", name, "Welcome to Python Programming!")

# Calling the function with arguments
greet("Alice")
greet("Bob")
```

Output:

Hello, Alice Welcome to Python Programming!

Hello, Bob Welcome to Python Programming!

Result:
The concept of functions in Python was successfully implemented.
We learned how to define a function, pass parameters, and call the function to achieve reusable and modular code.

Exercise No: 15	**PYTHON PROGRAM TO MAKE A SIMPLE CALCULATOR**
Date:	

AIM

To write a python program to make a simple calculator.

PROCEDURE

1. Get instructions in string
2. Use the function arguments and user defined function to make a simple calculator
3. Print the result

SOURCE CODE

```
# This function adds two numbers
def add(x, y):
   return x + y
# This function subtracts two numbers
def subtract(x, y):
   return x - y

# This function multiplies two numbers
def multiply(x, y):
   return x * y

# This function divides two numbers
def divide(x, y):
   return x / y

print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")
```

```
while True:
  # take input from the user
  choice = input("Enter choice(1/2/3/4): ")
  # check if choice is one of the four options
  if choice in ('1', '2', '3', '4'):
    try:
      num1 = float(input("Enter first number: "))
      num2 = float(input("Enter second number: "))
    except ValueError:
      print("Invalid input. Please enter a number.")
      continue
    if choice == '1':
      print(num1, "+", num2, "=", add(num1, num2))

    elif choice == '2':
      print(num1, "-", num2, "=", subtract(num1, num2))

    elif choice == '3':
      print(num1, "*", num2, "=", multiply(num1, num2))

    elif choice == '4':
      print(num1, "/", num2, "=", divide(num1, num2))
    # check if user wants another calculation
    # break the while loop if answer is no
    next_calculation = input("Let's do next calculation? (yes/no): ")
    if next_calculation == "no":
     break
  else:
    print("Invalid Input")
```

OUTPUT

Select operation.

1.Add

2.Subtract

3.Multiply

4.Divide

Enter choice(1/2/3/4): 3

Enter first number: 15

Enter second number: 14

15.0 * 14.0 = 210.0

Let's do next calculation? (yes/no): no

RESULT

Thus, the program is executed successfully.

Exercise No: 16

Date:

STRING OPERATIONS IN PYTHON

Aim:
To study and implement various string operations in Python such as concatenation, repetition, slicing, and built-in string functions.

Procedure:

1. Open Python in any IDE (e.g., Jupyter Notebook, IDLE, or VS Code).
2. Create a Python file or notebook.
3. Declare string variables and perform the following operations:
   - Concatenation (+)
   - Repetition (*)
   - Indexing and slicing ([])
   - Common functions (len(), upper(), lower(), strip(), replace(), split(), find(), etc.)
4. Execute the program and observe the output.

Code Example:

```
# Declaring string variables
str1 = "Hello"
str2 = "World"

# Concatenation
result1 = str1 + " " + str2

# Repetition
result2 = str1 * 3

# Slicing
result3 = str2[0:3]
```

```
# String functions
length = len(str1)
upper_case = str1.upper()
lower_case = str2.lower()
replaced = str2.replace("World", "Python")
split_str = "Python Programming".split()

# Displaying results
print("Concatenation:", result1)
print("Repetition:", result2)
print("Slicing:", result3)
print("Length of str1:", length)
print("Upper case:", upper_case)
print("Lower case:", lower_case)
print("Replaced string:", replaced)
print("Split string:", split_str)
```

Output:

```
Concatenation: Hello World
Repetition: HelloHelloHello
Slicing: Wor
Length of str1: 5
Upper case: HELLO
Lower case: world
Replaced string: Python
Split string: ['Python', 'Programming']
```

Result:

Various string operations in Python such as concatenation, repetition, slicing, and built-in string functions were successfully implemented.

Exercise No: 17	PYTHON PROGRAM TO CHECK WHETHER A STRING IS PALINDROME OR NOT
Date:	

AIM

To write a python program to find the sum of natural numbers

PROCEDURE

1. Get the instruction in string
2. Use the string method to check a string is palindrome or not
3. Print the result

SOURCE CODE

```
# Program to check if a string is palindrome or not
my_str = 'aIbohPhoBiA'
# make it suitable for caseless comparison
my_str = my_str.casefold()
# reverse the string
rev_str = reversed(my_str)
# check if the string is equal to its reverse
if list(my_str) == list(rev_str):
  print("The string is a palindrome.")
else:
  print("The string is not a palindrome.")
```

OUTPUT

The string is a palindrome.

RESULT

Thus, the program is executed successfully.

Exercise No: 18	PYTHON PROGRAM TO COMPUTE THE POWER OF A NUMBER
Date:	

AIM

To write a python program to compute a power of a number.

PROCEDURE

1. Get the value for base and exponent
2. Use the while loop to compute the power of a number
3. Print the result

SOURCE CODE

```
base = 3
exponent = 4
result = 1
while exponent != 0:
    result *= base
    exponent-=1
print("Answer = " + str(result))
```

OUTPUT

81

RESULT

Thus the program is executed successfully.

Exercise No: 19	PYTHON PROGRAM TO COUNT THE NUMBER OF DIGITS PRESENT IN A NUMBER
Date:	

AIM

To write a python program to compute a power of a number.

PROCEDURE

1. Get the value of 3452
2. Use the while loop to count digits present In a number
3. Print the result

SOURCE CODE

```
num = 3452
count = 0

while num != 0:
   num //= 10
   count += 1

print("Number of digits: " + str(count))
```

OUTPUT

Number of digits: 4

RESULT

Thus the program is executed successfully.

Exercise No: 20

Date:

INSTALLING & USING PACKAGES IN PYTHON

Aim

To learn how to install and use external packages in Python for performing specific tasks.

Procedure

1. Open the command prompt (or terminal).
2. Use pip to install a package. Example:
3. pip install numpy
4. Import the installed package in Python using the import statement.
5. Use functions from the package in your program.

Program

```
# Step 1: Install numpy (done in terminal)
pip install numpy
# Step 2: Import and use numpy
import numpy as np
# Create an array
arr = np.array([1, 2, 3, 4, 5])
print("Array:", arr)
print("Mean of Array:", np.mean(arr))
```

Output

```
Array: [1 2 3 4 5]
Mean of Array: 3.0
```

Result

We successfully installed the NumPy package using pip and used its functions in Python to perform mathematical operations.

Exercise No: 21

Date:

FILE HANDLING IN PYTHON

Aim:

To study and implement basic file handling operations such as creating, writing, reading, and appending data in a file using Python.

Procedure:

1. Start Python (IDLE / Jupyter Notebook / VS Code).
2. Create or open a file using the built-in open() function with different modes:
   - "w" → Write mode (creates a new file or overwrites an existing one).
   - "r" → Read mode (reads the content of a file).
   - "a" → Append mode (adds new data without deleting existing content).
3. Use file object methods like:
   - write() → To write content into the file.
   - read() / readline() / readlines() → To read content from the file.
   - close() → To close the file after operations.
4. Save the program and run it.
5. Verify the output by checking the file contents in your system.

Program & Output:

```
# File Handling Example


# Step 1: Create and write to a file
file = open("example.txt", "w")
file.write("Hello, this is the first line.\n")
file.write("Python File Handling Example.\n")
file.close()


# Step 2: Append new data to the file
file = open("example.txt", "a")
```

```python
file.write("This line is appended to the file.\n")
file.close()

# Step 3: Read data from the file
file = open("example.txt", "r")
content = file.read()
print("File Content:\n", content)
file.close()
```

Sample Output (on screen):

```
File Content:
 Hello, this is the first line.
Python File Handling Example.
This line is appended to the file.
```

Result:

File handling in Python was successfully implemented. A text file was created, written with content, appended with new data, and read back using Python.

Exercise No: 22

Date:

LIST IN PYTHON

Aim

To understand how to create, access, modify, and perform operations on **list in Python**.

Procedure

1. Open **Python IDE** (Jupyter Notebook, PyCharm, VS Code, or IDLE).
2. Create a new Python file or notebook.
3. Define a list using square brackets [].
4. Perform different list operations:
   - Access elements using index.
   - Modify elements by assigning new values.
   - Use built-in functions like append(), insert(), remove(), pop(), sort(), reverse(), etc.
5. Print the results to observe how lists work.
6. Run the program.

Program (Example Code)

```
# Creating and working with lists in Python

# Creating a list
fruits = ["apple", "banana", "cherry"]

# Displaying the list
print("Original List:", fruits)

# Accessing elements
print("First fruit:", fruits[0])
print("Last fruit:", fruits[-1])

# Modifying elements
```

```
fruits[1] = "mango"
print("After modification:", fruits)

# Adding elements
fruits.append("orange")
print("After appending:", fruits)

# Inserting at specific position
fruits.insert(1, "grapes")
print("After inserting grapes:", fruits)

# Removing elements
fruits.remove("cherry")
print("After removing cherry:", fruits)

# Popping last element
fruits.pop()
print("After popping last element:", fruits)

# Sorting the list
fruits.sort()
print("Sorted List:", fruits)

# Reversing the list
fruits.reverse()
print("Reversed List:", fruits)
```

Output

Original List: ['apple', 'banana', 'cherry']

First fruit: apple

Last fruit: cherry

After modification: ['apple', 'mango', 'cherry']

After appending: ['apple', 'mango', 'cherry', 'orange']

After inserting grapes: ['apple', 'grapes', 'mango', 'cherry', 'orange']

After removing cherry: ['apple', 'grapes', 'mango', 'orange']

After popping last element: ['apple', 'grapes', 'mango']

Sorted List: ['apple', 'grapes', 'mango']

Reversed List: ['mango', 'grapes', 'apple']

Result

Thus, we have successfully created and performed different operations on **lists in Python**, including creation, modification, insertion, deletion, sorting, and reversing.

Exercise No: 23

Date:

TUPLE IN PYTHON

Aim

To understand and implement tuples in Python, and learn their characteristics such as immutability, indexing, and usage.

Procedure

1. Open **Anaconda Navigator** or your preferred Python IDE.
2. Create a new Python file named tuple_example.py.
3. Write the following Python code to demonstrate tuple creation, accessing elements, and immutability:

```
# Creating a tuple
my_tuple = (10, 20, 30, 40, 50)

# Display the tuple
print("Tuple Elements:", my_tuple)

# Accessing elements using index
print("First Element:", my_tuple[0])
print("Last Element:", my_tuple[-1])

# Tuple slicing
print("Slice from 1 to 3:", my_tuple[1:4])

# Demonstrating immutability
try:
    my_tuple[0] = 100
except TypeError as e:
    print("Error:", e)
```

```
# Tuple with mixed data types
mixed_tuple = (1, "Python", 3.14, True)
print("Mixed Tuple:", mixed_tuple)

# Nested tuple
nested_tuple = (1, 2, (3, 4, 5))
print("Nested Tuple:", nested_tuple)
```

4. Save and run the file.
5. Observe the outputs to understand tuple features.

Output

Tuple Elements: (10, 20, 30, 40, 50)

First Element: 10

Last Element: 50

Slice from 1 to 3: (20, 30, 40)

Error: 'tuple' object does not support item assignment

Mixed Tuple: (1, 'Python', 3.14, True)

Nested Tuple: (1, 2, (3, 4, 5))

Result

Tuples in Python are immutable ordered collections that can hold elements of different data types. They support indexing, slicing, and nesting but do not allow modification of elements after creation.

Exercise No: 24

Date:

DICTIONARY IN PYTHON

Aim

To study and implement dictionaries in Python, and understand their characteristics such as key–value pairs, mutability, and common operations.

Procedure

1. Open your Python IDE (IDLE, Jupyter Notebook, PyCharm, or VS Code).
2. Create a new Python file named dictionary_example.py.
3. Write the following Python code to demonstrate dictionary creation, accessing, updating, deleting, and iterating through key–value pairs.

```
# Creating a dictionary
student = {
    "name": "Rahul",
    "age": 21,
    "course": "BBA",
    "marks": 88
}

# Display the dictionary
print("Student Dictionary:", student)

# Accessing values using keys
print("Name:", student["name"])
print("Age:", student.get("age"))

# Adding a new key-value pair
student["semester"] = 5
print("After Adding Semester:", student)
```

```
# Updating a value
student["marks"] = 92
print("After Updating Marks:", student)

# Deleting a key-value pair
del student["course"]
print("After Deleting Course:", student)

# Iterating through dictionary
print("\nIterating through dictionary:")
for key, value in student.items():
    print(key, ":", value)

# Checking if a key exists
print("\nIs 'age' present?", "age" in student)
print("Is 'course' present?", "course" in student)
```

4. Save and run the file.
5. Observe the output to analyze dictionary operations.

Output

```
Student Dictionary: {'name': 'Rahul', 'age': 21, 'course': 'BBA', 'marks': 88}
Name: Rahul
Age: 21
After Adding Semester: {'name': 'Rahul', 'age': 21, 'course': 'BBA', 'marks': 88, 'semester': 5}
After Updating Marks: {'name': 'Rahul', 'age': 21, 'course': 'BBA', 'marks': 92, 'semester': 5}
After Deleting Course: {'name': 'Rahul', 'age': 21, 'marks': 92, 'semester': 5}

Iterating through dictionary:
name : Rahul
age : 21
```

marks : 92

semester : 5

Is 'age' present? True

Is 'course' present? False

Result

Dictionaries in Python are **mutable, unordered collections of key–value pairs**. They allow efficient data retrieval using keys, support dynamic updates, and provide useful methods for adding, updating, deleting, and iterating over elements.

Exercise No: 25

Date:

SETS IN PYTHON

Aim:

To understand the concept of **sets in Python**, and perform basic set operations such as creation, adding elements, removing elements, and performing union, intersection, and difference.

Procedure:

1. Open a Python environment (IDLE, Jupyter Notebook, or any IDE).
2. Create a set using curly braces {} or the set() function.
3. Add elements to the set using the add() method.
4. Remove elements using remove() or discard().
5. Perform common set operations such as **union (| or union()), intersection (& or intersection()), and difference (- or difference())**.
6. Print the results to verify the operations.

Program / Code:

```
# Creating a set
fruits = {"apple", "banana", "cherry"}
print("Initial Set:", fruits)

# Adding an element
fruits.add("orange")
print("After Adding Orange:", fruits)

# Removing an element
fruits.remove("banana")
print("After Removing Banana:", fruits)

# Set Operations
set1 = {1, 2, 3, 4}
```

```
set2 = {3, 4, 5, 6}

print("Union:", set1 | set2)
print("Intersection:", set1 & set2)
print("Difference:", set1 - set2)
```

Output:

Initial Set: {'apple', 'banana', 'cherry'}

After Adding Orange: {'apple', 'banana', 'cherry', 'orange'}

After Removing Banana: {'apple', 'cherry', 'orange'}

Union: {1, 2, 3, 4, 5, 6}

Intersection: {3, 4}

Difference: {1, 2}

Result:

The concept of **sets in Python** was successfully implemented. We learned how to create sets, add/remove elements, and perform set operations like union, intersection, and difference.

Exercise No: 26

Date:

DATA SLICING IN PYTHON

Aim

To understand and demonstrate **data slicing in Python** using the **pandas DataFrame**, which allows us to extract specific rows, columns, or subsets of data for analysis.

Procedure

1. Import the **pandas** library.
2. Create a **DataFrame** with sample data (e.g., employee details).
3. Perform different slicing operations:
    - Select a **single column**.
    - Select **multiple columns**.
    - Slice **rows by index**.
    - Use **loc** for label-based slicing.
    - Use **iloc** for integer position-based slicing.
4. Display the sliced outputs to understand how slicing works.

Code

```
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [24, 27, 22, 32, 29],
    'Department': ['HR', 'IT', 'Finance', 'Marketing', 'IT'],
    'Salary': [40000, 50000, 45000, 60000, 52000]
}

df = pd.DataFrame(data)
```

```
# 1. Selecting a single column
print("Single Column (Name):\n", df['Name'], "\n")

# 2. Selecting multiple columns
print("Multiple Columns (Name & Salary):\n", df[['Name', 'Salary']], "\n")

# 3. Selecting rows by index (slicing rows)
print("Row Slicing (rows 1 to 3):\n", df[1:4], "\n")

# 4. Using loc for label-based slicing
print("loc Slicing (rows 1 to 3, Name & Department):\n", df.loc[1:3, ['Name', 'Department']], "\n")

# 5. Using iloc for integer-based slicing
print("iloc Slicing (rows 0 to 2, columns 1 to 2):\n", df.iloc[0:3, 1:3], "\n")
```

Output

1. Single Column (Name):

```
0     Alice
1       Bob
2   Charlie
3     David
4       Eva
Name: Name, dtype: object
```

2. Multiple Columns (Name & Salary):

```
      Name  Salary
0    Alice   40000
1      Bob   50000
2  Charlie   45000
3    David   60000
4      Eva   52000
```

3. Row Slicing (rows 1 to 3):

```
      Name  Age Department  Salary
1      Bob   27         IT   50000
2  Charlie   22    Finance   45000
3    David   32  Marketing   60000
```

4. loc Slicing (rows 1 to 3, Name & Department):

```
      Name Department
1      Bob         IT
2  Charlie    Finance
3    David  Marketing
```

5. iloc Slicing (rows 0 to 2, columns 1 to 2):

```
   Age Department
0   24         HR
1   27         IT
2   22    Finance
```

Result

We successfully performed **data slicing operations** in Python using pandas.

- Extracted single and multiple columns.
- Sliced rows using index ranges.
- Used **loc** (label-based) and **iloc** (integer-based) indexing to select specific rows and columns.

Exercise No: 27

Date:

IMPORTING CSV DATA IN PANDAS

Aim

To learn how to import data from a CSV file using **Pandas** in Python.

Procedure

1. Install Pandas (if not already installed):
2. pip install pandas
3. Import the pandas library in Python.
4. Use the read_csv() function to load data from a CSV file.
5. Display the data using print() or head().

Program

```
import pandas as pd

# Import CSV file
data = pd.read_csv("D:/students.csv")

# Display first 5 rows
print(data.head())
```

Output

```
   Name  Age  Marks
0  John   20    85
1  Mary   21    90
2  Alex   19    78
3  Sara   22    88
4  Tom    20    92
```

Result

We successfully imported data from a CSV file into a **Pandas DataFrame** and displayed it.

Exercise No: 28

Date:

IMPORTING EXCEL DATA IN PANDAS

Aim

To learn how to import data from an **Excel file** using **Pandas** in Python.

Procedure

1. Install Pandas and openpyxl (Excel engine):

   pip install pandas openpyxl

2. Import the pandas library.
3. Use the read_excel() function to load data from an Excel file.
4. Display the data using print() or head().

Program

```
import pandas as pd

# Import Excel file
data = pd.read_excel("students.xlsx")

# Display first 5 rows
print(data.head())
```

Output

```
   Name  Age  Marks
0  John   20    85
1  Mary   21    90
2  Alex   19    78
3  Sara   22    88
4  Tom    20    92
```

Result

We successfully imported data from an **Excel file** into a **Pandas DataFrame** and displayed it.

Exercise No: 29	INTRODUCTION TO R PROGRAMMING
Date:	

Introduction to R:

- ✓ R is a programming language and software environment for statistical analysis, graphics representation and reporting.
- ✓ R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.
- ✓ This programming language was named **R**, based on the first letter of first name of the two R authors (Robert Gentleman and Ross Ihaka).
- ✓ R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

Features of R:

- R is a well-developed, simple and effective programming language which includes conditionals, loops; user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

Data types:

The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are −

- Vectors
- Lists
- Matrices
- Arrays
- Factors
- Data Frames

Variables:

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R objects.

A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validi t y	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name , var.name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

Types of Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

We have the following types of operators in R programming −

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

Decision Making:

R provides the following types of decision-making statements. Click the following links to check their detail.

S.No	Statement & Description
1	**if statement** - An **if** statement consists of a Boolean expression followed by one or more statements.
2	**if...else statement** - An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.
3	**switch statement** - A **switch** statement allows a variable to be tested for equality against a list of values.

Loop:

R programming language provides the following kinds of loop to handle looping requirements. Click the following links to check their detail.

S.No.	Loop Type & Description
1	**Repeat loop** - Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
2	**While loop** - Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
3	**for loop** - Like a while statement, except that it tests the condition at the end of the loop body.

Function:

Function Definition:

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows –

```
function_name <- function(arg_1, arg_2, ...) {
  Function body
}
```

Function Components

The different parts of a function are −

- **Function Name** − This is the actual name of the function. It is stored in R environment as an object with this name.
- **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.
- **Function Body** − The function body contains a collection of statements that defines what the function does.
- **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.

String:

Any value written within a pair of single quote or double quotes in R is treated as a string. Internally R stores every string within double quotes, even when you create them with single quote.

Rules Applied in String Construction

- The quotes at the beginning and end of a string should be either double quotes or both single quotes. They cannot be mixed.
- Double quotes can be inserted into a string starting and ending with single quote.
- Single quote can be inserted into a string starting and ending with double quotes.
- Double quotes cannot be inserted into a string starting and ending with double quotes.
- Single quote cannot be inserted into a string starting and ending with single quote.

Vector:

Vectors are the most basic R data objects and there are six types of atomic vectors. They are logical, integer, double, complex, character and raw.

List:

Lists are the R objects which contain elements of different types like − numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using **list()** function.

Matrix:

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations. A Matrix is created using the **matrix ()** function.

Syntax

The basic syntax for creating a matrix in R is −

```
matrix(data, nrow, ncol, byrow, dimnames)
```

Following is the description of the parameters used −

- **data** is the input vector which becomes the data elements of the matrix.
- **nrow** is the number of rows to be created.
- **ncol** is the number of columns to be created.
- **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
- **dimname** is the names assigned to the rows and columns.

Array:

Arrays are the R data objects which can store data in more than two dimensions. For example − If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. Arrays can store only data type.

An array is created using the **array()** function. It takes vectors as input and uses the values in the **dim** parameter to create an array.

Factor:

Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in the columns which have a limited number of unique values. Like "Male, "Female" and True, False etc. They are useful in data analysis for statistical modeling.

Factors are created using the **factor ()** function by taking a vector as input.

Data Frames:

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

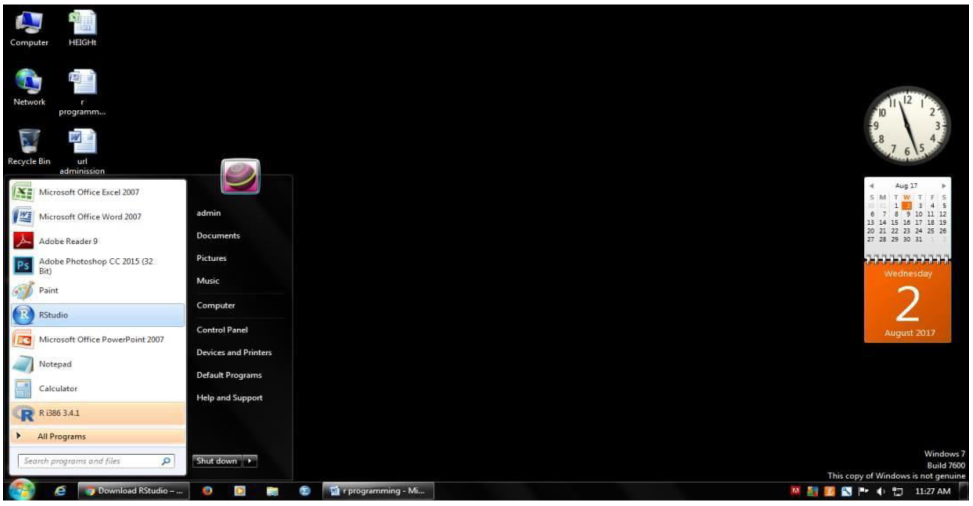
R Studio

RStudio is a free and open-source integrated development environment (IDE) for R, a programming language for statistical computing and graphics.

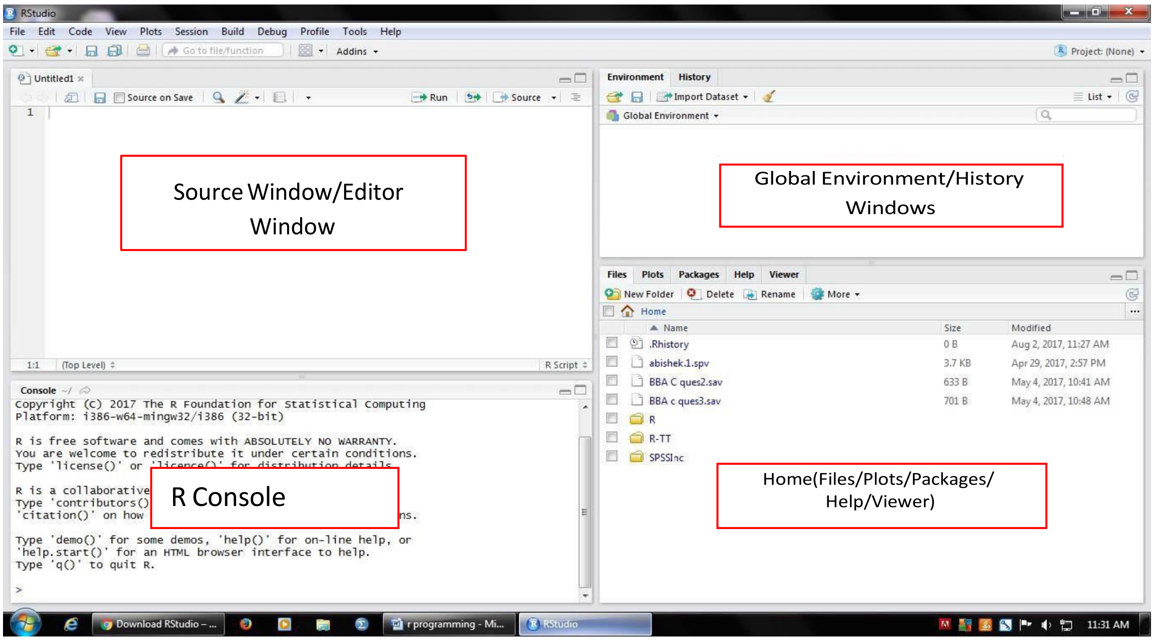
How to open R Studio:

Open in R programming for the following steps

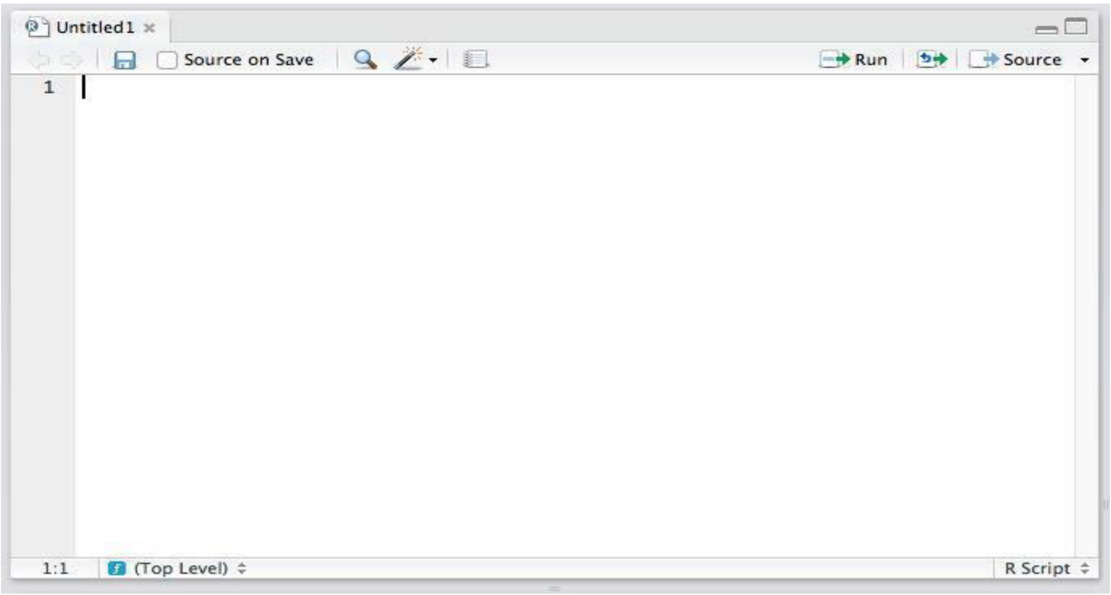
1. First go to start menu & open R Studio



2. In the next step to be display four windows.
   - Source Window /Editor Window
   - Console window
   - Global Environment/History
   - Home(files, plot, packages, help, viewer)



Editor window:



The top left quadrant is the editor. It's where you write R code you want to keep for later – functions, classes, packages, etc. This is, for all intents and purposes, identical to every other code editor's main window.

Apart from some self-explanatory buttons, and others that needn't concern you at this starting point, there is also a "Source on Save" checkbox.

This means "Load contents of file into my console's runtime every time I save the file". You should have this on at all times; it makes your development flow faster by one click.

Console Window:

1. The lower left quadrant is the console. It's a REPL for R in which you can test out your ideas, datasets, filters, and functions.
2. This is where you'll be spending most of your time in the beginning – it's here that you verify an idea you had works before copying it over into the editor above.
3. This is also the environment into which your R files will be sourced on save (see above), so whenever you develop a new function in an R file above, it automatically becomes available in this REPL. We'll be spending a lot of time in the REPL in the remainder of this article.

Global Environment / History:

The top right quadrant has two tabs: environment and history.

Environment refers to the console environment (see above) and will list, in detail, every single symbol you defined in the console (whether via sourcing or directly). That is, if you have a function available in the REPL, it will be listed in the environment. If you have a variable, or a dataset, it will be listed there.

This is where you can also import custom datasets manually and make them instantly available in the console, if you don't feel like typing out the commands to do so. You can also inspect the environment of other packages you installed and loaded (more on packages at a later time). Play around with it – you can't break anything.

History lists every single console command you executed since the last project started. It is saved into a hidden .Rhistory file in your project's folder. If you don't choose to save your environment after a session, the history won't be saved.

Home (files, plot, packages, help, viewer)

The bottom right panel is the misc panel, and contains five separate tabs. The first one, **Files**, is self-explanatory.

The **Plots** tab will contain the graphs you generated with R. It is there you can zoom, export, configure and inspect your charts and plots.

The **Packages** tab lets you install additional packages into R.

The **Help** tab lets you search the incredibly extensive help directory and will automatically open whenever you call help on a command in the console (help is called by pretending a command name with a question mark, like so: ?data. frame).

Packages:

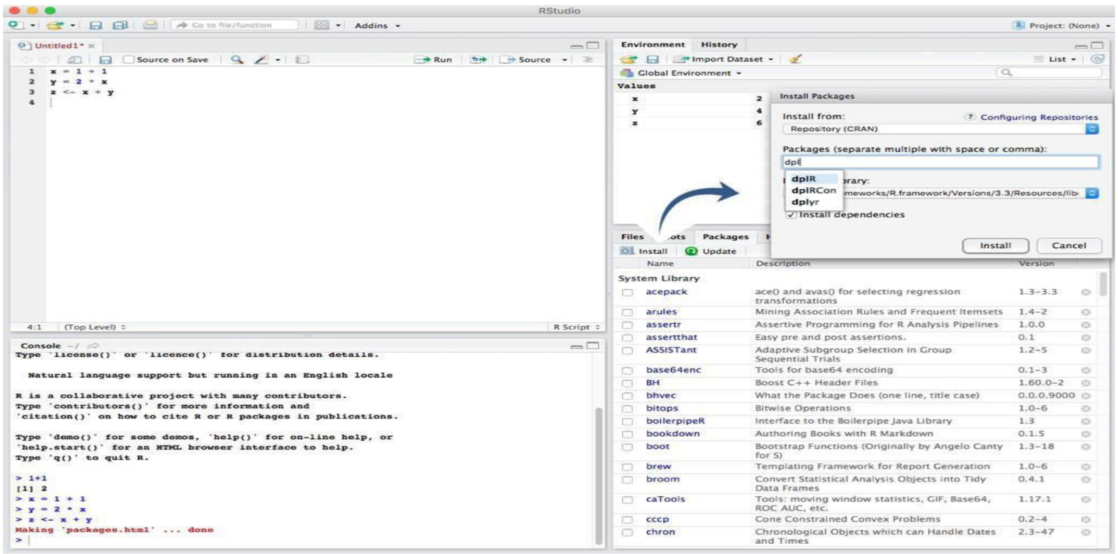
R packages are a collection of R functions, complied code and sample data. They are stored under a directory called **"library"** in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

All the packages available in R language are listed at R Packages.

Below is a list of commands to be used to check, verify and use the R packages.

Installing Packages

There are world-wide R package repositories or Comprehensive R Archive Network (CRAN) sites that allow packages to be downloaded and installed. You almost never have to directly work with them since RStudio makes it easy to install the packages as shown in the figure below, where we have clicked on the Packages tab and clicked on the Install button. Note how as you type the name of a package, you get auto-completion. (In fact, RStudio provides auto-completion even as you type R commands, showing you various options you can use for the commands)



Recommended Packages

Many useful R function come in packages, free libraries of code written by R's active user community. To install an R package, open an R session and type at the command line

install.packages("<the package's name>")

R will download the package from CRAN, so you'll need to be connected to the internet. Once you have a package installed, you can make its contents available to use in your current R session by running

library("<the package's name>")

There are thousands of helpful R packages for you to use, but navigating them all can be a challenge. To help you out, we've compiled this guide to some of the best. We've used each of these, and found them to be outstanding – we've even written some of them. But you don't have to take our word for it, these packages are also some of the top most downloaded R packages.

To load data

DBI - The standard for for communication between R and relational database management systems. Packages that connect R to databases depend on the DBI package.

odbc - Use any ODBC driver with the odbc package to connect R to your database. Note: RStudio professional products come with professional drivers for some of the most popular databases.

RMySQL, RPostgresSQL, RSQLite - If you'd like to read in data from a database, these packages are a good place to start. Choose the package that fits your type of database.

XLConnect, xlsx - These packages help you read and write Micorsoft Excel files from R. You can also just export your spreadsheets from Excel as. csv's.

foreign - Want to read a SAS data set into R? Or an SPSS data set? Foreign provides functions that help you load data files from other programs into R.

haven - Enables R to read and write data from SAS, SPSS, and Stata.

R can handle plain text files – no package required. Just use the functions read.csv, read.table, and read.fwf. If you have even more exotic data, consult the CRAN guide to data import and export.

To manipulate data

dplyr - Essential shortcuts for subsetting, summarizing, rearranging, and joining together data sets. dplyr is our go to package for fast data manipulation.

tidyr - Tools for changing the layout of your data sets. Use the gather and spread functions to convert your data into the tidy format, the layout R likes best.

stringr - Easy to learn tools for regular expressions and character strings.

lubridate - Tools that make working with dates and times easier.

To visualize data

ggplot2 - R's famous package for making beautiful graphics. ggplot2 lets you use the grammar of graphics to build layered, customizable plots.

ggvis - Interactive, web based graphics built with the grammar of graphics.

rgl - Interactive 3D visualizations with R

htmlwidgets - A fast way to build interactive (javascript based) visualizations with R. Packages that implement htmlwidgets include:

leaflet (maps)

dygraphs (time series)

DT (tables)

diagrammeR (diagrams)

network3D (network graphs)

threeJS (3D scatterplots and globes).

googleVis - Let's you use Google Chart tools to visualize data in R. Google Chart tools used to be called Gapminder, the graphing software Hans Rosling made famous in hie TED talk.

To model data

car - car's Anova function is popular for making type II and type III Anova tables.

mgcv - Generalized Additive Models

lme4/nlme - Linear and Non-linear mixed effects models
randomForest - Random forest methods from machine learning

multcomp - Tools for multiple comparison testing

vcd - Visualization tools and tests for categorical data

glmnet - Lasso and elastic-net regression methods with cross validation

survival - Tools for survival analysis

caret - Tools for training regression and classification models

To report results

shiny - Easily make interactive, web apps with R. A perfect way to explore data and share findings with non-programmers.

R Markdown - The perfect workflow for reproducible reporting. Write R code in your markdown reports. When you run render, R Markdown will replace the code with its results and then export your report as an HTML, pdf, or MS Word document, or a HTML or pdf slideshow. The result? Automated reporting. R Markdown is integrated straight into RStudio.

xtable - The xtable function takes an R object (like a data frame) and returns the latex or HTML code you need to paste a pretty version of the object into your documents. Copy and paste, or pair up with R Markdown.

For Spatial data

sp, maptools - Tools for loading and using spatial data including shapefiles.

maps - Easy to use map polygons for plots.

ggmap - Download street maps straight from Google maps and use them as a background in your ggplots.

For Time Series and Financial data

zoo - Provides the most popular format for saving time series objects in R.

xts - Very flexible tools for manipulating time series data sets.

quantmod - Tools for downloading financial data, plotting common charts, and doing technical analysis.

To write high performance R code

Rcpp - Write R functions that call C++ code for lightning fast speed.

data.table - An alternative way to organize data sets for very, very fast operations. Useful for big data.

parallel - Use parallel processing in R to speed up your code or to crunch large data sets.

To work with the web

XML - Read and create XML documents with R

jsonlite - Read and create JSON data tables with R

httr - A set of useful tools for working with http connections

To write your own R packages

devtools - An essential suite of tools for turning your code into an R package.

testthat - testthat provides an easy way to write unit tests for your code projects.

roxygen2 - A quick way to document your R packages. roxygen2 turns inline code comments into documentation pages and builds a package namespace.

You can also read about the entire package development process online in Hadley Wickham's R Packages book

Exercise No: 30	HOW TO INSTALL R PROGRAMMING
Date:	

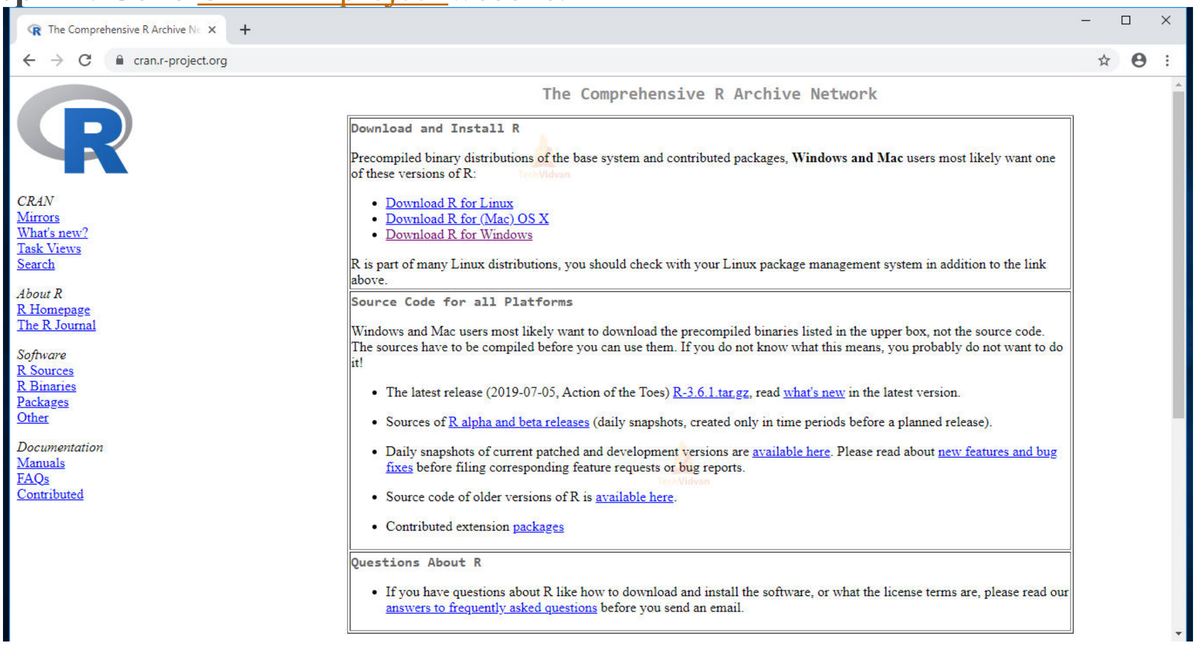
Aim:

Procedure:

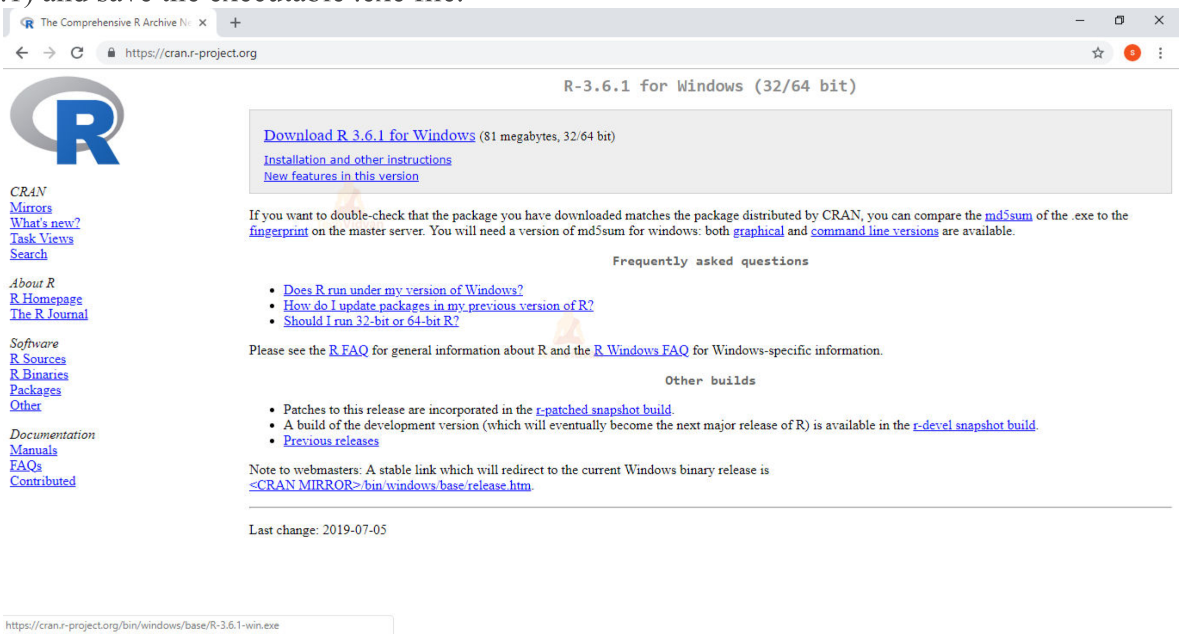
To install R and RStudio on windows, go through the following steps:

Install R on windows

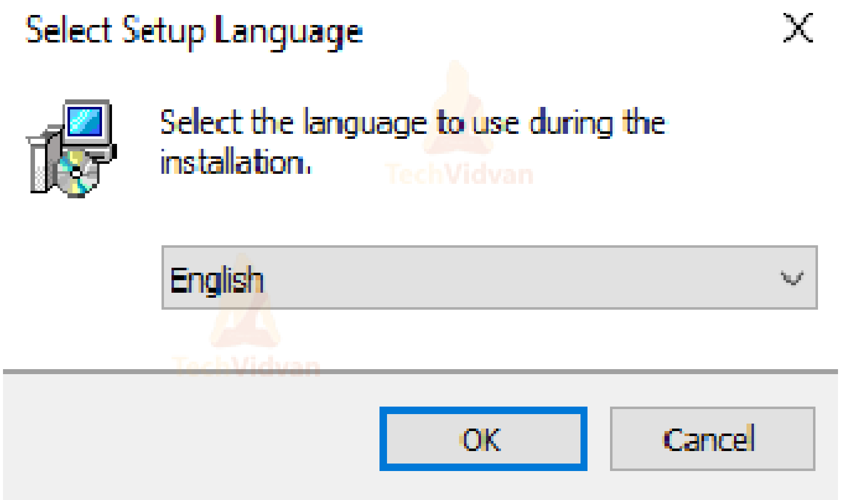
Step – 1: Go to CRAN R project website.



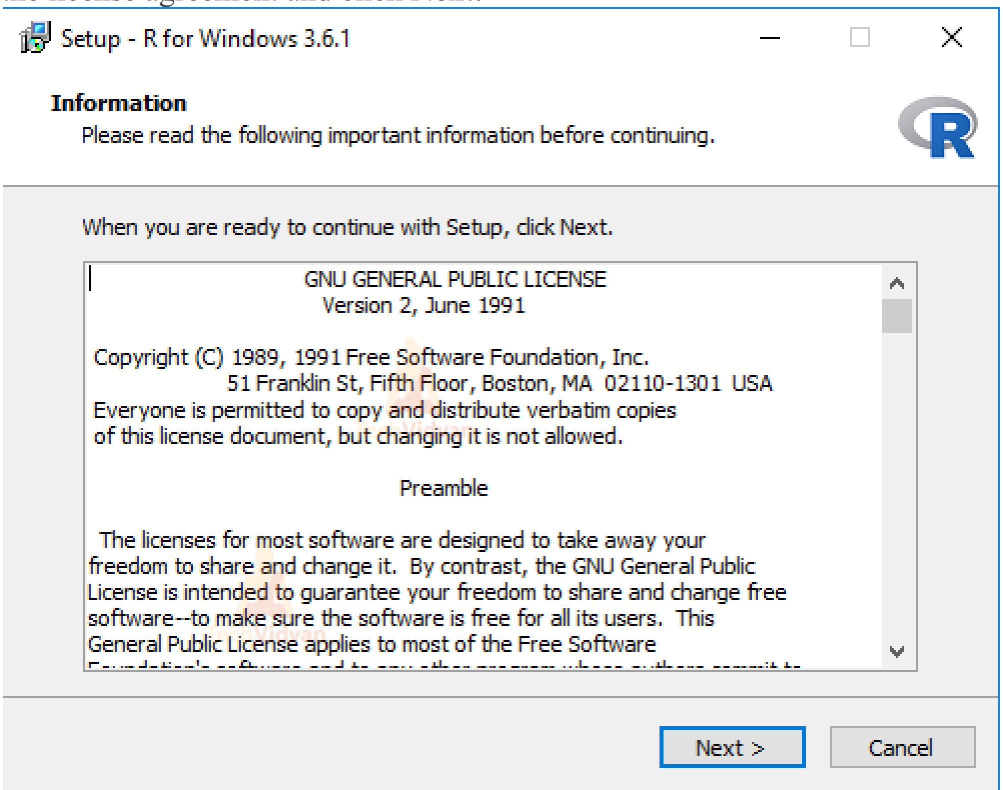
Step – 2: Click on the **Download R for Windows** link.
Step – 3: Click on the **base** subdirectory link or **install R for the first time** link.
Step – 4: Click **Download R X.X.X for Windows** (X.X.X stand for the latest version of R. eg: 3.6.1) and save the executable .exe file.



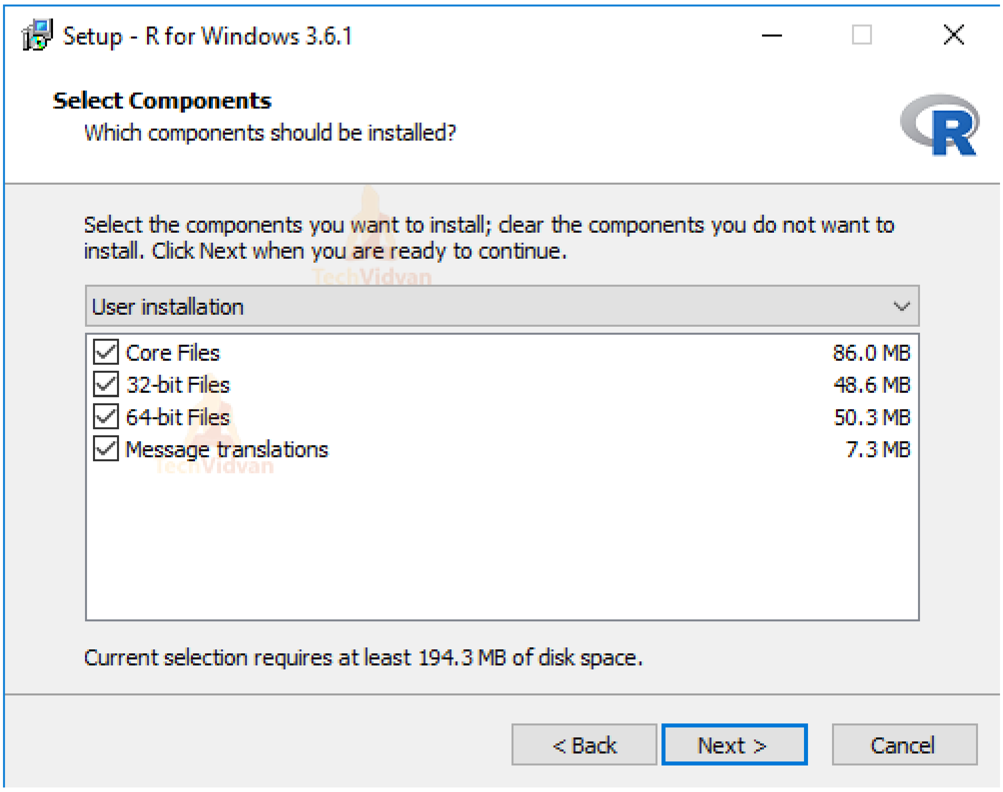
Step – 5: Run the **.exe** file and follow the installation instructions.
5.a. Select the desired language and then click **Next**.



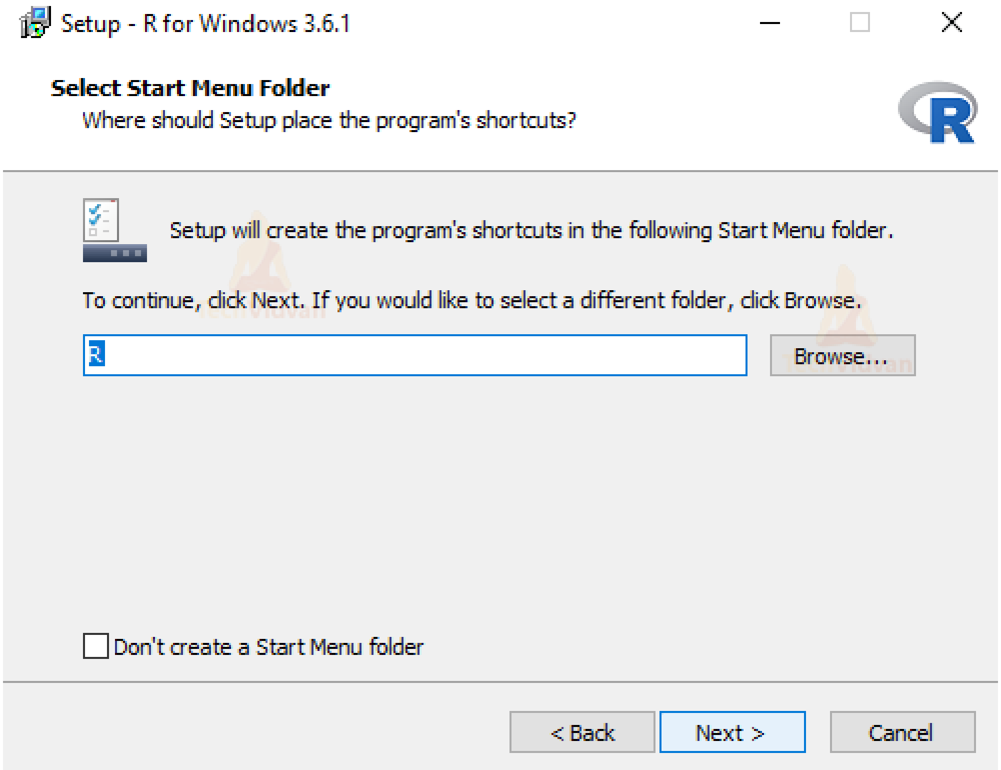
5.b. Read the license agreement and click **Next**.



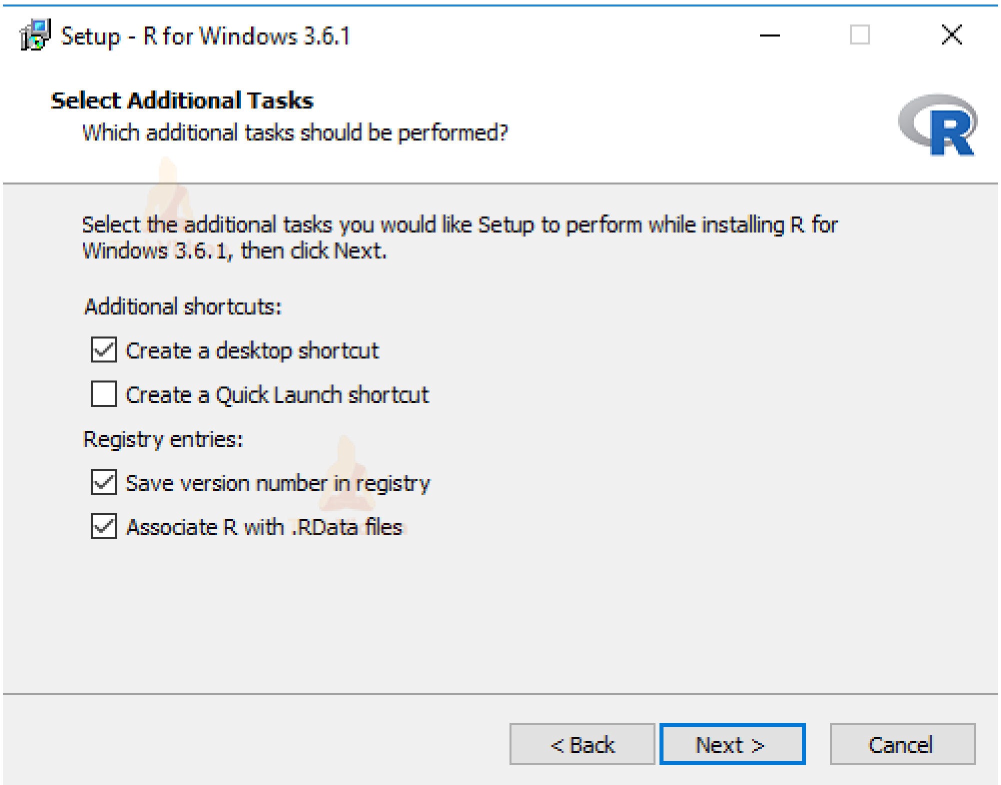
5.c. Select the components you wish to install (it is recommended to install all the components). Click **Next**.



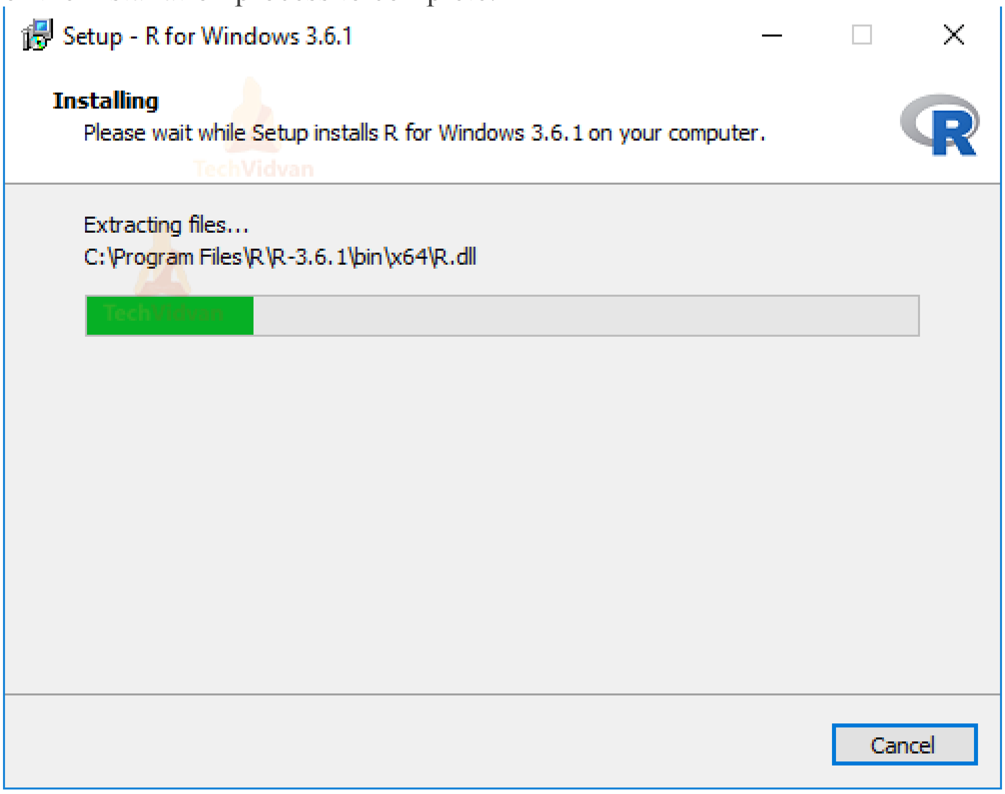
5.d. Enter/browse the folder/path you wish to install R into and then confirm by clicking **Next**.



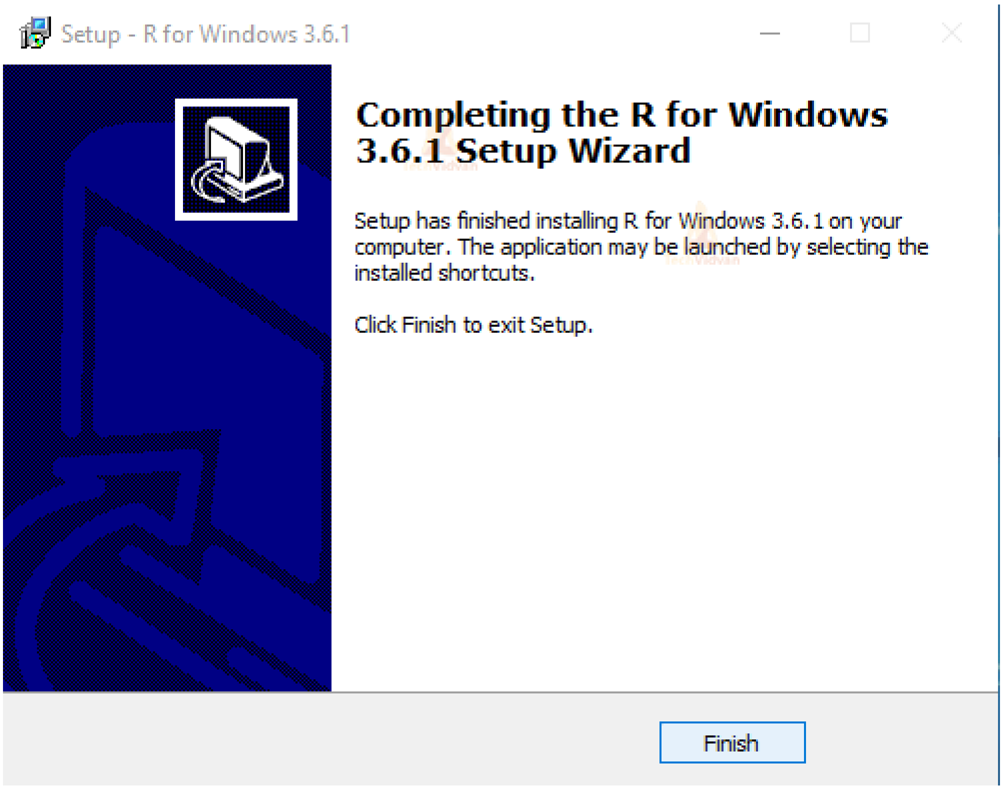
5.e. Select additional tasks like creating desktop shortcuts etc. then click **Next**.



5.f. Wait for the installation process to complete.

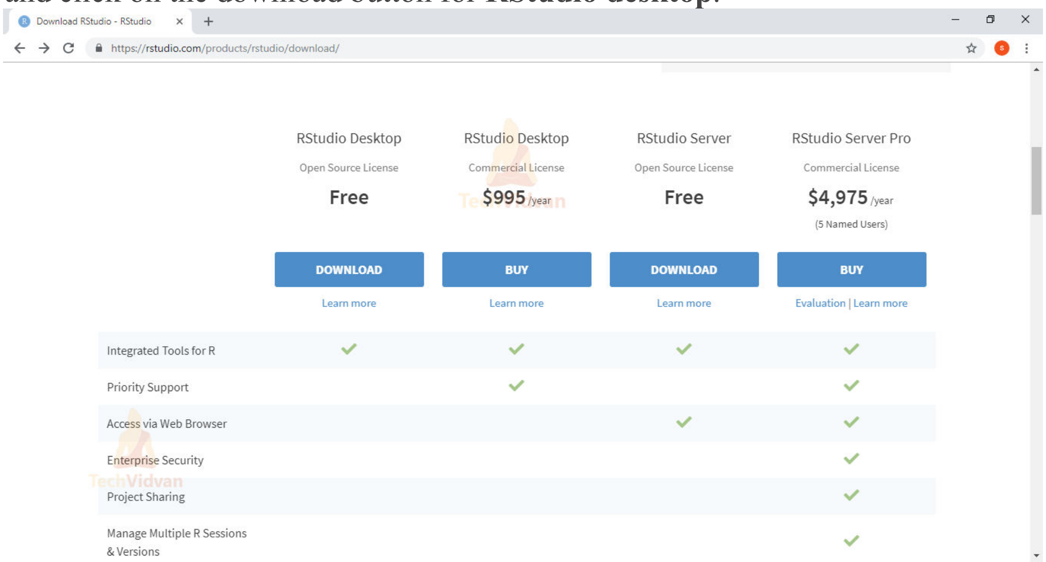


5.g. Click on **Finish** to complete the installation.

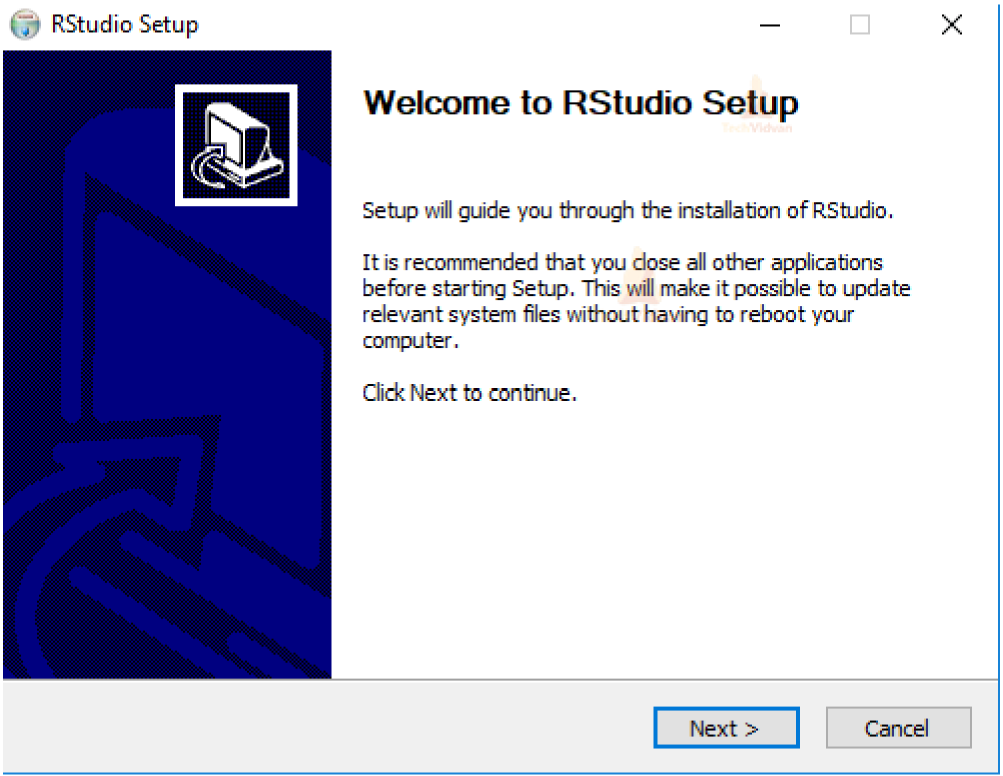


Install RStudio on Windows

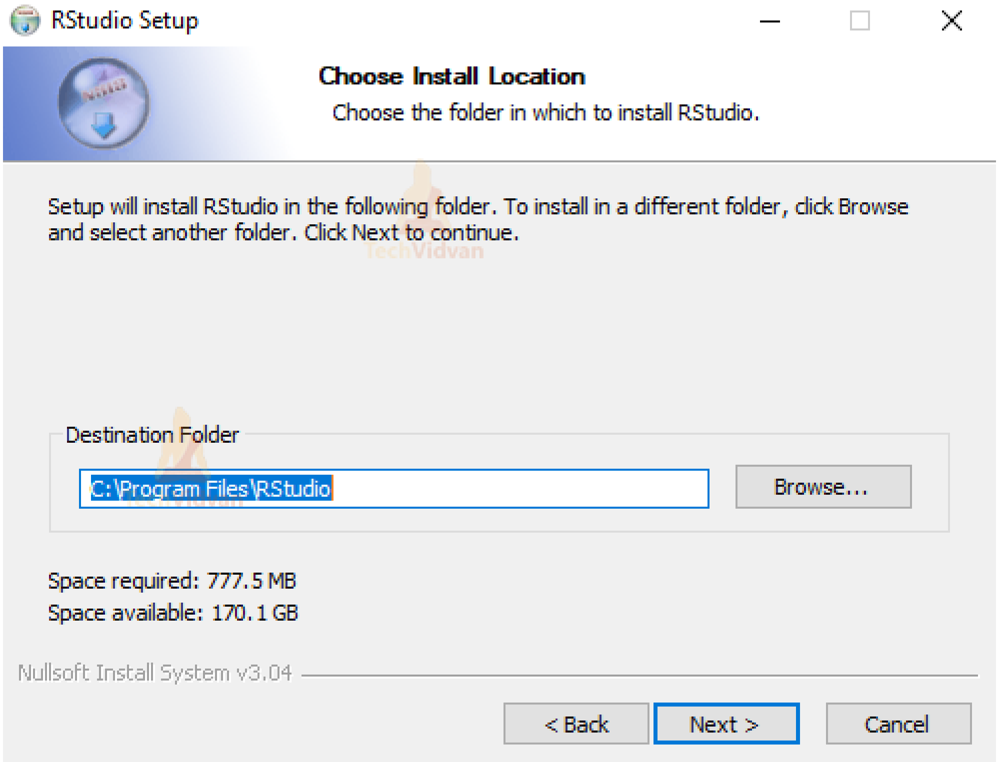
Step – 1: With R-base installed, let's move on to installing RStudio. To begin, go to download RStudio and click on the download button for **RStudio desktop**.



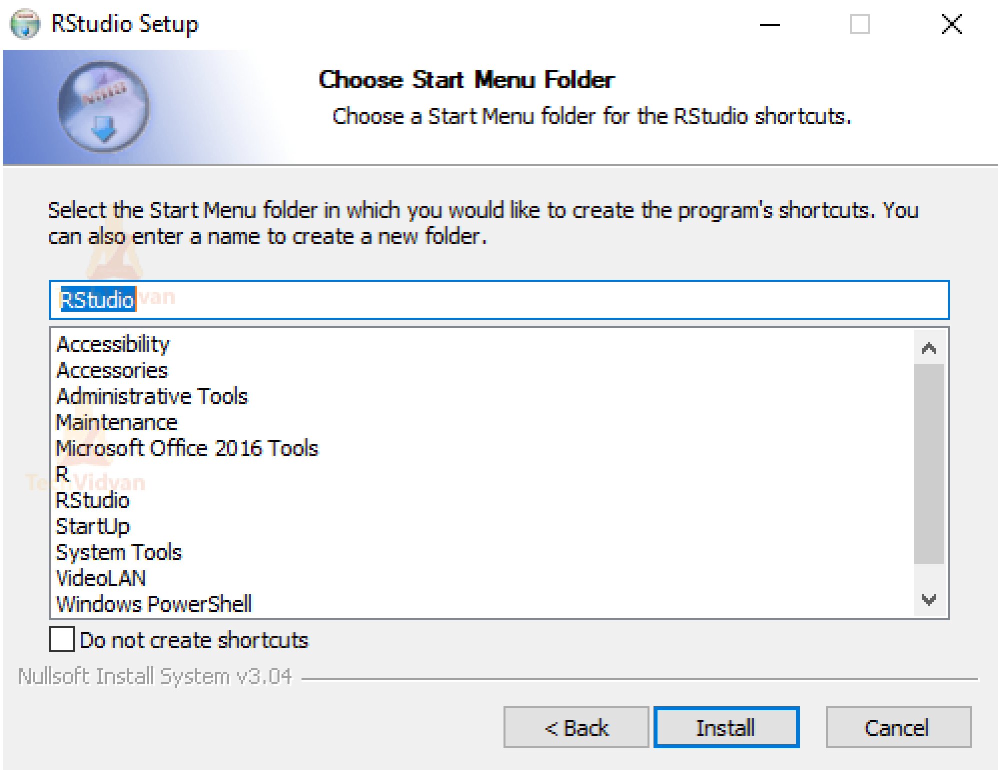
Step – 2: Click on the link for the windows version of RStudio and save the .exe file.
Step – 3: Run the .exe and follow the installation instructions.
3.a. Click **Next** on the welcome window.



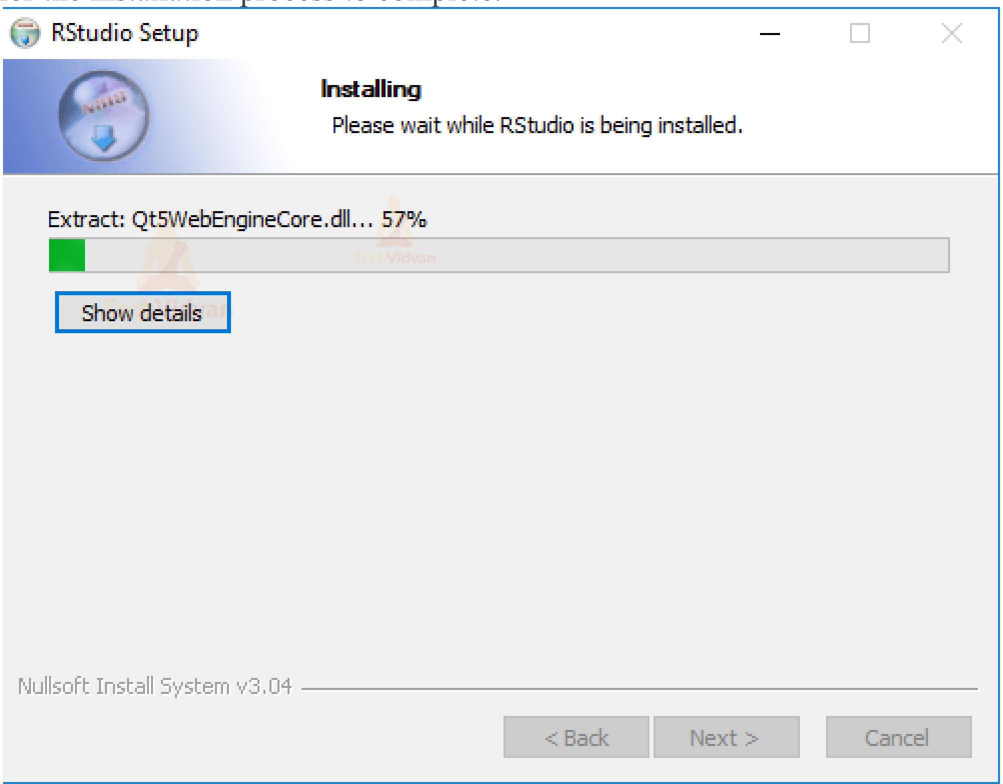
3.b. Enter/browse the path to the installation folder and click **Next** to proceed.



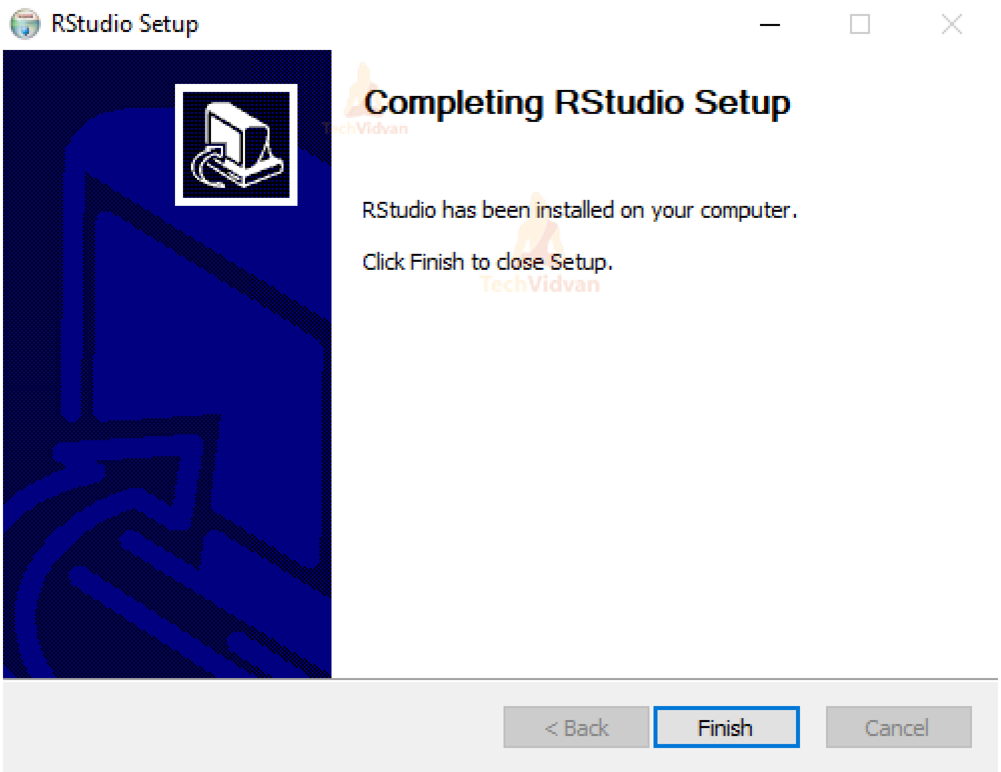
3.c. Select the folder for the start menu shortcut or click on do not create shortcuts and then click **Next**.



3.d. Wait for the installation process to complete.



3.e. Click **Finish** to end the installation.


RStudio Setup
Completing RStudio Setup
RStudio has been installed on your computer.
Click Finish to close Setup.
< Back
Finish
Cancel

INFERENCE:

Exercise No:31	UNDERSTANDING BASIC COMMANDS IN R
Date:	

AIM:

To understand the basic commands in R programming.

PROCEDURE:

R as a calculator

```
> 1 + 2
[1] 3
```

Variable Assignment

We assign values to variables with the assignment operator "=". Just typing the variable by itself at the prompt will print out the value. We should note that another form of assignment operator "<-" is also in use.

```
> x = 1
> x
[1]
1
```

Functions

R functions are invoked by its name, then followed by the parenthesis, and zero or more arguments. The following apply the function c to combine three numeric values into a vector.

```
> c(1, 2, 3)
[1] 1 2 3
```

Comments

All text after the pound sign "#" within the same line is considered a comment.

```
> 1 + 1      # this is a comment
[1] 2
```

R Data Type

- ➢ Numeric
- ➢ Integer
- ➢ Complex
- ➢ Logical
- ➢ Character

Numeric:

Decimal values are called numeric sin R. It is the default computational data type. If we assign a decimal value to a variable x as follows, x will be of numeric type.

```
> x = 10.5       # assign a decimal value
> x              # print the value of x
[1] 10.5
> class(x)       # print the class name of x
[1] "numeric"
```

Furthermore, even if we assign an integer to a variable k, it is still being saved as a numeric value.

```
> k = 1
> k              # print the value of k
[1] 1
> class(k)       # print the class name of k
[1] "numeric"
```

Integer:

In order to create an integer variable in R, we invoke the as.integerfunction. We can be assured that y is indeed an integer by applying the is.integer function.

```
> y = as.integer(3)
> y              # print the value of y
[1] 3
> class(y)       # print the class name of y
[1] "integer"
> is.integer(y) # is y an integer?
[1] TRUE
```

Incidentally, we can compel a numeric value into an integer with the same as.integerfunction.

```
> as.integer(3.14)    # coerce a numeric value
[1] 3
```

And we can parse a string for decimal values in much the same way.

```
> as.integer("5.27") # coerce a decimal string [1] 5
```

On the other hand, it is erroneous trying to parse a non-decimal string.

```
> as.integer("Joe") # coerce an non−decimal string
[1] NA
Warning message:
NAs introduced by coercion
```

Often, it is useful to perform arithmetic on logical values. Like the C language, TRUE has the value 1, while FALSE has value 0.

```
> as.integer(TRUE)    # the numeric value of TRUE [1]
1
> as.integer(FALSE) # the numeric value of FALSE [1]
0
```

Complex:

complexvalue in R is defined via the pure imaginary value i.

```
> z = 1 + 2i    # create a complex number
> z            # print the value of z [1]
1+2i
> class(z)      # print the class name of z [1]
"complex"
```

The following gives an error as −1 is not a complex value.

```
> sqrt(−1)      # square root of −1
[1] NaN
Warning message:
In sqrt(−1) : NaNs produced
```

Instead, we have to use the complex value −1 + 0i.

```
> sqrt(−1+0i)    # square root of −1+0i
[1] 0+1i
```

An alternative is to coerce −1 into a complex value.

```
> sqrt(as.complex(−1))
[1] 0+1i
```

Logical:

A logical value is often created via comparison between variables.

```
> x = 1; y = 2 # sample values
> z = x > y     # is x larger than y?
```

```
> z              # print the logical value
[1] FALSE
> class(z)       # print the class name of z
[1] "logical"
```

Standard logical operations are "&" (and), "|" (or), and "!" (negation).

```
> u = TRUE; v = FALSE
> u & v          # u AND v
[1] FALSE
> u | v          # u OR v
[1] TRUE
> !u             # negation of u
[1] FALSE
```

Character:

A character object is used to represent string values in R. We convert objects into character values with the as.character() function:

```
> x = as.character(3.14)
> x              # print the character string
[1] "3.14"
> class(x)       # print the class name of x
[1] "character"
```

Two character values can be concatenated with the paste function.

```
> fname = "Joe"; lname ="Smith"
> paste(fname, lname)  [1]
"Joe Smith"
```

However, it is often more convenient to create a readable string with the sprintf function, which has a C language syntax.

```
> sprintf("%s has %d dollars", "Sam", 100)  [1]
"Sam has 100 dollars"
```

To extract a substring, we apply the substr function. Here is an example showing how to extract the substring between the third and twelfth positions in a string.

```
> substr("Mary has a little lamb.", start=3, stop=12)  [1]
"ry has a l"
```

And to replace the first occurrence of the word "little" by another word "big" in the string, we apply the sub function.

```
> sub("little", "big", "Mary has a little lamb.")
[1] "Mary has a big lamb."
```

Vector:

A vector is a sequence of data elements of the same basic type. Members in a vector are officially called components. Nevertheless, we will just call them members in this site.

Here is a vector containing three numeric values 2, 3 and 5.

```
> c(2, 3, 5)
[1] 2 3 5
```

And here is a vector of logical values.

```
> c(TRUE, FALSE, TRUE, FALSE, FALSE)
[1] TRUE FALSE TRUE FALSE FALSE
```

A vector can contain character strings.

```
> c("aa", "bb", "cc", "dd", "ee")
[1] "aa" "bb" "cc" "dd" "ee"
```

Incidentally, the number of members in a vector is given by the length function.

```
> length(c("aa", "bb", "cc", "dd", "ee"))
[1] 5
```

Combining vectors:

Vectors can be combined via the function c. For examples, the following two vectors n and s are combined into a new vector containing elements from both vectors.

```
> n = c(2, 3, 5)
> s = c("aa", "bb", "cc", "dd", "ee")
> c(n, s)
[1] "2" "3" "5" "aa" "bb" "cc" "dd" "ee"
```

Vector Arithmetic:

Arithmetic operations of vectors are performed member-by-member, i.e., memberwise.

For example, suppose we have two vectors a and b.

```
> a = c(1, 3, 5, 7)
> b = c(1, 2, 4, 8)
```

Then, if we multiply a by 5, we would get a vector with each of its members multiplied by 5.

```
> 5 * a
[1] 5 15 25 35
```

And if we add a and b together, the sum would be a vector whose members are the sum of the corresponding members from a and b.

```
> a + b
[1]  2  5  9 15
```

Similarly for subtraction, multiplication and division, we get new vectors via memberwise operations.

```
> a - b
[1]  0  1  1 -1

> a * b
[1] 1 6 20 56

> a / b
[1] 1.000 1.500 1.250 0.875
```

Recycling Rule

If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. For example, the following vectors u and v have different lengths, and their sum is computed by recycling values of the shorter vector u.

```
> u = c(10, 20, 30)
> v = c(1, 2, 3, 4, 5, 6, 7, 8, 9)
> u + v
[1] 11 22 33 14 25 36 17 28 39
```

Vector Index:

We retrieve values in a vector by declaring an index inside a single square bracket "[]" operator.

For example, the following shows how to retrieve a vector member. Since the vector index is 1-based, we use the index position 3 for retrieving the third member.

```
> s = c("aa", "bb", "cc", "dd", "ee")
> s[3]
[1] "cc"
```

Unlike other programming languages, the square bracket operator returns more than just individual members. In fact, the result of the square bracket operator is another vector, and s[3] is a vector slice containing a single member "cc".

Negative Index

If the index is negative, it would strip the member whose position has the same absolute value as the negative index. For example, the following creates a vector slice with the third member removed.

```
> s[-3]
[1] "aa" "bb" "dd" "ee"
```

Out-of-Range Index

If an index is out-of-range, a missing value will be reported via the symbol NA.

```
> s[10]
[1] NA
```

Numeric Index vector:

A new vector can be sliced from a given vector with a numeric index vector, which consists of member positions of the original vector to be retrieved.

Here it shows how to retrieve a vector slice containing the second and third members of a given vector s.

```
> s = c("aa", "bb", "cc", "dd", "ee")
> s[c(2, 3)]
[1] "bb" "cc"
```

Duplicate Indexes

The index vector allows duplicate values. Hence the following retrieves a member twice in one operation.

```
> s[c(2, 3, 3)]
[1] "bb" "cc" "cc"
```

Out-of-Order Indexes

The index vector can even be out-of-order. Here is a vector slice with the order of first and second members reversed.

```
> s[c(2, 1, 3)]
[1] "bb" "aa" "cc"
```

Range Index

To produce a vector slice between two indexes, we can use the colon operator ":". This can be convenient for situations involving large vectors.

```
> s[2:4]
[1] "bb" "cc" "dd"
```

Logical Index vector:

A new vector can be sliced from a given vector with a logical index vector, which has the same length as the original vector. Its members are TRUE if the corresponding members in the original vector are to be included in the slice, and FALSE if otherwise.

For example, consider the following vector s of length 5.

```
> s = c("aa", "bb", "cc", "dd", "ee")
```

To retrieve the the second and fourth members of s, we define a logical vector L of the same length, and have its second and fourth members set as TRUE.

```
> L = c(FALSE, TRUE, FALSE, TRUE, FALSE)
> s[L]
[1] "bb" "dd"
```

The code can be abbreviated into a single line.

```
> s[c(FALSE, TRUE, FALSE, TRUE, FALSE)]
[1] "bb" "dd"
```

List

A list is a generic vector containing other objects.

For example, the following variable x is a list containing copies of three vectors n, s, b, and a numeric value 3.

```
> n = c(2, 3, 5)
> s = c("aa", "bb", "cc", "dd", "ee")
> b = c(TRUE, FALSE, TRUE, FALSE, FALSE)
> x = list(n, s, b, 3)   # x contains copies of n, s, b
```

List Slicing

We retrieve a list slice with the *single square bracket* "[]" operator. The following is a slice containing the second member of x, which is a copy of s.

```
> x[2]
[[1]]
[1] "aa" "bb" "cc" "dd" "ee"
```

With an index vector, we can retrieve a slice with multiple members. Here a slice containing the second and fourth members of x.

```
> x[c(2, 4)]
[[1]]
[1] "aa" "bb" "cc" "dd" "ee"


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

List Member Reference

In order to reference a list member directly, we have to use the *double square bracket*"[[]]" operator. The following object x[[2]] is the second member of x. In other words, x[[2]] is a copy of s, but is *not* a slice containing s or its copy.

```
> x[[2]]
[1] "aa" "bb" "cc" "dd" "ee"
```

We can modify its content directly.

```
> x[[2]][1] = "ta"
> x[[2]]
[1] "ta" "bb" "cc" "dd" "ee"
> s
[1] "aa" "bb" "cc" "dd" "ee" # s is unaffected
```

Data Frame

A data frame is used for storing data tables. It is a list of vectors of equal length. For example, the following variable dfis a data frame containing three vectors n, s, b.

```
> n = c(2, 3, 5)
> s = c("aa", "bb", "cc")
> b = c(TRUE, FALSE, TRUE)
> df = data. Frame (n, s, b)       # df is a data frame
```

Build-in Data Frame

We use built-in data frames in R for our tutorials. For example, here is a built-in data frame in R, called mtcars.

```
> mtcars
            mpg cyl disp hp drat wt ...
Mazda RX4      21.0 6 160 110 3.90 2.62 ...
Mazda RX4 Wag 21.0 6 160 110 3.90 2.88 ...
Datsun 710    22.8 4 108 93 3.85 2.32 ...
            ............
```

The top line of the table, called the header, contains the column names. Each horizontal line afterward denotes a data row, which begins with the name of the row, and then followed by the actual data. Each data member of a row is called a cell.

To retrieve data in a cell, we would enter its row and column coordinates in the *single square bracket* "[]" operator. The two coordinates are separated by a comma. In other words, the coordinates begins with row position, then followed by a comma, and ends with the column position. The order is important.

Here is the cell value from the first row, second column of mtcars.

```
> mtcars[1, 2]
[1] 6
```

Moreover, we can use the row and column names instead of the numeric coordinates.

- Mtcars(“Mazda RX4”,”cyl”)
  [1] 6

Lastly, the number of data rows in the data frame is given by the nrowfunction.

```
> nrow(mtcars)    # number of data rows
[1] 32
```

And the number of columns of a data frame is given by the ncolfunction.

```
> ncol(mtcars)    # number of columns
[1] 11
```

Further details of the mtcarsdata set is available in the R documentation.

```
>  help(mtcars)
```

Instead of printing out the entire data frame, it is often desirable to preview it with the head function beforehand.

```
> head(mtcars)
           mpg cyl disp hp drat wt ...
Mazda RX4     21.0 6 160 110 3.90 2.62 ...
           ............
```

INFERENCE:

Exercise No: 32	IMPORTING & EXPORTING DATA IN R
Date:	

EX.NO: 32

AIM:

To importing and exporting the data using R Programming

PROCEDURE:

The sample data can also be in **comma separated values** (CSV) format. Each cell inside such data file is separated by a special character, which usually is a comma, although other characters can be used as well.

The first row of the data file should contain the column names instead of the actual data. Here is a sample of the expected format.

```
Col1, Col2, Col3
100, a1, b1
200, a2, b2
300, a3, b3
```

After we copy and paste the data above in a file named "mydata.csv" with a text editor, we can read the data with the function read.csv.

```
> mydata = read.csv("mydata.csv") # read csv file
> mydata
  Col1 Col2 Col3
1 100    a1   b1
2 200    a2   b2
3 300    a3   b3
```

```
# Write CSV in R
>write.csv (My Data, file = "MyData.csv",row.names=FALSE)
```

INFERENCE:

Exercise No: 33	EXPLORING THE DATA USING R STUDIO
Date:	

AIM

To write a R program to explore the data using R studio.

PROCEDURE

Step 1: Create a sample dataset**.**

Step 2: Install and load the ggplot2 package for data visualization.

Step 3: Create a histogram of ages.

Step 4: Create a box plot of income.

SOURCE CODE

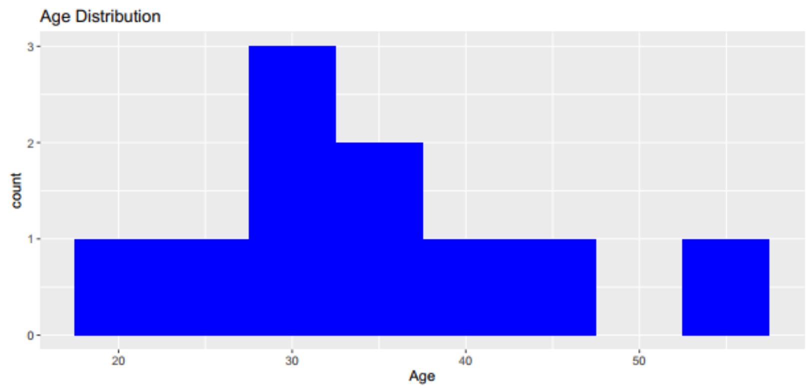
```
# Create a sample dataset
data <- data.frame(
  ID = 1:10,
  Age = c(25, 30, 22, 40, 35, 28, 55, 29, 37, 45),
  Income = c(40000, 50000, 32000, 60000, 55000, 45000, 75000, 52000, 68000, 80000),
  Education = c("High School", "Bachelor's", "High School", "Master's", "PhD", "Bachelor's", "PhD",
"Master's", "PhD", "Bachelor's"),
  Gender = c("Male", "Female", "Male", "Female", "Male", "Male", "Female", "Female", "Male",
"Male")
)
head(data)
summary(data$Age)
summary(data$Income)
table(data$Education)
table(data$Gender)
# Install and load the ggplot2 package for data visualization
install.packages("ggplot2")
library(ggplot2)

# Create a histogram of ages
ggplot(data, aes(x = Age)) + geom_histogram(binwidth = 5, fill = "blue") + labs(title = "Age
Distribution")
```

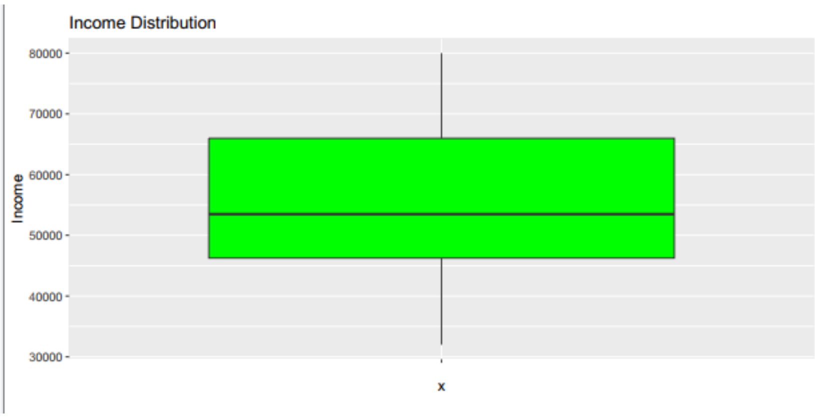
Create a box plot of income

ggplot(data, aes(x = "", y = Income)) + geom_boxplot(fill = "green") + labs(title = "Income Distribution")

OUTPUT

```
ID Age Income   Education Gender
1  1  25  40000 High School   Male
2  2  30  50000  Bachelor's Female
3  3  22  32000 High School   Male
4  4  40  60000    Master's Female
5  5  35  55000         PhD   Male
6  6  28  45000  Bachelor's   Male

Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  22.00   28.25   32.50   34.60   39.25   55.00

Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  32000   46250   53500   55700   66000   80000

Bachelor's High School    Master's         PhD
          3           2           2           3

Female   Male
     4      6
```





RESULT

Thus the data has been explored in various ways using R studio successfully.

Exercise No: 34	EXPLORING THE DATA USING R STUDIO
Date:	

AIM:

To creating a Pie Chart in R Programming using Command

PROCEDURE:

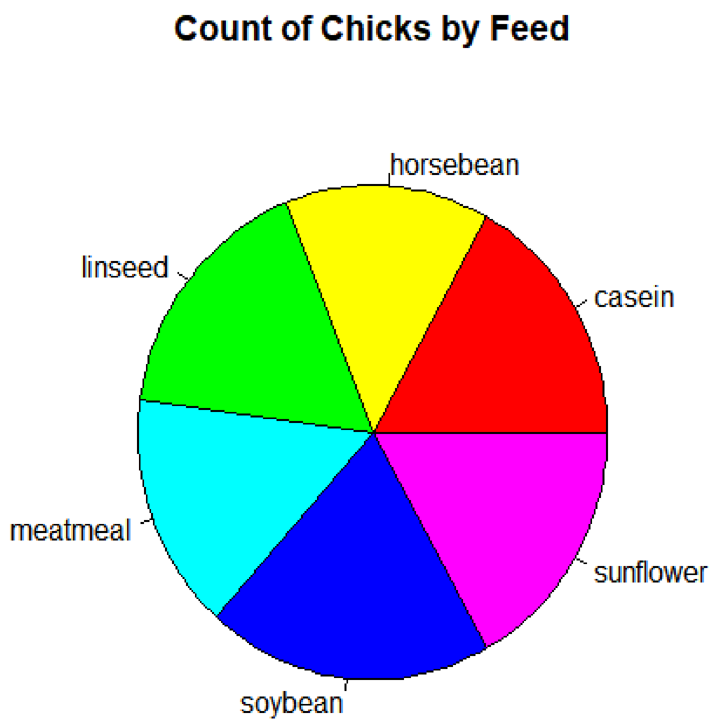
A **pie chart** of a qualitative data sample consists of pizza wedges that show the frequency distribution graphically.

The following script creates a pie chart.

1. This starts a pie chart function. The "x" parameter is the data that needs to be charted. In this line, the *feed* variable in the chickwts data frame is extracted to a table since the pie chart 2.These lines define the main title and colors used for the pie chart. These parameters are the same as was seen in other graphs in this lab.

3. This tells *R* to use the labels used in the *feed* variable as the labels on the pie chart.

```
# Count of Chicks by Feed pie(x =

table(chickwts$feed),

  main = "Count of Chicks by Feed", col =

  rainbow(6),

  labels = c(levels(chickwts$feed))
  )
```


Count of Chicks by Feed
horsebean
linseed
casein
meatmeal
sunflower
soybean

INFERENCE:

Exercise No: 35	CREATING HISTOGRAM IN R
Date:	

AIM:

To Create a Histogram in R Programming Command.

PROCEDURE:

A histogram is a graph that shows the distribution of data that are continuous in nature, for example, age or height. A histogram resembles a bar chart but there is an important difference: a histogram is used for continuous data while a bar chart is used for categorical data. To emphasize that difference, histograms are normally drawn with no space between the bars (the data are continuous along the entire x-axis) while bar charts are normally drawn with a small space between bars (the data are categorical along the x-axis).

The following script creates an example histogram.

Line 2: This is the beginning of the histogram function (it ends on Line 8). For this histogram the Speed variable from the morley data frame is specified as the data source for the histogram.

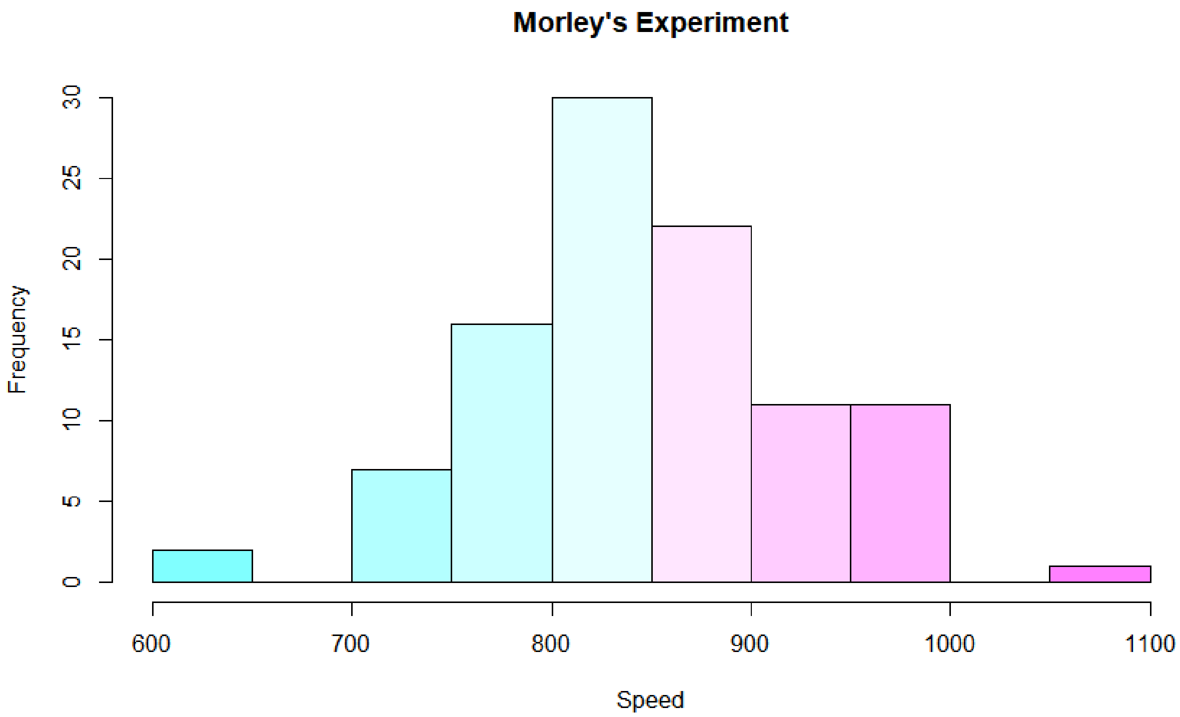
Lines 3-5: This creates the main title of the histogram along with the labels for the x-axis and y-axis.

Line 6: To create a histogram, R analyzes the values contained in a variable and creates “bins” for those values. That means that many of the continuous values will be grouped into a single bin for analysis. The “breaks” parameter tells R how many breaks to allow in the variable. In this case, eight breaks are specified, which would create nine bins. R will analyze the data and use the “breaks” parameter as a “suggestion” and will only use that number of breaks if it makes sense for the data being graphed. Often, changing the number of breaks by just one or two will not change the histogram produced so researchers should play around with the “breaks” number to get the best possible representation of the data.

Line 7: This specifies that 10 colors will be used from the “cm.color” palette to shade the various bars in the histogram. Researchers need to experiment a bit with the color palette and number of colors to get the best result; however, “cm.color” along with the number of bars in the histogram seems to work well for this histogram. (Note: More information about color can be found in the About Colors section in the appendix.)

```
# Histogram

   hist(morley$Speed,

      main = "Morley's Experiment",

      xlab = "Speed",
```

```
ylab = "Frequency",

breaks = 8,

col = cm.colors(10)

)
```



INFERENCE:

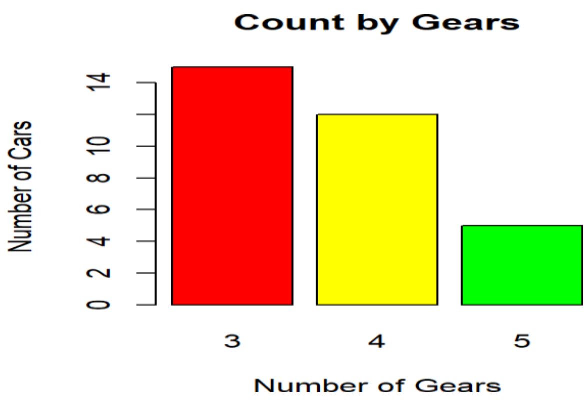
Exercise No: 36	CREATING BARPLOT IN R
Date:	

AIM:

Creating a Bar Plot using R Programming Commands.

PROCEDURE:

A bar graph of a qualitative data sample consists of vertical parallel bars that show the frequency distribution graphically. A bar plot is used to display the frequency count for categorical data. The following figure is a bar plot showing the number of automobiles with three, four, and five gears according to the mtcars data frame.



These types of visuals are more effective than a table full of numbers and they are easy to generate with *R*.

Steps: Bar Plot

1. The following script creates a simple bar plot. Note: this is one long R command that has been broken up over several lines to make it easier to understand.

2. This creates a bar plot using the barplot function. The first argument sent to the function is the data source for the heights of each bar in the plot. In this case, R creates a table from the gears variable in mtcars and then uses that table as data input for the plot. All of the other lines in this script embellish the bar plot to make it more usable.

3. The "main" attribute sets the main title for the bar plot. In general, for any graphic in R main is used to set the title of the graph.

4. This creates the label for the x-axis.

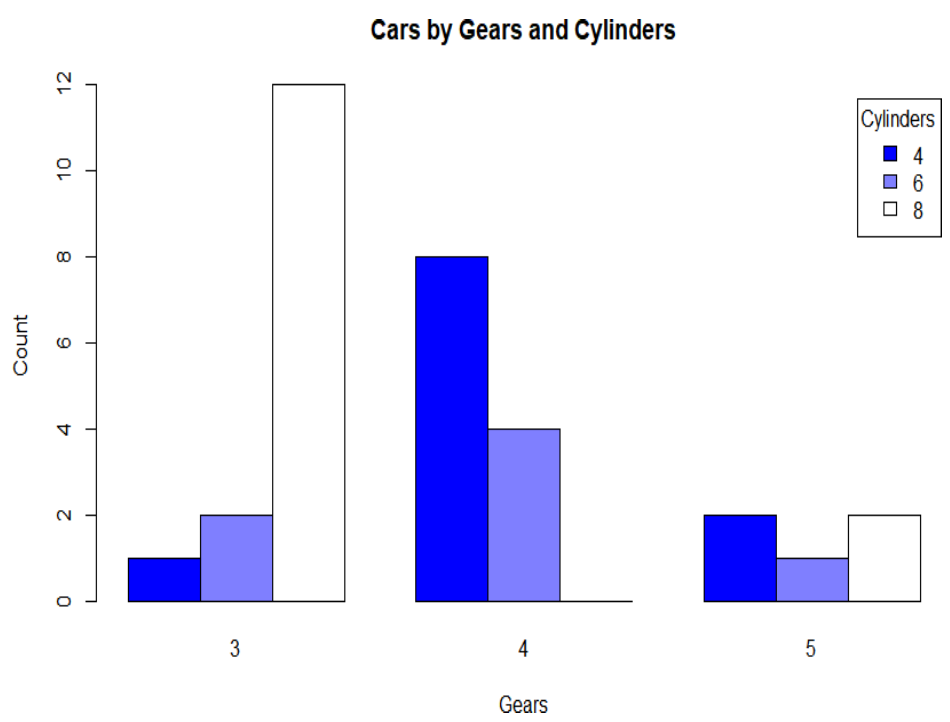
5. This creates the label for the y-axis.

6. This sets the color palette for the graph. In this case, the rainbow palette is used for the graph. Three colors were requested from that palette but specifying any number larger than three would have worked and created a slightly different palette. Experimentation is needed to find the most suitable palette for any given graph. (Note: More information can be found in the About Colors section in the appendix.)

```
# Simple Bar Plot
barplot(height = table(mtcars$gear),
  main = "Number of Cars By Gears",
  xlab = "Gears",
  ylab = "Count",
  col = cm.colors(3)
)
```

Clustered Bar Plot

A clustered bar plot (sometimes called a “Grouped Bar plot”) displays two or more categorical variables. In general, clustered bar plots are best at showing relationships between variables but not so good for determining the absolute size of each variable.



```
# Clustered Bar Plot With Gradient Colors colpal <-
colorRampPalette(c("blue", "white")) barplot(height
= table(mtcars$cyl, mtcars$gear), main = "Cars by
Gears and Cylinders",
  xlab = "Gears", ylab =
  "Count",    legend    =
  TRUE,     beside     =
  TRUE,
  args.legend = list(title = "Cylinders"),  col =
  colpal(3)
)
```

INFERENCE:

Exercise No: 37	HANDLING GGPLOT PACKAGE IN R STUDIO
Date:	

AIM

To give a demonstration about handling ggplot package in R studio.

PROCEDURE

ep 1: Load the ggplot2 library

Step 2: Supply sample data.

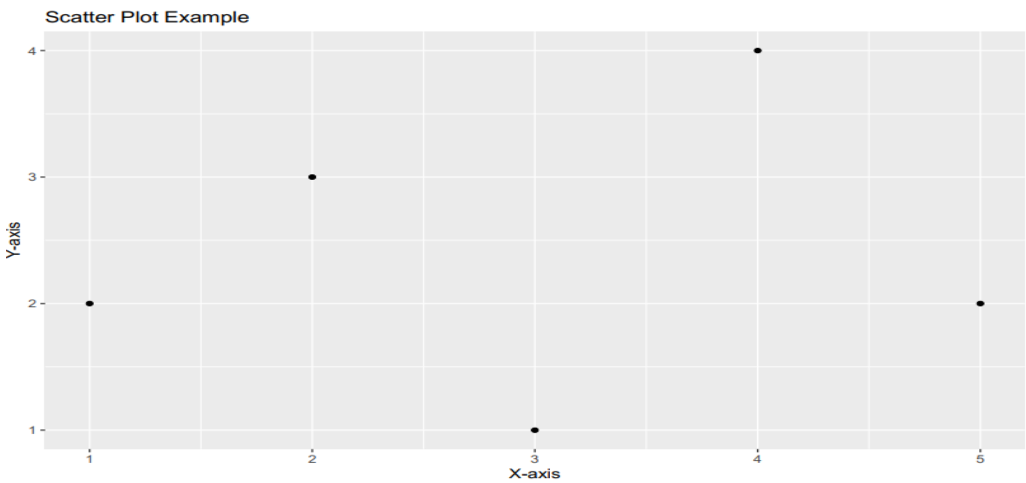
Step 3: Create a scatter plot

Step 4: Print the results.

SOURCE CODE

```
# Load the ggplot2 library
library(ggplot2)
# Sample data
data <- data.frame(
  X = c(1, 2, 3, 4, 5),
  Y = c(2, 3, 1, 4, 2)
)
# Create a scatter plot
ggplot(data, aes(x = X, y = Y)) +
  geom_point() +
  labs(title = "Scatter Plot Example", x = "X-axis", y = "Y-axis")
```

OUTPUT



RESULT

Thus the ggplot package has been handled and demonstrated successfully

Exercise No: 38	CREATING FREQUENCY DISTRIBUTION IN R
Date:	

AIM:

To calculate the Frequency distribution in R programming**.**

PROCEDURE:

A data sample is called **qualitative**, also known as **categorical**, if its values belong to a collection of known defined non-overlapping classes. Common examples include student letter grade (A, B, C, D or F), commercial bond rating (AAA, AAB,) and consumer clothing shoe sizes (1, 2, 3,).

The tutorials in this section are based on an R built-in data frame named **painters**. It is a compilation of technical information of a few eighteenth century classical painters. The data set belongs to the MASS package, and has to be pre-loaded into the R workspace prior to its use.

```
> install.packages(MASS)      # install the MASS package

> library(MASS)      # load the MASS package
> painters
              Composition Drawing Colour Expression School
Da Udine           10       8      16         3       A
Da Vinci           15      16       4        14       A
Del Piombo          8      13      16         7       A
Del Sarto          12      16       9         8       A
Fr. Penni           0      15       8         0       A
Guilio Romano      15      16       4        14       A
                 .................
```

The last School column contains the information of school classification of the painters.
The schools are named as A, B, ...,*etc*, and the School variable is qualitative.

```
> painters$School
 [1] A A A A A A A A A A B B B B B B C C C C C C D D D D
[27] D D D D D D E E E E E E E F F F F G G G G G G G H H
[53] H H
Levels: A B C D E F G H
```

For further details of the paintersdata set, please consult the R documentation.

```
>  help(painters)
```

The **frequency distribution** of a data variable is a summary of the data occurrence in a collection of non-overlapping categories.

Example

In the data set painters, the frequency distribution of the School variable is a summary of the number of painters in each school.

Problem

Find the frequency distribution of the painter schools in the data set painters.

Solution

We apply the table function to compute the frequency distribution of the School variable.

```
> library(MASS)                  # load the MASS package
> school = painters$School       # the painter schools
> school.freq = table(school) # apply the table function
```

Answer

The frequency distribution of the schools is:

```
> school.freq
school
 A B C D E F G H
10 6 6 10 7 4 7 4
```

INFERENCE:

Exercise No: 39	DESCRIPTIVE STATISTICS USING R
Date:	

AIM:

To create a Descriptive Statistics using R Programming Commands.

PROCEDURE:

```
>require("datasets") # Load Dataset Package
>data(cars)
>summary(cars$speed) # Summary for one variable  Min.
   1st Qu.  Median              Mean  3rd Qu.     Max.
        4.0    12.0     15.0          15.4      19.0         25.0
>summary(cars) # Summary for entire table  speeddist
 Min.  : 4.0   Min.   :  2.00  1st
 Qu.:12.0 1st Qu.: 26.00
Median :15.0   Median : 36.00
 Mean   :15.4   Mean   :   42.98   3rd
 Qu.:19.0   3rd   Qu.:   56.00   Max.
 :25.0   Max.   :120.00
>fivenum(cars$speed)  [1] 4
12 15 19 25
>help(package = "psych")
>install.packages("psych") #Install psych package
>require("psych") # Load psych package
>describe(cars)
vars  n  mean     sd median trimmed mad min max range skew kurtosis se  speed
            1 50 15.40  5.29      15    15.47  5.93   4  25     21 -0.11     -0.67 0.75
dist      2 50 42.98 25.77      36   40.88 23.72   2 120    118  0.76      0.12 3.64
```

INFERENCE:

Exercise No: 40	CORRELATION IN R
Date:	

AIM:
To analyze and interpret correlation using Pearson’s r, Spearman’s Rho and Kendall’s Tau Measures.

PROCEDURE:
Correlation is a method used to describe a relationship between the independent (or x-axis) and dependent (or y-axis) variables in some research project. A correlation is a number between -1.0 and +1.0, where 0.0 means there is no correlation between the two variables and either +1.0 or -1.0 means there is a perfect correlation. A positive correlation means that as one variable increases the other also increases.

Correlation Descriptions	
Correlation	**Description**
+.70 or higher	Very strong positive
+.40 to +.69	Strong positive
+.30 to +.39	Moderate positive
+.20 to +.29	Weak positive
+.19 to -.19	No or negligible
-.20 to -.29	Weak negative
-.30 to -.39	Moderate negative
-.40 to -.69	Strong negative
-.70 or less	Very strong negative

Pearson’s r

Pearson’s Product-Moment Correlation Coefficient (normally called Pearson’s r) is a measure of the strength of the relationship between two variables having continuous data that are normally distributed (they have bell-shaped curves).

The following script demonstrates how to calculate Pearson’s r.

Line 2: This is the start of the cor.test function, which calculates the correlation between two variables. That function requires the x-axis variable be listed first then the y-axis variable.

Line 3: This is a continuation of the cor.test function call and specifies the method to be Pearson’s r. Since Pearson’s r is the default method for the cor.test function this line did not need to be included but it is used in this example from the mtcars dataset,since the specification will be important in later examples in this lab.

```
# Pearson's r
cor.test(airquality$Wind,    airquality$Ozone,
  method = "pearson")
```

Pearson's product-moment correlation

```
data: airquality$Wind and airquality$Ozone
t = -8.0401, df = 114, p-value = 9.272e-13
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.7063918 -0.4708713
sample estimates:
       cor
-0.6015465
```

Categorical Data

When the one or both data elements are categorical then Spearman’s rho or Kendall’s tau is used to calculate the correlation. Other than the process used, the concept is exactly the same as for Pearson’s r and the result is a correlation between -1.0 and +1.0 where the strength and direction of the correlation is determined by its value. Spearman’s rho is used when at least one variable is ordered data and typically involves larger data samples while Kendall’s tau can be used for any type of categorical data but is more accurate for smaller data samples.

Spearman’s rho

The following script demonstrates using cor.test to calculate correlations from the esoph data frame using Spearman’s rho. The process for this calculation is the same as for Pearson’s r except the method specified is “spearman”. There is one other difference between this script and the first. Notice on line 2 that esoph$agegp is inside an as.numeric function. Since agegp uses text like “25-34” instead of a number this converts that to a number for Spearman’s rho. (Note: This script will generate a warnings about p-values but that can be safely ignored for this tutorial.)

```
# Spearman's rho
cor.test(as.numeric(esoph$agegp),    esoph$ncases,
method = "spearman")
```

Spearman's rank correlation rho

data: as.numeric(esoph$agegp) and
esoph$ncases S = 57515, p-value = 1.029e-
06
alternative hypothesis: true rho is not equal
to 0 sample estimates:
rho
0.49354
37

Kendall's tau

The following script demonstrates using cor.test to calculate correlations from the npk data frame using Kendall's tau. The process for this calculation is the same as for Pearson's r except the method specified is "kendall". As in the Spearman example, the first variable must be converted to numeric values. Also, this function will generate a warning but that can be ignored for this lab.

```
# Kendall's tau
cor.test(as.numeric(npk$N),
npk$yield, method = "kendall")
```

Kendall's rank correlation tau

data: as.numeric(npk$N) and
npk$yield z = 2.3687, p-value =
0.01785
alternative hypothesis: true tau is not equal to
0 sample estimates:
tau
0.41357
21

INFERENCE: