

FOUNDATIONS FOR DATA ANALYTICS

(CSE1006)

LAB MANUAL



Department of Computer Science and Engineering
VIT-AP University, Amaravati
Andhra Pradesh

LIST OF EXPERIMENTS

1. Basic Operations in R
2. Data Types and Functions in R
3. Data sorting
4. Finding and removing Duplicate records
5. Data Cleaning
6. Data Recording
7. Data Merging
8. Reading and Writing Data in R
9. Data Cleaning and Summarizing with dplyr package
10. Exploratory Data Analysis
11. Basic operations on Numpy
12. Computations on Arrays
13. NumPy's Structured arrays
14. Introducing Pandas Objects
15. Data Indexing and Selection
16. Operating on Data in Pandas
17. Handling missing data
18. Hierarchical Indexing
19. Vectorized String Operations
20. Visualization with Matplotlib

LESSON PLAN

Lecture No.	Lecture Topic
	R - Programming
1.	Basic Operations in R
2.	Data Types and Functions in R
3.	Data sorting
4.	Finding and removing Duplicate records
5.	Data Cleaning
6.	Data Recording
7.	Data Merging
8.	Reading and Writing Data in R
9.	Data Cleaning and Summarizing with dplyr package
10.	Exploratory Data Analysis
	Python Programming
1.	Basic operations on Numpy
2.	Computations on Arrays
3.	NumPy's Structured arrays
4.	Introducing Pandas Objects
5.	Data Indexing and Selection
6.	Operating on Data in Pandas
7.	Handling missing data, Hierarchical Indexing
8.	Vectorized String Operations

INTRODUCTION TO R

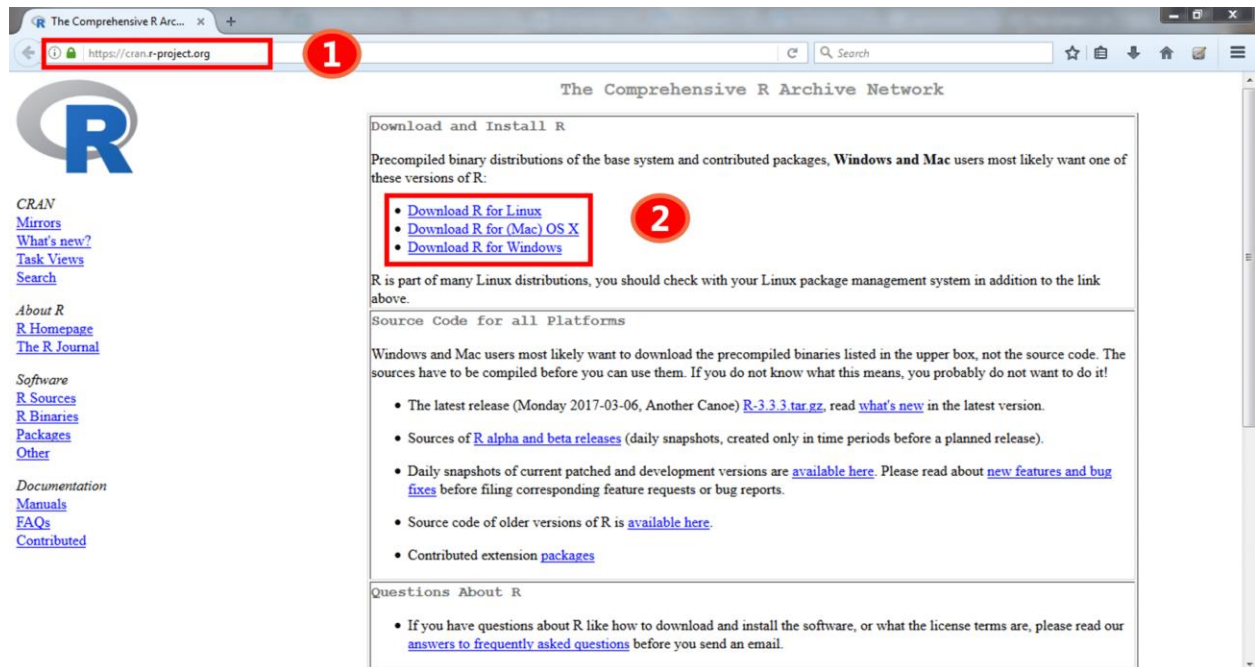
R Tutorial: Installation of R

Let me guide you through the process of installing R on your system. Just follow the below steps:

Step 1 : Go to the link- <https://cran.r-project.org/>

Step 2 : Download and install latest version of R on your system.

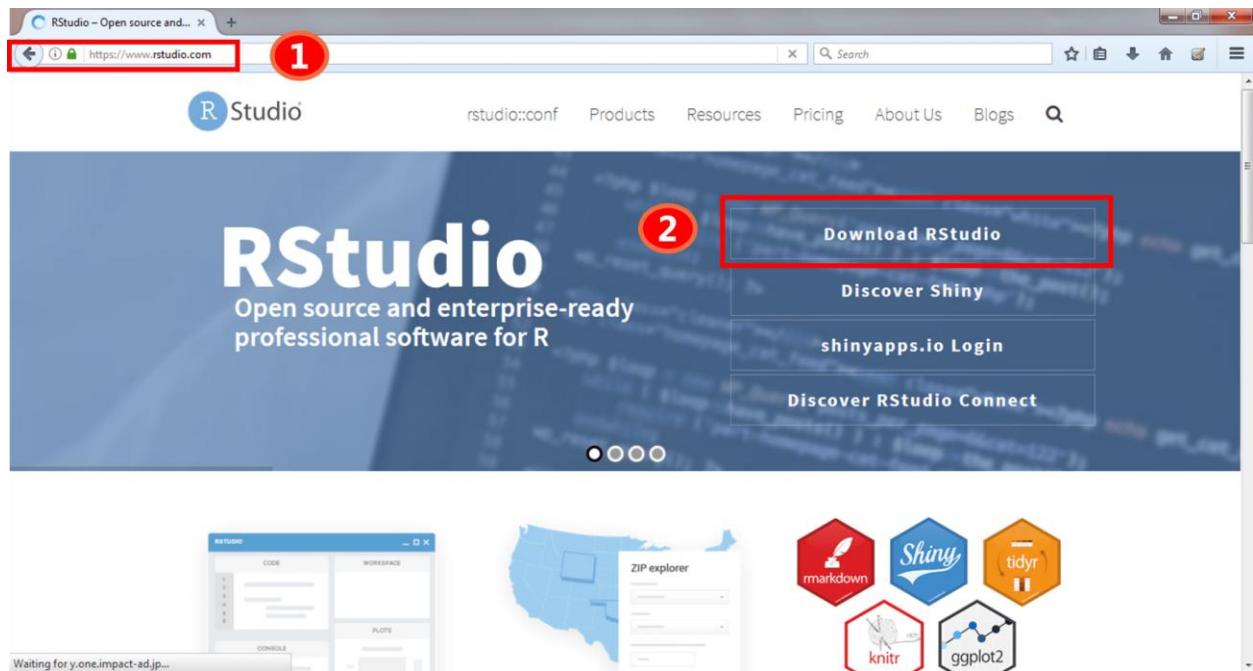
Refer to the below screenshot to get a better understanding.



By following the above steps, you are done with the R installation part. Now, you can directly start coding in R by downloading RStudio IDE. To download this, follow the below steps:

Step 1: Go to the link- <https://www.rstudio.com/>

Step 2: Download and install Rstudio on your system.



After installing everything, you are all set to code!

Next, let us move ahead in R Tutorial blog and understand what are data operators in R.
R Tutorial : Data Operators in R

There are mainly 5 different types of operators, which are listed below:

1. **Arithmetic Operators:** Perform arithmetic operations such as addition, subtraction, multiplication, division etc.
2. **Assignment Operators:** Assignment operators are used to assign values. For example:
 - **Assignment Operator =**
Syntax: variable name = value

```
>x=5
```

```
>x
```

```
Output : [1] 5
```

- **Assignment Operator <-**
Syntax: variable name <- value

- >x<- 15

>x

Output : [1] 15

- **Assignment Operator<<-**
Syntax : variable name <<- value

- >x <<- 2

>x

Output : [1] 2

- **Assignment Operator ->**
Syntax : value -> variable name

- >25 -> x
- >x

Output : [1] 25

3. Relational Operator: It defines a relation between two entities. For example: <,>,<=,!= etc.

>x<-3

>x!=2

Output: [1] TRUE

4. Logical Operators: These operators compare the two entities and are typically used with boolean (logical) values such as &, | and !.

>x<-2

>2&3

Output: [1] TRUE

5. Special Operators: These operators are used for specific purpose, not for logical computation. For example:

- It creates the series of numbers in sequence for a vector.

```
>x<-2:8
```

```
>x
```

```
Output: [1] 2 3 4 5 6 7 8
```

- %in% This operator is used to identify if an element belongs to a vector.
Example

- >x<- 2:8
- >y<-5
- >y %in% x

```
Output : [1] TRUE
```

R Tutorial: Data Types

Data types are used to store information. In R, we do not need to declare a variable as some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are mainly six data types present in R:

Let us go into more detail on each one of them:

Vector: A Vector is a sequence of data elements of the same basic type. Example:

```
vtr = (1, 3, 5, 7, 9)
```

or

```
vtr <- (1, 3, 5, 7, 9)
```

There are 5 Atomic vectors, also termed as five classes of vectors.

List: Lists are the R objects which contain elements of different types like – numbers, strings, vectors and another list inside it.

```
>n = c(2, 3, 5)
>s = c("aa", "bb", "cc", "dd", "ee")
>x = list(n, s, TRUE)
>x
```

Output –

```
[[1]]
[1] 2 3 5
[[2]]
[1] "aa" "bb" "cc" "dd" "ee"
[[3]]
[1] TRUE
```


Arrays: Arrays are the R data objects which can store data in more than two dimensions. It takes vectors as input and uses the values in the dim parameter to create an array.

```
vector1 <- c(5,9,3)
```

```
vector2 <- c(10,11,12,13,14,15)
```

```
result <- array(c(vector1,vector2),dim = c(3,3,2))
```

Output –

```
, , 1
```

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

```
[1,]  5  10  13
```

```
[2,]  9  11  14
```

```
[3,]  3  12  15
```

```
, , 2
```

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

```
[1,]  5  10  13
```

```
[2,]  9  11  14
```

```
[3,]  3  12  15
```

Matrices: Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. A Matrix is created using the `matrix()` function. Example: `matrix(data, nrow, ncol, byrow, dimnames)` where,

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.

```
>Mat <- matrix(c(1:16), nrow = 4, ncol = 4 )
```

```
>Mat
```

Output :

```
[,1] [,2] [,3] [,4]
[1,]  1   5   9  13
[2,]  2   6  10  14
[3,]  3   7  11  15
[4,]  4   8  12  16
```

Factors: 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 data analysis for statistical modeling.

```
>data <- c("East","West","East","North","North","East","West","West","East")
```

```
>factor_data <- factor(data)
```

```
>factor_data
```

Output :

```
[1] East West East North North East West West East
Levels: East North West
```

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.

```
>std_id = c (1:5)

>std_name = c("Rick","Dan","Michelle","Ryan","Gary")

>marks = c(623.3,515.2,611.0,729.0,843.25)

>std.data <- data.frame(std_id, std_name, marks)

>std.data
```

Output :

	std_id	std_name	marks
1	1	Rick	623.30
2	2	Dan	515.20
3	3	Michelle	611.00
4	4	Ryan	729.00
5	5	Gary	843.25

By this, we come to the end of different data types in R. Next, let us move forward in R Tutorial blog and understand another key concept – flow control statements.

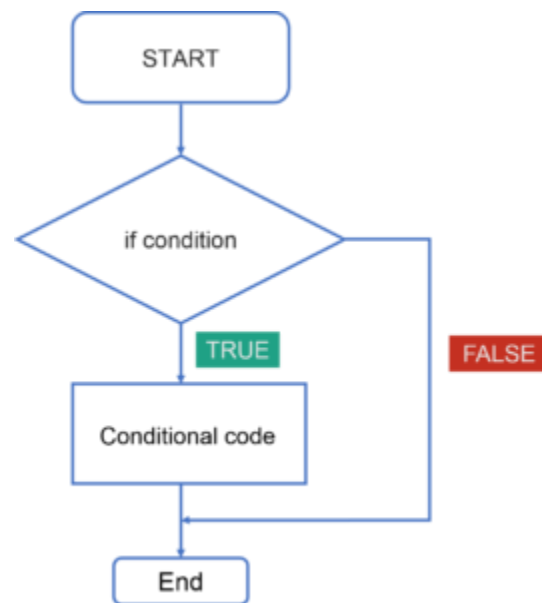
R Tutorial: Flow Control Statements

Flow control statements play a very important role as they allow you to control the flow of execution of a script inside a function. The most commonly used flow control statements are represented in the below image:

Now, let us discuss each one of them with examples.

R Tutorial: Selector Statements

- **If control Statement:** This control statement evaluates a single condition. It is quite easy as it just has a single keyword “if” followed by the condition and then certain set of statements that needs to get executed in case it is true. Refer to the below flowchart to get a better understanding:



In this flowchart, the code will respond in the following way:

1. First of all, it will enter the loop where it checks the condition.
2. If the condition is true, conditional code or the statements written will be executed.
3. If the condition is false, the statements gets ignored.

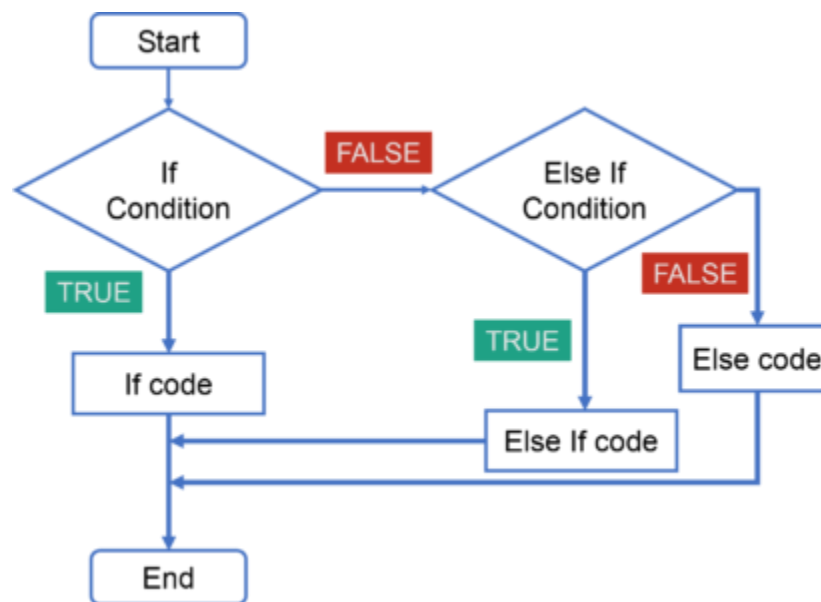
Below is an example of *if* control statement in R. Try running this example in R Studio.

```
1 x=2
2 repeat {
3   x= x^2
4   print(x)
5   if(x>100) {
6     break
7   }
```

Output :

```
[1] 4  
[1] 16  
[1] 256
```

- **If Else Control Statement** : This type of control statement evaluates a group of conditions and selects the statements. Refer to the below flowchart to get a better understanding:



In this flowchart, the code will respond in the following way:

1. First of all, it will enter the loop where it checks the condition.
2. If the condition is true, the first 'if' statements will get executed.
3. If the condition is false, then it goes to 'else if' condition and if it is true, the 'else if' code will be executed.
4. Finally, if the 'else if' code is also false, then it will go to 'else' code and it gets executed. This means if none of these conditions are true, then the 'else' statement gets executed.

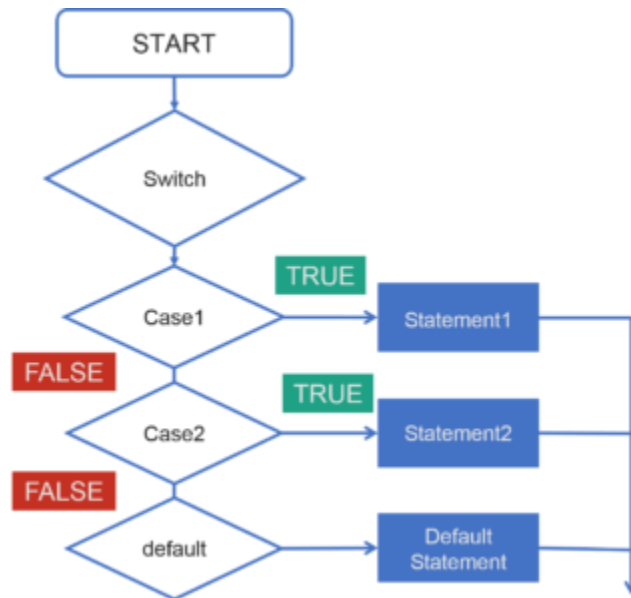
Below is an example of *if else* control statement in R. Try running this example in R Studio.

```
1  
2   x<-5  
3   if(x>5) {  
4     print("x is greater than 5")  
5   }  
6   elseif(x==5) {  
7     print("x is equal to 5")  
8   }  
9   else {  
10    print("x is not greater than 5")  
11  }
```

Output:

```
[1] "x is equal to 5"
```

Switch Statements: These control statements are basically used to compare a certain expression to a known value. Refer to the below flowchart to get a better understanding:



In this Switch case flowchart, the code will respond in the following steps:

1. First of all it will enter the switch case which has an expression.
2. Next it will go to Case 1 condition, checks the value passed to the condition. If it is true, Statement block will execute. After that, it will break from that switch case.
3. In case it is false, then it will switch to the next case. If Case 2 condition is true, it will execute the statement and break from that case, else it will again jump to the next case.
4. Now let's say you have not specified any case or there is some wrong input from the user, then it will go to the default case where it will print your default statement.

Below is an example of switch statement in R. Try running this example in R Studio.

```
1  vtr <- c(150,200,250,300,350,400)
2  option <- "mean"
3  switch(option,
4    "mean" = print(mean(vtr)),
5    "mode" = print(mode(vtr)),
6    "median" = print(median(vtr))
7  )
```

Output :

```
[1] 275
```

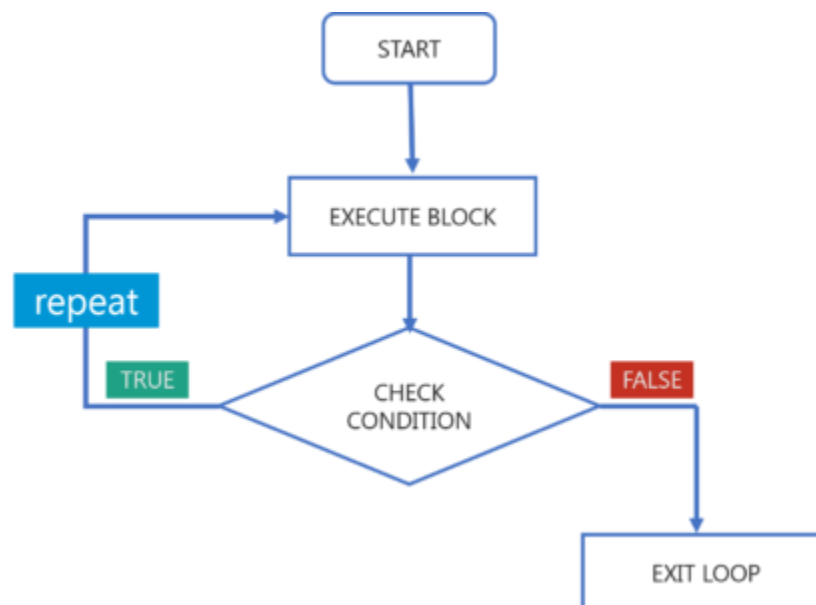
R Tutorial : Loop Statements

Loops help you to repeat certain set of actions so that you don't have to perform them repeatedly. Imagine you need to perform an operation 10 times, if you start writing the code for each time, the length of the program increases and it would be difficult for you to understand it later. But at the same time by using a loop, if I write the same statement inside a loop, it saves time and makes easier for code readability. It also gets more optimized with respect to code efficiency.

In the above image, '*repeat*' and '*while*' statements help you to execute a certain set of rules until the condition is true but '*for*' is a loop statement that is used when you know how many times you want to repeat a block of statement. Now, if you know that you want to repeat it for 10 times, then you will go with '*for*' statement but if you are not sure about how many times you want the code to be repeated, you will go with '*repeat*' or '*while*' loop.

Let's discuss each one of them with examples.

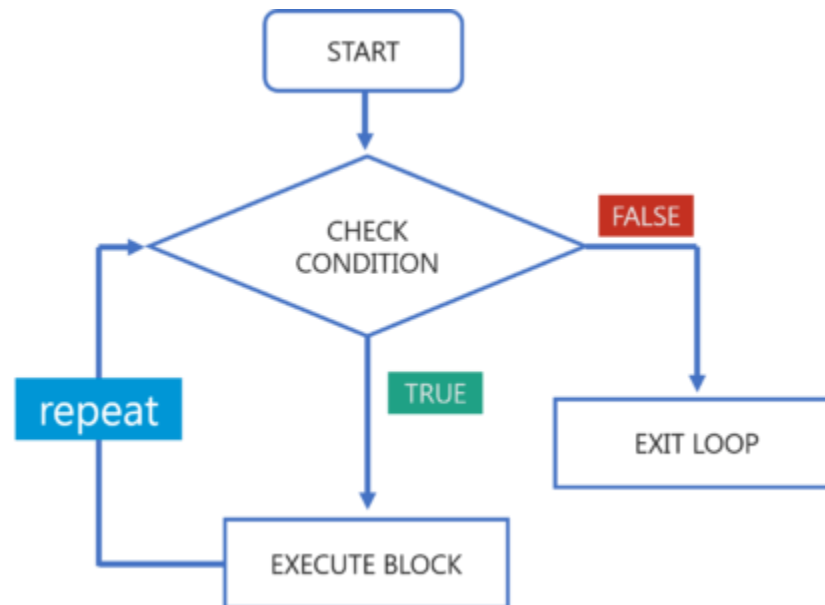
- **Repeat:** The repeat loop helps to execute the same set of code again and again until a stop condition is met. Refer to the below flowchart to get a better understanding:



In the above flowchart, the code will respond in the following steps:

1. First of all it will enter and execute a set of code.
2. Next it will check the condition, if it is true it will go back and execute the same set of code again until it is meant to be false.
3. If it is found to be false, it will directly exit the loop.

- **While:** The while statement also helps to execute the same set of code again and again until a stop condition is met. Refer to the below flowchart to get a better understanding:



In the above flowchart, the code will respond in the following steps:

1. First of all it will check the condition.
2. If it is found to be true, it will execute the set of code.
3. Next, it again checks the condition, if its true it will execute the same code again.
As soon as the condition is found to be false, it immediately exits the loop.

Below is an example of while statement in R. Try running this example in R Studio.

```
1 x=2
2 while(x<1000)
3 {
4 x=x^2
5 print(x)
6 }
```

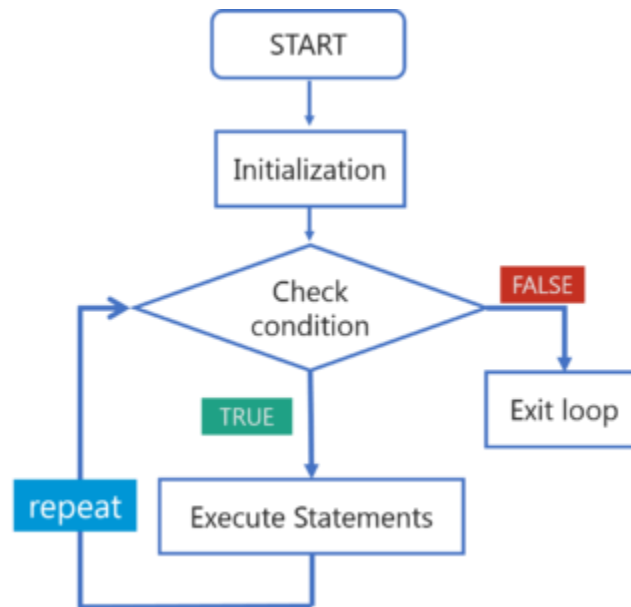
Output:

```
4
16
256
65536
```

So you must be wondering how these two statements differ? Let me clear your doubt! Here the major difference between the repeat and while statement is that it changes with respect to your condition. *While* loop basically defines when you are going to enter the loop

to execute the statements and *repeatloop* defines when you leave from the loop after the execution of the statements. So these two statements are known as entry control loop and exit control loop. That's how while and repeat statements are different.

- **For Loop:** For loops are used when you need to execute a block of code several number of times. Refer to the below flowchart to get a better understanding:



In the above flowchart, the code will respond in the following steps:

1. First of all there is initialization where you specify how many times you want the loop to repeat.
2. Next, it checks the condition. If the condition is true, it will execute the set of code for the specified number of times.
3. As soon as the condition is found to be false, it immediately exits the loop.

Below is an example of for statement in R. Try running this example in R Studio.

```
1 vtr <- c(7,19,25,65, 45)
2 for( i in vtr) {
3   print(i)
4 }
```

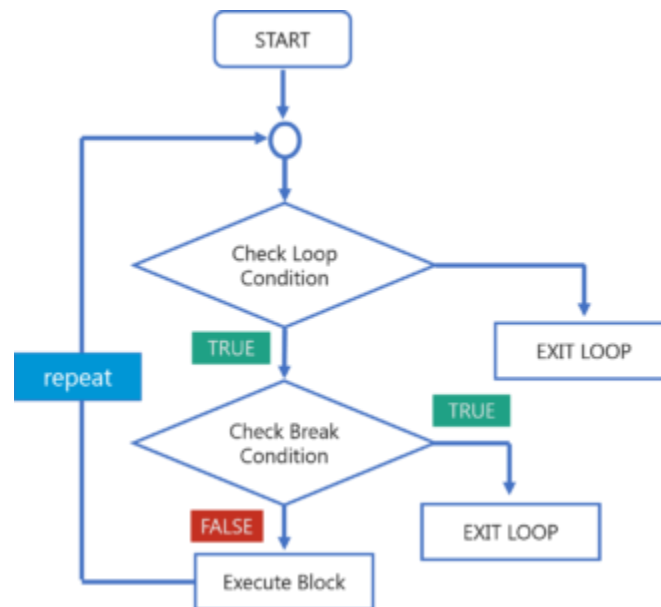
Output :

```
7
19
25
65
45
```

Next, let us move to our last set of statements in R Tutorial blog, i.e jump statements.

R Tutorial : Jump Statements

Break Statement: Break statements help to terminate the program and resumes the control to the next statement following the loop. These statements are also used in switch case. Refer to the below flowchart to get a better understanding:



In the above flowchart, the code will respond in the following steps:

1. First of all, it will enter the loop where it checks the condition.
2. If the loop condition is false, it directly exits the loop.
3. If the condition is true, it will then check the break condition.
4. If break condition is true, it exists from the loop.
5. If the break condition is false, then it will execute the statements that are remaining in the loop and then repeat the same steps.

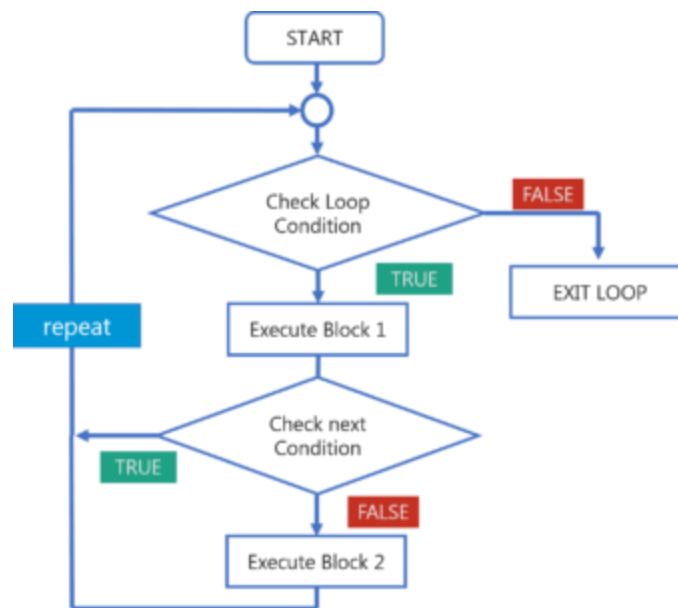
Below is an example of jump statement in R. Try running this example in R Studio.

```
1 x <- 1:5
2 for (val in x) {
3   if (val == 3){
4     break
5   }
6   print(val)
7 }
```

Output:

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

Next Statement: A next statement is used when you want to skip the current iteration of the loop without terminating it. Next statement is quite similar to ‘continue’ in other programming language. Refer to the below flowchart to get a better understanding:



In the above flowchart, the code will respond in the following steps:

1. First of all, it will enter the loop where it checks the condition.
2. If the loop condition is false, it directly exits the loop.
3. If the loop condition is true, it will execute block 1 statements.
4. After that it will check for ‘next’ statement. If it is present, then the statements after that will not be executed in the same iteration of the loop.
5. If ‘next’ statement is not present, then all the statements after that will be executed.

Below is an example of next statement in R. Try running this example in R Studio.

```
1  for(i in 1:15)
2  {
3    if((i%%2)==0) {
4      next
5    }
6    print(i)
7  }
```

Output :

```
1
3
5
7
```

9

11

13

15

LAB PROGRAMS

Exercises on - Basics of R:

1. Write a program in R to find the perfect numbers between 1 and 500.

The perfect numbers between 1 to 500 are:

6

28

496

2. Write a program in R to check whether a number is prime or not.

Sample Output:

Input a number to check prime or not: 13

The entered number is a prime number.

3. Write a program in R to find prime number within a range.

Input number for starting range: 1

Input number for ending range: 100

The prime numbers between 1 and 100 are:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

The total number of prime numbers between 1 to 100 is: 25

4. Write a program in R to find the factorial of a number.

Sample output:

Input a number to find the factorial: 5

The factorial of the given number is: 120

5. Write a program in R to find the Greatest Common Divisor (GCD) of two numbers.

Sample Output:

Input the first number: 25

Input the second number: 15

The Greatest Common Divisor is: 5

6. Write a program in R to find the sum of digits of a given number.

Sample Output:

Input a number: 1234

The sum of digits of 1234 is: 10

7. Write a program in R to list non-prime numbers from 1 to an upperbound.

Sample Output:

Input the upperlimit: 25

The non-prime numbers are:

4 6 8 9 10 12 14 15 16 18 20 21 22 24 25

8. Write a program in R to print a square pattern with # character.

Sample Output:

Print a pattern like square with # character:

```
-----  
Input the number of characters for a side: 4  
# # # #  
# # # #  
# # # #  
# # # #
```

9. Write a program in R to display the cube of the number upto given an integer.

Sample Output:

```
Input the number of terms : 5  
Number is : 1 and the cube of 1 is: 1  
Number is : 2 and the cube of 2 is: 8  
Number is : 3 and the cube of 3 is: 27  
Number is : 4 and the cube of 4 is: 64  
Number is : 5 and the cube of 5 is: 125
```

10. Write a program in R to display the first n terms of Fibonacci series.

Sample Output:

```
Input number of terms to display: 10  
Here is the Fibonacci series upto to 10 terms:  
0 1 1 2 3 5 8 13 21 34
```

11. Write a program in R to display the number in reverse order.

Sample Output:

```
Input a number: 12345  
The number in reverse order is : 54321
```

12. Write a program in R to find out the sum of an A.P. series.

Sample Output:

```
Input the starting number of the A.P. series: 1  
Input the number of items for the A.P. series: 8  
Input the common difference of A.P. series: 5  
The Sum of the A.P. series are :  
1 + 6 + 11 + 16 + 21 + 26 + 31 + 36 = 148
```

13. Write a program in R to Check Whether a Number can be Express as Sum of Two Prime Numbers.

Sample Output:

```
Input a positive integer: 20  
20 = 3 + 17  
20 = 7 + 13
```

14. Write a program in R to find the length of a string without using the library function.

Sample Output:

Input a string: w3resource.com

The string contains 14 number of characters.

So, the length of the string w3resource.com is:14

15. Write a program in R to display the pattern like right angle triangle using an asterisk.

Sample Output:

Input number of rows: 5

*

**

16. Write a program in R to display the pattern like right angle triangle with number.

Sample Output:

Input number of rows: 5

1

12

123

1234

12345

17. Write a program in R to make such a pattern like right angle triangle using number which will repeat the number for that row.

Sample Output:

Input number of rows: 5

1

22

333

4444

55555

18. Write a program in R to make such a pattern like right angle triangle with number increased by 1.

Sample Output:

Input number of rows: 4

1

2 3

4 5 6

7 8 9 10

19. Write a program in R to find the sum of first and last digit of a number.

Sample Output:

Input any number: 12345

The first digit of 12345 is: 1

The last digit of 12345 is: 5

The sum of first and last digit of 12345 is: 6

20. Write a program in R to find the frequency of each digit in a given integer.

Sample Output:

Input any number: 122345

The frequency of 0 = 0

The frequency of 1 = 1

The frequency of 2 = 2

The frequency of 3 = 1

The frequency of 4 = 1

The frequency of 5 = 1

The frequency of 6 = 0

The frequency of 7 = 0

The frequency of 8 = 0

The frequency of 9 = 0

21. Write a program in R to display the given number in words.

Sample Output:

Input any number: 8309

Eight Three Zero Nine

22. Write a program in R to enter any number and print all factors of the number.

Sample Output:

Input a number: 63

The factors are: 1 3 7 9 21 63

23. Write a program in R to find one's complement of a binary number.

Sample Output:

Input a 8 bit binary value: 10100101

The original binary = 10100101

After ones complement the number = 01011010

24. Write a program in R to find two's complement of a binary number.

Sample Output:

Input a 8 bit binary value: 01101110

The original binary = 01101110

After ones complement the value = 10010001

After twos complement the value = 10010010

25. Write a program in R to convert a decimal number to binary number.

Sample Output:

Input a decimal number: 35

The binary number is: 100011

26. Write a program in R to convert a decimal number to hexadecimal number.

Sample Output:

Input a decimal number: 43

The hexadecimal number is : 2B

27. Write a program in R to convert a decimal number to octal number.

Sample Output:

Input a decimal number: 15

The octal number is: 17

28. Write a program in R to convert a binary number to decimal number.

Sample Output:

Input a binary number: 1011

The decimal number: 11

29. Write a program in R to convert a binary number to hexadecimal number.

Sample Output:

Input a binary number: 1011

The hexadecimal value: B

30. Write a program in R to convert a binary number to hexadecimal number.

Sample Output:

Input a binary number: 1011

The equivalent octal value of 1011 is : 13

Data Types in R:

Exercises with Answers:

1. How many numbers are there in the vector x? 

2. `length(x)`

3.

4. `[1] 3`

5. How many numbers will `x + y` generate? 

6. `length(x + y)`

7.

8. `[1] 3`

9. What is the sum of all values in x? 

10. `sum(x)`

11.

12. `[1] 13`

13. What is the sum of y times y? 

14. `sum(y*y)`

15.

16. `[1] 147`

17. What do you get if you add x and y? 

18. `x + y`

19.

20. `[1] 3 9 18`

21. Assign x times 2 to a new vector named z 


```
22. z <- x * 2
```

23. How many numbers will z have, why? 

```
24. length(z)
```

```
25.
```

```
26. [1] 3
```

27. Assign the mean of z to a new vector named z.mean and determine the length of z.mean 

```
28. z.mean <- mean(z)
```

```
29. length(z.mean)
```

```
30.
```

```
31. [1] 1
```

32. Create a numeric vector with all integers from 5 to 107 

```
33. vec.tmp <- 5:107
```

```
34. vec.tmp
```

```
35.
```

```
36. [1] 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
```

```
37. [19] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

```
38. [37] 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
```

```
39. [55] 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76
```

```
40. [73] 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
```

```
41. [91] 95 96 97 98 99 100 101 102 103 104 105 106 107
```

42. Create a numeric vector with the same length as the previous one, but only containing the number 3

```
43. vec.tmp2 <- rep(3, length(vec.tmp))
```

44. vec.tmp2

45.

46.[1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

47.[38] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

48. [75] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

49. Create a vector that contain all numbers from 1 to 17, where each number occurs the the same number of times as the number itself eg. 1, 2, 2, 3, 3, 3...🔑

50. What will be the result of the following calculations?

- $c(1, 3, 5) + c(2, 4, 6)$
- $c(1, 3, 5) + c(2, 4, 6, 8)$
- $c(1, 3) - c(2, 4, 6, 8)$

Try to think about your expectations prior to running the code in R.

51. Create two numeric vectors of length 4 and test run all the basic operators (as seen in the table earlier) with these two as arguments. Make sure you understand the output generated by R.

Modify and subset vectors

Create a new character vector that the following words and save it using a suitable name:
apple, banana, orange, kiwi, potato



```
veggies <- c("apple", "banana", "orange", "kiwi", "potato")
```

Do the following on your newly created vector.

Select orange from the vector

```
veggies[3]
```

```
[1] "orange"
```

Select all fruits from the vector 

```
veggies[-5]
```

```
veggies[1:4]
```

```
[1] "apple" "banana" "orange" "kiwi"
```

```
[1] "apple" "banana" "orange" "kiwi"
```

Do the same selection as in question 2 without using index positions 

```
veggies[veggies=="apple" | veggies == "banana" | veggies == "orange" | veggies == "kiwi"]
```

```
veggies[veggies!="potato"]
```

```
[1] "apple" "banana" "orange" "kiwi"
```

```
[1] "apple" "banana" "orange" "kiwi"
```


Convert the character string to a numeric vector 

```
as.numeric(veggies)
```

```
[1] NA NA NA NA NA
```

Warning message:

NAs introduced by coercion

Create a vector of logic values that can be used to extract every second value from your character vector 

```
selection <- c(FALSE, TRUE, FALSE, TRUE, FALSE)
```

```
veggies[selection]
```

```
[1] "banana" "kiwi"
```

🔑 Alternative solution, why do this work?

```
selection2 <- c(FALSE, TRUE)
veggies[selection2]
```

```
[1] "banana" "kiwi"
```

Add the names a, b, o, k and p to the vector 🔑

```
names(veggies) <- c("a", "b", "o", "k", "p")
```

Create a vector containing all the letters in the alphabet (NB! this can be done without having to type all letters). Google is your friend 🔑

```
letters
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

Sample 30 values randomly with replacement from your letter vector and convert the character vector to factors. Which of the levels have most entries in the vector? 🔑


```
letter.sample <- sample(letters, size = 30, replace = TRUE)
letter.sample <- factor(letter.sample)
summary(letter.sample)
```

```
a b c e g k l m n o q r t v w x z
3 1 2 1 3 1 1 1 3 1 2 2 1 3 2 1 2
```

Extract the letter 14 to 19 from the created vector 

```
letters[14:19]
```

```
[1] "n" "o" "p" "q" "r" "s"
```

Extract all but the last letter 

```
letters[1:length(letters)-1]
```

```
letters[-length(letters)]
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
```

```
[20] "t" "u" "v" "w" "x" "y"
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
```

```
[20] "t" "u" "v" "w" "x" "y"
```

Which is the index position of the letter u in the vector? 


```
which(letters=="u")
```

```
[1] 21
```

Create a new vector of length one that holds all the alphabet a single entry 

```
paste(letters, sep = "", collapse = "")
```

```
[1] "abcdefghijklmnopqrstuvwxyz"
```

Create a numeric vector by sampling 100 numbers from a normal distribution with mean 2 and standard deviation 4. Hint! Check the function `rnorm()` 


```
norm.rand <- rnorm(100, mean = 2, sd = 4)
```

How many of the generated values are negative? 

```
length(norm.rand[norm.rand<0])  
[1] 23
```

Calculate the standard deviation, mean, median of your random numbers 

```
sd(norm.rand)  
mean(norm.rand)  
median(norm.rand)  
  
[1] 3.541989  
[1] 1.910667  
[1] 1.631083
```

Replace the 11th value in your random number vector with NA and calculate the same summary statistics again 

```
norm.rand[11] <- NA  
sd(norm.rand, na.rm = TRUE)  
mean(norm.rand, na.rm = TRUE)  
median(norm.rand, na.rm = TRUE)  
  
[1] 3.553763  
[1] 1.889685  
[1] 1.62893
```


Replace the last position in the vector with the letter L and calculate the same summary statistics. 

```
norm.rand[100] <- "L"  
sd(norm.rand, na.rm = TRUE)  
mean(norm.rand, na.rm = TRUE)  
median(norm.rand, na.rm = TRUE)
```

Warning message:

In var(if (is.vector(x) || is.factor(x)) x else as.double(x), na.rm = na.rm) :

NAs introduced by coercion

[1] NA

Warning message:

In mean.default(norm.rand, na.rm = TRUE) :

argument is not numeric or logical: returning NA

[1] NA

Warning message:

In mean.default(sort(x, partial = half + 0L:1L)[half + 0L:1L]) :

argument is not numeric or logical: returning NA

Create two numeric vectors of length 4 and 5 respectively and test run all the basic operators (as seen in the following table) with these vectors two as arguments. Make sure you understand the output generated by R

Operation	Description	Sample Example on one element of a vector	
		Example	Example Result
$x + y$	Addition	$1 + 3$	4
$x - y$	Subtraction	$1 - 3$	-2
$x * y$	Multiplication	$2 * 3$	6
x / y	Division	$1 / 2$	0.5
$x ^ y$	Exponent	$2 ^ 2$	4
$x \% \% y$	Modular arithmetic	$1 \% \% 2$	1
$x \% / \% y$	Integer division	$1 \% / \% 2$	0
$x == y$	Test for equality	$1 == 1$	TRUE
$x <= y$	Test less or equal	$1 <= 1$	TRUE
$x >= y$	Test for greater or equal	$1 >= 2$	FALSE
$x \&\& y$	Boolean AND for scalar	$3 >= 2 \&\& 3 < 10$	TRUE
$x \& y$	The same for vectors		
$x y$	Boolean OR for scalar	$1 >= 2 3 < 10$	TRUE
$x y$	The same for vectors		
$!x$	Boolean not	$1 != 2$	TRUE

1. How many numbers are there in the vector x?
2. How many numbers will $x + y$ generate?
3. What is the sum of all values in x?
4. What is the sum of y times y?
5. What do you get if you add x and y?
6. Assign x times 2 to a new vector named z
7. How many numbers will z have, why?
8. Assign the mean of z to a new vector named z.mean and determine the length of z.mean

9. Create a numeric vector with all integers from 5 to 107
10. Create a numeric vector with the same length as the previous one, but only containing the number 3
11. Create a vector that contain all numbers from 1 to 17, where each number occurs the the same number of times as the number itself eg. 1, 2, 2, 3, 3, 3...

Consider a vector:

`x <- c(4,6,5,7,10,9,4,15)`. What is the value of: `x < 7`

Consider two vectors: `p <- c(3, 5, 6, 8)` and `q <- c(3, 3, 3)`. What is the value of: `p+q`

What is the output for the following statements?

`c(1, 3, 5) + c(2, 4, 6)`

`c(1, 3, 5) + c(2, 4, 6, 8)`

`c(1, 3) - c(2, 4, 6, 8)`

Consider two vectors, `x`, `y`

`X <- c(4,6,5,7,10,9,4,15)`

`Y <- c(0,10,1,8,2,3,4,1)`. What is the value of: `x*y`

If `z <- 0:9` then what is the output from the following R-statements:

`digits <- as.character(z)`

`as.integer(digits)`

Consider the vector: `x <- c(1,2,3,4)`

What is the value of `k` for: `(x+2)[(!is.na(x)) & x > 0] -> k`

If `x <- c(2, 4, 6, 8)` and `y <- c(TRUE, TRUE, FALSE, TRUE)`

What is the value of: `sum(x[y])`

Consider the vector: `x <- c(34, 56, 55, 87, NA, 4, 77, NA, 21, NA, 39)`

What is the value of: `is.na(x)`

Which R-statement will find the locations of NA values in `x`?

Which R-statement will count the number of NA values in `x`?

Consider two vectors, `a`, `b`

`a <- c(1,2,4,5,6)`

`b <- c(3,2,4,1,9)`. What is the value of: `cbind(a,b)`, `rbind(a,b)`, `append(a,b)`, `cat(a,b)`

Consider two vectors, `a`, `b`

`a = c(1,5,4,3,6)`

`b = c(3,5,2,1,9)`

What is the value of: `a <= b`

If `x=c(1:12)`

What is the value of: `dim(x)`

What is the value of: `length(x)`

If `a=c(12:5)`

What is the value of: `is.numeric(a)`

Consider two vectors, `x`, `y`

`x=letters[1:10]`

`y=letters[15:24]`

What is the value of: `x<y`

If `x=c('blue','red','green','yellow')`. What is the value of: `is.character(x)`

If `x=c('blue',10,'green',20)`. What is the value of: `is.character(x)`

If `x <- c(a = 1, b = 2,c=3,d=4)`

What is the output for the code:

`seq(5,11,along.with =x)`

If `x= seq(4,12,4)` ,

what is the output for the code:

`rep(x,each=2)`

What is the output for the code:

`seq(5,11,by=2,length.out=3)`

What is the output for the code:

`rep(letters[1:10],3)`

Create a sequence with values:

100 95 90 85 80 75 70 65 60 55 50

What is the output for the code:

`seq(10,0,by=5)`

What is the output for the code:

`seq(2,10,by=4)==c(2,6,10)`

What is the output for the code:

`rep(c('seq','rep'),each=4)`

Consider two variables, `A` and `B`,

`A= as.Date("2016-11-01") B = as.Date("2016-11-15")`

What is the output for the code:

```
seq.Date(A,B, by = "1 day")
```

Exercise 10

Consider two variables, C and D,

```
C= as.Date("2016-02-01")
```

```
D = as.Date("2016-06-15")
```

What is the output for the code:

```
seq.Date(D,C, by = "-1 month")
```

If:

```
Age <- c(22, 25, 18, 20)
```

```
Name <- c("James", "Mathew", "Olivia", "Stella")
```

```
Gender <- c("M", "M", "F", "F")
```

then what is the R-code for getting the following output;

```
## Age Name Gender
## 1 22 James    M
## 2 25 Mathew   M
```

Matrices

If `M=matrix(c(1:10),nrow=5,ncol=2, dimnames=list(c('a','b','c','d','e'),c('A','B')))`

What is the value of: M

Consider the matrix M,

What is the value of:

```
M[1,]
```

```
M[,1]
```

```
M[3,2]
```

```
M['e','A']
```

Consider the matrix N

```
N=matrix(c(1:9),nrow=3,ncol=3, dimnames=list(c('a','b','c'),c('A','B','C')))
```

What is the value of: `diag(N)`

What is the value of: `diag(4,3,3)`

If `M=matrix(c(1:9),3,3,byrow=T, dimnames=list(c('a','b','c'),c('d','e','f')))`

What is the value of:

```
rownames(M)
```

```
colnames(M)
```

What is the value of:

`upper.tri(M)`

`lower.tri(M)`

`lower.tri(M,diag=T)`

Consider two matrix, M,N

`M=matrix(c(1:9),3,3,byrow=T)`

`N=matrix(c(1:9),3,3)`

What is the value of: `M*N`

Consider two matrix, M,N

`M=matrix(c(1:9),3,3,byrow=T)`

`N=matrix(c(1:9),3,3)`

What is the value of: `M%*%N`

Consider two matrix, M,N

`M=matrix(c(1:9),3,3,byrow=T)`

`N=matrix(c(1:9),3,3)`

What is the value of: `(M+N)^2`

Consider two matrix, M,N

`M=matrix(c(1:9),3,3,byrow=T)`

`N=matrix(c(1:9),3,3)`

What is the value of: `M/N`

1. Consider `A=matrix(c(2,0,1,3), ncol=2)` and `B=matrix(c(5,2,4,-1), ncol=2)`.
 - a) Find `A + B`
 - b) Find `A - B`
2. Scalar multiplication. Find the solution for `a*A` where `a=3` and A is the same as in the previous question.
3. Using the `diag` function build a diagonal matrix of size 4 with the following values in the diagonal 4,1,2,3.
4. Find the solution for `Ab`, where A is the same as in the previous question and `b<-c(7,4)`.
5. Find the solution for `AB`, where B is the same as in question 1.
6. Find the transpose matrix of A.

7. Find the inverse matrix of A.
8. Find the value of x on $Ax=b$.
9. Using the function eigen find the eigenvalue for A.
10. Find the eigenvalues and eigenvectors of $A'A$. Hint: Use crossprod to compute $A'A$.

Answers

1. Ans:

```
A <- matrix(c(2,0,1,3), ncol=2)
B <- matrix(c(5,2,4,-1),ncol=2)
```

A+B

```
##      [,1] [,2]
## [1,]   7   5
## [2,]   2   2
```

A-B

```
##      [,1] [,2]
## [1,]  -3  -3
## [2,]  -2   4
```

2. Ans:

```
a <- 3
```

a*A

```
##      [,1] [,2]
## [1,]   6   3
## [2,]   0   9
```

3. Ans:

```
diag(4)*c(4,1,2,3)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   4   0   0   0
## [2,]   0   1   0   0
## [3,]   0   0   2   0
## [4,]   0   0   0   3
```

4. Ans:

```
b <- c(7,4)
b%%A
##      [,1] [,2]
## [1,]  14  19
```

5. Ans

```
A%%B
##      [,1] [,2]
## [1,]  12   7
## [2,]   6  -3
```

6. Ans:

```
t(A)
##      [,1] [,2]
## [1,]   2   0
## [2,]   1   3
```

7. Ans:

```
solve(A)
##      [,1] [,2]
## [1,]  0.5 -0.1666667
## [2,]  0.0  0.3333333
```

8. Ans:

```
solve(A,b)
## [1] 2.833333 1.333333
```

9. Ans:

```
eigen(A)$values
## [1] 3 2
```

10. Ans:

```
eigen(crossprod(A))
## $values
## [1] 10.605551 3.394449
##
## $vectors
##      [,1] [,2]
## [1,] 0.2897841 -0.9570920
## [2,] 0.9570920 0.2897841
```

DATA FRAMES

Write the code to create a Data Frame with name *df*

```
id  = c(5, 6, 7, 8, 9),  
prod = c("F", "H", "B", "S", "D"),  
units = c(12, 19, 44, 26, 43)
```

How to access the values of attribute *id*

Which of the following is a categorical variable?

- a) Weight
- b) Volume
- c) Distance
- d) Weekday

Functions in R:

What is a Function in R?

A function, in a programming environment, is a set of instructions. A programmer builds a function to avoid repeating the same task, or reduce complexity.

A function should be

- written to carry out a specified a tasks
- may or may not include arguments
- contain a body
- may or may not return one or more values.

A general approach to a function is to use the argument part as inputs, feed the body part and finally return an output. The Syntax of a function is the following:

```
function (arglist) {  
  #Function body  
}
```

In this tutorial, we will learn

- R important built-in functions
- General functions
- Math functions
- Statistical functions
- Write function in R
- When should we write function?
- Functions with condition

R important built-in functions:

There are a lot of built-in function in R. R matches your input parameters with its function arguments, either by value or by position, then executes the function body. Function arguments can have default values: if you do not specify these arguments, R will take the default value.

Note: It is possible to see the source code of a function by running the name of the function itself in the console.

We will see three groups of function in action

- General function
- Maths function
- Statistical function

General functions

We are already familiar with general functions like `cbind()`, `rbind()`, `range()`, `sort()`, `order()` functions. Each of these functions has a specific task, takes arguments to return an output. Following are important functions one must know-

Math functions

R has an array of mathematical functions.

Operator	Description
<code>abs(x)</code>	Takes the absolute value of x
<code>log(x,base=y)</code>	Takes the logarithm of x with base y; if base is not specified, returns the natural logarithm
<code>exp(x)</code>	Returns the exponential of x
<code>sqrt(x)</code>	Returns the square root of x
<code>factorial(x)</code>	Returns the factorial of x (x!)

Examples:

sequence of number from 44 to 55 both including incremented by 1

```
x_vector <- seq(45,55, by = 1)
```

#logarithm

```
log(x_vector)
```

Output:

```
## [1] 3.806662 3.828641 3.850148 3.871201 3.891820 3.912023 3.931826
```

```
## [8] 3.951244 3.970292 3.988984 4.007333
```

#exponential

```
exp(x_vector)
```

#squared root

```
sqrt(x_vector)
```

Output:

```
## [1] 6.708204 6.782330 6.855655 6.928203 7.000000 7.071068 7.141428
```

```
## [8] 7.211103 7.280110 7.348469 7.416198
```

#factorial

```
factorial(x_vector)
```

Output:

```
## [1] 1.196222e+56 5.502622e+57 2.586232e+59 1.241392e+61 6.082819e+62
```

```
## [6] 3.041409e+64 1.551119e+66 8.065818e+67 4.274883e+69 2.308437e+71
```

```
## [11] 1.269640e+73
```

Statistical functions

R standard installation contains wide range of statistical functions. In this tutorial, we will briefly look at the most important function..

Basic statistic functions

Operator	Description
mean(x)	Mean of x
median(x)	Median of x
var(x)	Variance of x
sd(x)	Standard deviation of x
scale(x)	Standard scores (z-scores) of x
quantile(x)	The quartiles of x
summary(x)	Summary of x: mean, min, max etc..

Examples:

```
speed <- dt$speed
```

```
speed
```

```
# Mean speed of cars dataset
```

```
mean(speed)
```

```
Output:
```

```
## [1] 15.4
```

```
# Median speed of cars dataset
```

```
median(speed)
```

```
Output:
```

```
## [1] 15
```

```
# Variance speed of cars dataset
```

```
var(speed)
```

```
Output:
```

```
## [1] 27.95918
```

```
# Standard deviation speed of cars dataset
```

```
sd(speed)
```

Output:

```
## [1] 5.287644
```

```
# Standardize vector speed of cars dataset
```

```
head(scale(speed), 5)
```

Output:

```
##      [,1]
```

```
## [1,] -2.155969
```

```
## [2,] -2.155969
```

```
## [3,] -1.588609
```

```
## [4,] -1.588609
```

```
## [5,] -1.399489
```

```
# Quantile speed of cars dataset
```

```
quantile(speed)
```

Output:

```
##  0%  25%  50%  75% 100%
```

```
##   4   12   15   19   25
```

```
# Summary speed of cars dataset
```

```
summary(speed)
```

Output:

```
##  Min. 1st Qu.  Median   Mean 3rd Qu.   Max.
```

```
##   4.0   12.0   15.0   15.4   19.0   25.0
```

Up to this point, we have learned a lot of R built-in functions.

Note: Be careful with the class of the argument, i.e. numeric, Boolean or string. For instance, if we need to pass a string value, we need to enclose the string in quotation mark: "ABC" .

Data Sorting:

----- DISPLAYING AND SAMPLING -----

- Open Rstudio/Rconsole
- In console, Print the dataset mtcars
- Print the structure of the dataset
- What is the datatype of the dataset?
- How many columns and rows are there in the dataset??
- What information (structure summary) you will get from str() function?
- Print the row names
- Print the column names
- Print the number of columns in mtcars (Hint: Use function - ncol)
- Print the number of rows (Hint: Use function - nrow)
- Print all the elements of 2nd row
- Print all the elements of 2nd, 5th and 13th row
- Print the elements of rows from 15 to 20
- Print the elements of rows from 13 to 24, 28 and 30
- Print all odd indexed rows (rows 1,3,5,...) (Hint: Use function - seq)
- Print all even indexed rows (rows 2,4,6,...)
- Print every 3rd row from 1st row (1,4,7,10..)
- Print first row and last row (Hint: Use function - ncol)

- Print last 3 rows without using tail() function
- Print the elements of 3rd column
- Print the elements of column with name "wt"
- Print the elements of columns "mpg" and "qsec"
- Print first three columns
- Print the elements of columns from 5 to 10
- Print the elements of columns from 3 to 7, 9 and 11
- Print all odd indexed columns (1,3,5,...)
- Print all even indexed columns (2,4,6,...)
- Print every 3rd column from 1st column (1,4,7,10..)
- Print first column and last column
- Print last 3 columns
- Print first Row and 2nd and third column
- Print First, Second Row and Second and Third Column
- Print element at 2nd row, third column
- Print all the rows having "mpg" value greater than 14
- Print all the rows having "hp" value less than 100
- Print all the rows having "dis" value is between 100 and 200

ATTACH & DETACH FUNCTIONS

- find what attach() and detach() commands do???

HEAD & TAIL FUNCTIONS

- find what head() and tail() commands do
- Use head() and tail() commands to display sample observations of mtcars dataset
- Use head() command to Print first 10 observations
- Use tail() commands to Print last 15 observations

SORTING

- Sort the observations of the dataset “mtcars” in increasing order based on the values in the column "mpg"
- Sort the observations of the dataset “mtcars” in decreasing order based on the values in the column "cyl"
- Sort the observations of the dataset “mtcars” in increasing order based on the values in the columns both "mpg" and "cyl"
- Sort the observations of the dataset “mtcars” in decreasing order based on the values in the columns both "mpg" and "cyl"
- Sort the observations of the dataset “mtcars” by column “mpg” in increasing order and column “cyl” in decreasing order

Data Cleaning:

DETERMINE DUPLICATE ELEMENTS

Finding Duplicate Values:

duplicated() determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.

anyDuplicated() returns the index of the first duplicate value if any, otherwise 0.
anyDuplicated() is a “generalized” more efficient shortcut for *any(duplicated())*

EXERCISES

- Create a vector as `x <- c(9:20, 1:5, 3:7, 0:8)`
- Use *duplicated()* function to print the logical vector indicating the duplicate values present in x
- Observe the output of *duplicated(x, fromLast = TRUE)*
- What is the difference between *duplicated(x)* and *duplicated(x, fromLast=TRUE)*
- Extract duplicate elements from x
- Extract unique elements from x
- Print duplicate elements from x in different order (**Hint:** Use *duplicated(x, fromLast = TRUE)*)
- Extract unique elements from x in different order (**Hint:** Use *duplicated(x, fromLast = TRUE)*)
- Print the indices of duplicate elements
- Print the indices of unique elements
- How many unique elements are in x
- How many duplicate elements are in x
- Create a dataframe df :

```
a <- c(rep("A", 3), rep("B", 3), rep("C", 2))  
b <- c(1, 1, 2, 4, 1, 1, 2, 2)  
df <- data.frame(a, b)
```

- Use *duplicated()* function to print the logical vector indicating the duplicate values present in dataframe “df”
- Extract duplicate elements from dataframe “df”
- Extract unique elements from dataframe “df”
- Print the indices of duplicate elements
- Print the indices of unique elements
- How many unique elements are in dataframe "df"
- How many duplicate elements are in dataframe "df"

DATASET – INTRODUCTION

Fisher’s Iris Dataset



Iris Flowers

Description

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica.

Format

iris is a data frame with 150 cases (rows) and 5 variables (columns) named Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species

EXERCISES

- Print the dataset *iris*
- Print the structure of the dataset *iris*
- Print the summary of all the variables of the dataset *iris* (**Hint:** Use function *summary()*)
- How many of the variables (columns) are in the dataset *iris*
- How many observations (rows) are in the dataset *iris*
- Use *duplicated()* function to print the logical vector indicating the duplicate values present in the dataset *iris*
- Extract duplicate elements from the dataset *iris*
- Extract unique elements from the dataset *iris*
- Print the indices of duplicate elements in the dataset *iris*
- Print the indices of unique elements in the dataset *iris*
- How many unique elements are in the dataset *iris*
- How many duplicate elements are in the dataset *iris*

Missing Values:

- A missing value is one whose value is unknown.
- Missing values are represented in R by the NA symbol.
- NA is a special value whose properties are different from other values.
- NA is one of the very few reserved words in R: you cannot give anything this name.
- Missing values are often legitimate: values really are missing in real life.
- NAs can arise when you read in a Excel spreadsheet with empty cells, for example.
- You will also see NA when you try certain operations that are illegal or don't make sense.

Here are some examples of operations that produce NA's.

```
> var (8) # Variance of one number
[1] NA
> as.numeric (c("1", "2", "three", "4")) # Illegal conversion
[1] 1 2 NA 4
Warning message:
NAs introduced by coercion
> c(1, 2, 3)[4] # Vector subscript out of range
[1] NA
> NA - 1 # Most operations on NAs produce NAs
[1] NA

> a <- data.frame (a = 1:3, b = 2:4)
> a[4,] # Data frame row subscript out of range
  a b
NA NA NA
# The first NA there is the row number
> a[,4] # Specifying a non-existent column just produces an error
Error in `[.data.frame`(a, , 4) : undefined columns selected
```

EXERCISES

- Practice above examples that generate NA values
- Create NA values by some illegal operations
- Practice exercises in lecture slide
- What happens when we try to sort the data with NA values
- How to find the length of a vector with NA values

DATASET – INTRODUCTION

In today's lab we are going to work on dataset “*airquality*”

Details of Dataset:

Daily readings of the following air quality values for May 1, 1973 (a Tuesday) to September 30, 1973.

Description of variables:

Ozone: Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island

Solar.R: Solar radiation in Langleys in the frequency band 4000--7700 Angstroms from 0800 to 1200 hours at Central Park

Wind: Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport

Temp: Maximum daily temperature in degrees Fahrenheit at La Guardia Airport.

EXERCISES

- Print the dataset *airquality*
- Print the structure of the dataset *airquality*
- Print the summary of all the variables of the dataset *airquality* (**Hint:** Use function *summary()*)
- How many of the variables (columns) are in the dataset *airquality*
- How many observations (rows) are in the dataset *airquality*
- Use the function *is.na()* to find whether any missing values are in the dataset *airquality*
- Print the indices of the missing values in the dataset *airquality* in column major representation

- Print the indices of the missing values in the dataset *airquality* in row major representation
- Print indices of the missing values in row and column numberwise (**Hint:** Use function *which()* and *argument arr.ind = TRUE*)
- How many missing values are in the dataset *airquality*?
- Which variables are the missing values concentrated in?
- How would you omit all rows containing missing values?
- Print the records without missing values in the dataset *airquality* using the function *complete.cases()*
- Print the records without missing values in the dataset *airquality* using the function *na.omit()*
- Print the records without missing values in the dataset *airquality* using the function *na.exclude()*
- Print the records containing missing values in the dataset *airquality* using the function *complete.cases()*

Data Recoding:

Data Recoding in R can be done by either replacing data in an existing field or recoding into a new field based on criteria you specify.

EXERCISES

Consider a numeric vector `x <- c(3,4,5,6,7,8)`

- Write a command to recode the values less than 6 with zero in the vector x
- Write a command to recode the values between 4 and 8 with 100
- Write a command to recode the values that are less than 5 or greater than 6 with 50
- Write a command to recode the values less than 6 with NA in the vector x
- Write a command to recode the values between 4 and 8 with NA
- Write a command to recode the values that are less than 5 or greater than 6 with NA
- Count number of NA values after each operation
- Find mean of x (**Hint:** exclude NA values)
- Find median of x (**Hint:** exclude NA values)
- Write a command to recode the values less than 6 with “NA” (enclose NA with double quotes) in the vector x
- Write a command to recode the values between 4 and 8 with “NA”
- Write a command to recode the values that are less than 5 or greater than 6 with “NA”
- Count number of NA values after each operation
- Find mean of x (**Hint:** exclude NA values)
- Find median of x (**Hint:** exclude NA values)
- What is the difference between NA and “NA”

EXERCISES

Consider the given vectors:

```
A <- c(3, 2, NA, 5, 3, 7, NA, NA, 5, 2, 6)
```

```
B <- c(3, 2, NA, 5, 3, 7, NA, "NA", 5, 2, 6)
```

- Find the length of the vector A
- Find the length of the vector B
- Sort the values in vector A and put it in p (**Hint:** use function `sort()`)
- Find the length of p
- Sort the values in vector B and put it in q
- Find the length of q
- What did you infer from the above results

EXERCISES

Let us work on dataset – *airquality*

- Print the dataset *airquality*
- Print the structure of the dataset *airquality*
- Print the summary of all the variables of the dataset *airquality* (**Hint:** Use function *summary()*)
- How many of the variables (columns) are in the dataset *airquality*
- How many observations (rows) are in the dataset *airquality*

Observe the results of *summary()* function on dataset *airquality*. Attributes Ozone and Solar.R have missing values. Number of missing values are displayed at the bottom of each column if any.

```
> summary(airquality)
```

Ozone	Solar.R	Wind	Temp	Month	Day
Min. : 1.00	Min. : 7.0	Min. : 1.700	Min. : 56.00	Min. : 5.000	Min. : 1.0
1st Qu.: 18.00	1st Qu.: 115.8	1st Qu.: 7.400	1st Qu.: 72.00	1st Qu.: 6.000	1st Qu.: 8.0
Median : 31.50	Median : 205.0	Median : 9.700	Median : 79.00	Median : 7.000	Median : 16.0
Mean : 42.13	Mean : 185.9	Mean : 9.958	Mean : 77.88	Mean : 6.993	Mean : 15.8
3rd Qu.: 63.25	3rd Qu.: 258.8	3rd Qu.: 11.500	3rd Qu.: 85.00	3rd Qu.: 8.000	3rd Qu.: 23.0
Max. : 168.00	Max. : 334.0	Max. : 20.700	Max. : 97.00	Max. : 9.000	Max. : 31.0
NA's : 37	NA's : 7				

- What are the values getting displayed when we use *summary()* function
- What is quartile how to find them
- What are 1st and 3rd quartiles
- Copy the dataset *airquality* to *aq* (Better work on a copy of original data instead of working on original data to avoid the loss of information)
- Print the dataset *aq*
- Print the structure of the dataset *aq*
- Print the summary of all the variables of the dataset *aq* (**Hint:** Use function *summary()*)
- Print top 6 observations

- Print last 6 observations
- Replace the NA values in the attribute *Ozone* in *aq* by zero
- Print the summary of all the variables of the dataset *aq*
- Replace the NA values in the attribute *Ozone* in *aq* by mean of the remaining values.
Print the summary of the dataset *aq*
- Copy the dataset *airquality* to *aq1*. Replace the NA values in the attribute *Ozone* in *aq1* by median of the remaining values. Print the summary of the dataset *aq1*
- Copy the dataset *airquality* to *aq2*. Replace the NA values in the attribute *Ozone* in *aq2* by mode of the remaining values. Print the summary of the dataset *aq2*
- Repeat the above five operations for the attribute *Solar.R*
- Replace all the values of Temp with global constant 50 in *aq1*
- Replace all the values below 60 of Temp with global constant 60 in *aq2*
- Replace the month numbers in the column *Month* in *aq* by name of the month. (**Ex:** Replace **5** with **May**). (**Hint:** use *gsub()* function.
`aq$Month <- gsub(5,"May",aq$Month)`)
- Create a new logical attribute *Solar.Danger* in *aq* by filling it's value with TRUE if the value in the attribute *Solar.R* is greater than 100, other with FALSE
- Discretize the values in Temp of *aq* to “Low”, “Medium” and “High”
- What does *cut()* function do?
- Create a numeric vector *brks* containing values 0, 50, 100, 200, 250, 300 and 350. Divide the range of *Solar.R* into intervals and recode the values in *Solar.R* according to which interval they fall using the vector *brks*.
`aq$Solar.R=cut(aq$Solar.R,breaks=brks,include.lowest=TRUE)`
- Practice the examples given in lecture slide 12 on dataset *airquality*

Data Merging:

MERGING DATA

Merging data:

When combining separate dataframes, (in the R programming language), into a single dataframe, using the `cbind()` function usually requires use of the “`Match()`” function. To simulate the database joining functionality in SQL, the “`Merge()`” function in R accomplishes dataframe merging with the following protocols;

- “Inner Join” where the left table has matching rows from one, or more, key variables from the right table.
- “Outer Join” where all the rows from both tables are joined.
- “Left Join” where all rows from the left table, and any rows with matching keys from the right table are returned.
- “Right Join” where all rows from the right table, and any rows with matching keys from the left table are returned.

EXERCISES

- Practice the exercises explained in the class room (Follow the lecture slides) first and solve the exercises given below later:

Exercise 1

Create the dataframes to merge:

```
buildings <- data.frame(location=c(1, 2, 3), name=c("building1", "building2",  
"building3"))
```

```
data <- data.frame(survey=c(1,1,1,2,2,2), location=c(1,2,3,2,3,1),  
efficiency=c(51,64,70,71,80,58))
```

The dataframes, buildings and data have a common key variable called, “location”. Use the merge() function to merge the two dataframes by “location”, into a new dataframe, “buildingStats”.

Exercise 2

Give the dataframes different key variable names:

```
buildings <- data.frame(location=c(1, 2, 3), name=c("building1", "building2",  
"building3"))
```

```
data <- data.frame(survey=c(1,1,1,2,2,2), LocationID=c(1,2,3,2,3,1),  
efficiency=c(51,64,70,71,80,58))
```

The dataframes, buildings and data now have corresponding variables called, location, and LocationID. Use the merge() function to merge the columns of the two dataframes by the corresponding variables.

Exercise 3

Inner Join:

The R merge() function automatically joins the frames by common variable names. In that case, demonstrate how you would perform the merge in Exercise 1 without specifying the key variable.

Exercise 4

Outer Join:

Merge the two dataframes from Exercise 1. Use the “all=” parameter in the merge() function to return all records from both tables. Also, merge with the key variable, “location”.

Exercise 5

Left Join:

Merge the two dataframes from Exercise 1, and return all rows from the left table. Specify the matching key from Exercise 1.

Exercise 6

Right Join:

Merge the two dataframes from Exercise 1, and return all rows from the right table. Use the matching key from Exercise 1 to return matching rows from the left table.

Exercise 7

Cross Join:

Merge the two dataframes from Exercise 1, into a “Cross Join” with each row of “buildings” matched to each row of “data”. What new column names are created in “buildingStats”?

Exercise 8

Merging Dataframe rows:

To join two data frames (datasets) vertically, use the rbind function. The two data frames must have the same variables, but they do not have to be in the same order.

Merge the rows of the following two dataframes:

```
buildings <- data.frame(location=c(1, 2, 3), name=c("building1", "building2", "building3"))
```

```
buildings2 <- data.frame(location=c(5, 4, 6), name=c("building5", "building4", "building6"))
```

Also, specify a new dataframe, “allBuidings”.

- Apply different join operations on the tables given below. Write the expected outputs and compare them with the outputs obtained by R commands

Super Heroes

Name	Alignment	Gender	Publisher
Magneto	bad	male	Marvel
Storm	good	female	Marvel
Mystique	bad	female	Marvel
Batman	good	male	DC
Joker	bad	male	DC
Catwoman	bad	female	DC
Hellboy	good	male	Dark Horse Comics

Publishers

publisher	yr_founded
DC	1934
Marvel	1939
Image	1992

Data Import and Export:

EXPORT DATA

Exporting data:

There are plenty of functions for writing data to files. Some of them are:

- write.table
- write.csv
- cat
- writeLines
- dump
- dput
- save
- serialize

EXERCISES

Consider the data set “airquality”.

Read first 6 lines into a new data frame “aq” (aq <- head(airquality))

Practice the following exercises on the following data.

```
> aq <- head(airquality)
> aq
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Note:

After creating files, open and see the files to comply with the required output.

.csv files can be opened either by excel or notepad

You can observe the delimiters when you open the file using notepad.

Cat command:

- Write a command to export (store/save) data into the file cat_test1.txt (Use only two arguments). After creating file check the output.
- Write a command to export data into the file cat_test2.txt. Use separator as comma
- Write a command to export data into the file cat_test3.txt. Use separator as semi-colon
- Write a command to export data into the file cat_test3.txt. Use separator as tab (use \t to insert tab)
- Can a separator be any string that you want to insert as delim?? Experiment it.
- What are Delimiters? What is the need of delimiters?
- What is the difference between over writing and appending?
- Write a command to append the same data into the file cat_test1.txt. After creating file check the output.
- Write a command to append the same data into the file cat_test2.txt. Use separator as comma
- Write a command to append the same data into the file cat_test3.txt. Use separator as semi-colon
- Write a command to append the same data into the file cat_test3.txt. Use separator as tab (use \t to insert tab)
- Can you store the data into .csv files using cat command?
- Write a command to export the data to cat_test1.csv.
- Write a command to export the data using various separators such as semi-colon and tab and store them in new .csv files. Open newly created files using both excel and notepad. Observe the difference.
- What are other arguments you can use while exporting data to a file using cat function. (**Hint:** Use ?cat command to learn about other arguments and experiment on them)

write.table command:

Repeat the above exercises using write.table command. Here, file extension is .txt as write.table stores data into text files.

write.csv command:

Repeat the above exercises using write.csv command. Here, file extension is .csv as write.csv stores data into comma separated files.

Other commands:

Practice the below commands using the examples given in lecture slides

- writeLines, readLines
- dump, source
- dput, dget
- save, load
- serialize, unserialize
- Scan

Importing data:

There are a few principal functions reading data into R.

- read.table, read.csv, for reading tabular data
- readLines, for reading lines of a text file
- source, for reading in R code files (inverse of dump)
- dget, for reading in R code files (inverse of dput)
- load, for reading in saved workspaces
- unserialize, for reading single R objects in binary form
- read.delim(), for reading data separated by tab

EXERCISES

Note:

`read.csv` is identical to `read.table` except that the default separator is a comma

There are variants in the functions in `read.csv` and `read.delim` such as `read.csv2` and `read.delim2`.

read.table command:

- Write a command to read the data from `cat_test1.txt`. Display the output.
- Write a command to read the data from `cat_test2.txt`. Display the output. Modify your command to get the output as given below

```
Ozone Solar.R Wind Temp Month Day
1    41    190  7.4   67     5   1
2    36    118  8.0   72     5   2
3    12    149 12.6   74     5   3
4    18    313 11.5   62     5   4
5    NA     NA 14.3   56     5   5
6    28     NA 14.9   66     5   6
```

- Write a command to read the data from remaining files and display the output as in the above figure.
- Write a command to read and display the data without quotes on strings
- Write a command to read the data without column names
- Write a command to read the data without row names

Read.csv command:

Repeat the above exercises using `read.csv` command

Data Cleaning and Summarizing with “dplyr” package:

The dplyr package is one of the most powerful and popular package in R. This package was written by the most popular R programmer Hadley Wickham who has written many useful R packages such as ggplot2, tidyr etc. This post includes several examples and tips of how to use dplyr package for cleaning and transforming data. It's a complete tutorial on data manipulation and data wrangling with R.

What is dplyr?

The dplyr is a powerful R-package to manipulate, clean and summarize unstructured data. In short, it makes data exploration and data manipulation easy and fast in R.

What's special about dplyr?

The package "dplyr" comprises many functions that perform mostly used data manipulation operations such as applying filter, selecting specific columns, sorting data, adding or deleting columns and aggregating data.

Another most important advantage of this package is that it's very easy to learn and use dplyr functions. Also easy to recall these functions. For example, filter() is used to filter rows.

dplyr vs. Base R Functions

dplyr functions process faster than base R functions. It is because dplyr functions were written in a computationally efficient manner. They are also more stable in the syntax and better supports data frames than vectors.

SQL Queries vs. dplyr

People have been utilizing SQL for analyzing data for decades. Every modern data analysis software such as Python, R, SAS etc supports SQL commands. But SQL was never designed to perform data analysis. It was rather designed for querying and managing data. There are many data

analysis operations where SQL fails or makes simple things difficult. For example, calculating median for multiple variables, converting wide format data to long format etc. Whereas, dplyr package was designed to do data analysis.

The names of dplyr functions are similar to SQL commands such as select() for selecting variables, group_by() - group data by grouping variable, join() - joining two data sets. Also includes inner_join() and left_join(). It also supports sub queries for which SQL was popular for.

How to install and load dplyr package

To install the dplyr package, type the following command.

```
install.packages("dplyr")
```

To load dplyr package, type the command below

```
library(dplyr)
```

dplyr Function	Description	Equivalent SQL
select()	Selecting columns (variables)	SELECT
filter()	Filter (subset) rows.	WHERE
group_by()	Group the data	GROUP BY
summarise()	Summarise (or aggregate) data	-
arrange()	Sort the data	ORDER BY
join()	Joining data frames (tables)	JOIN
mutate()	Creating New Variables	COLUMN ALIAS

ALIAS

Data : Income Data by States

In this tutorial, we are using the following data which contains income generated by states from year 2002 to 2015. Note : This data do not contain actual income figures of the states.

This dataset contains 51 observations (rows) and 16 variables (columns).

The snapshot of first 6 rows of the dataset is shown below.

Index	State	Y2002	Y2003	Y2004	Y2005	Y2006	Y2007	Y2008	Y2009
1	A	Alabama	1296530	1317711	1118631	1492583	1107408	1440134	1945229
2	A	Alaska	1170302	1960378	1818085	1447852	1861639	1465841	1551826
3	A	Arizona	1742027	1968140	1377583	1782199	1102568	1109382	1752886
4	A	Arkansas	1485531	1994927	1119299	1947979	1669191	1801213	1188104
5	C	California	1685349	1675807	1889570	1480280	1735069	1812546	1487315
6	C	Colorado	1343824	1878473	1886149	1236697	1871471	1814218	1875146
		Y2010	Y2011	Y2012	Y2013	Y2014	Y2015		
1		1237582	1440756	1186741	1852841	1558906	1916661		
2		1629616	1230866	1512804	1985302	1580394	1979143		
3		1300521	1130709	1907284	1363279	1525866	1647724		
4		1669295	1928238	1216675	1591896	1360959	1329341		
5		1624509	1639670	1921845	1156536	1388461	1644607		
6		1913275	1665877	1491604	1178355	1383978	1330736		

Download the Dataset

How to load Data

Submit the following code. Change the file path in the code below.

```
mydata = read.csv("C:\\Users\\Deepanshu\\Documents\\sampledata.csv")
```

Example 1 : Selecting Random N Rows

The `sample_n` function selects random rows from a data frame (or table).

The second parameter of the function tells R the number of rows to select.

```
sample_n(mydata,3)
```

```
Index State Y2002 Y2003 Y2004 Y2005 Y2006 Y2007 Y2008 Y2009
```

```
2 A Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
```

```
8 D Delaware 1330403 1268673 1706751 1403759 1441351 1300836 1762096 1553585
```

```
33 N New York 1395149 1611371 1170675 1446810 1426941 1463171 1732098 1426216
```

```
Y2010 Y2011 Y2012 Y2013 Y2014 Y2015
```

```
2 1629616 1230866 1512804 1985302 1580394 1979143
```

```
8 1370984 1318669 1984027 1671279 1803169 1627508
```

```
33 1604531 1683687 1500089 1718837 1619033 1367705
```

Example 2 : Selecting Random Fraction of Rows

The `sample_frac` function returns randomly N% of rows. In the example below, it returns randomly 10% of rows.

```
sample_frac(mydata,0.1)
```

Example 3 : Remove Duplicate Rows based on all the variables

(Complete Row)

The `distinct` function is used to eliminate duplicates.

```
x1 = distinct(mydata)
```

In this dataset, there is not a single duplicate row so it returned same number of rows as in `mydata`.

Example 4 : Remove Duplicate Rows based on a variable

The `.keep_all` function is used to retain all other variables in the output data frame.

```
x2 = distinct(mydata, Index, .keep_all= TRUE)
```

Example 5 : Remove Duplicates Rows based on multiple variables

In the example below, we are using two variables - `Index`, `Y2010` to determine uniqueness.

```
x2 = distinct(mydata, Index, Y2010, .keep_all= TRUE)
```

select() Function

It is used to select only desired variables.

`select()` syntax : `select(data ,)`

`data` : Data Frame

`....` : Variables by name or by function

Example 6 : Selecting Variables (or Columns)

Suppose you are asked to select only a few variables. The code below selects variables "Index", columns from "State" to "Y2008".

```
mydata2 = select(mydata, Index, State:Y2008)
```

Example 7 : Dropping Variables

The minus sign before a variable tells R to drop the variable.

```
mydata = select(mydata, -Index, -State)
```

The above code can also be written like :

```
mydata = select(mydata, -c(Index,State))
```

Example 8 : Selecting or Dropping Variables starts with 'Y'

The starts_with() function is used to select variables starts with an alphabet.

```
mydata3 = select(mydata, starts_with("Y"))
```

Adding a negative sign before starts_with() implies dropping the variables starts with 'Y'

```
mydata33 = select(mydata, -starts_with("Y"))
```

The following functions helps you to select variables based on their names.

Helpers Description

starts_with() Starts with a prefix

ends_with() Ends with a prefix

contains() Contains a literal string

matches() Matches a regular expression

Output

num_range() Numerical range like x01, x02, x03.

one_of() Variables in character vector.

everything() All variables.

Example 9 : Selecting Variables contain 'T' in their names

```
mydata4 = select(mydata, contains("T"))
```


Example 10 : Reorder Variables

The code below keeps variable 'State' in the front and the remaining variables follow that.

```
mydata5 = select(mydata, State, everything())
```

New order of variables are displayed below -

```
[1] "State" "Index" "Y2002" "Y2003" "Y2004" "Y2005" "Y2006" "Y2007"  
"Y2008" "Y2009"  
[11] "Y2010" "Y2011" "Y2012" "Y2013" "Y2014" "Y2015"
```

rename() Function

It is used to change variable name.

rename() syntax : `rename(data , new_name = old_name)`

data : Data Frame

new_name : New variable name you want to keep

old_name : Existing Variable Name

Example 11 : Rename Variables

The rename function can be used to rename variables.

In the following code, we are renaming 'Index' variable to 'Index1'.

```
mydata6 = rename(mydata, Index1=Index)
```

filter() Function

It is used to subset data with matching logical conditions.

filter()

Syntax : `filter(data ,)`

data : Data Frame

.... : Logical Condition

Example 12 : Filter Rows

Suppose you need to subset data. You want to filter rows and retain only those values in which Index is equal to A.

```
mydata7 = filter(mydata, Index == "A")
```

Index State Y2002 Y2003 Y2004 Y2005 Y2006 Y2007 Y2008 Y2009

1 A Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173

2 A Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541

3 A Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330

4 A Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980

Y2010 Y2011 Y2012 Y2013 Y2014 Y2015

1 1237582 1440756 1186741 1852841 1558906 1916661

2 1629616 1230866 1512804 1985302 1580394 1979143

3 1300521 1130709 1907284 1363279 1525866 1647724

Example 13 : Multiple Selection Criteria

The %in% operator can be used to select multiple items. In the following program, we are telling R to select rows against 'A' and 'C' in column 'Index'.

```
mydata7 = filter(mydata6, Index %in% c("A", "C"))
```

Example 14 : 'AND' Condition in Selection Criteria

Suppose you need to apply 'AND' condition. In this case, we are picking data for 'A' and 'C' in the column 'Index' and income greater than 1.3 million in Year 2002.

```
mydata8 = filter(mydata6, Index %in% c("A", "C") & Y2002 >= 1300000 )
```

Example 15 : 'OR' Condition in Selection Criteria

Output

The 'I' denotes OR in the logical condition. It means any of the two conditions.

```
mydata9 = filter(mydata6, Index %in% c("A", "C") | Y2002 >= 1300000)
```

Example 16 : NOT Condition

The "!" sign is used to reverse the logical condition.

```
mydata10 = filter(mydata6, !Index %in% c("A", "C"))
```

Example 17 : CONTAINS Condition

The grepl function is used to search for pattern matching. In the following code, we are looking for records wherein column state contains 'Ar' in their name.

```
mydata10 = filter(mydata6, grepl("Ar", State))
```

summarise() Function

It is used to summarize data.

summarise() syntax : summarise(data ,)

data : Data Frame

..... : Summary Functions such as mean, median etc

Example 18 : Summarize selected variables

In the example below, we are calculating mean and median for the variable Y2015.

```
summarise(mydata, Y2015_mean = mean(Y2015),  
Y2015_med=median(Y2015))
```

Example 19 : Summarize Multiple Variables

In the following example, we are calculating

number of records, mean and median for

variables Y2005 and Y2006. The summarise_at function allows us to select multiple variables by their names.

```
summarise_at(mydata, vars(Y2005, Y2006), funs(n(), mean, median))
```

Example 20 : Summarize with Custom Functions

We can also use custom functions in the summarise function. In this case, we are computing the number of records, number of missing values, mean and median for variables Y2011 and Y2012. The dot (.) denotes each variables specified in the second argument of the function.

```
summarise_at(mydata, vars(Y2011, Y2012),  
funs(n(), missing = sum(is.na(.)), mean(., na.rm = TRUE), median(.,na.rm = TRUE)))
```

Summarize : Output

How to apply Non-Standard Functions

Suppose you want to subtract mean from its original value and then calculate variance of it.

```
set.seed(222)
mydata <- data.frame(X1=sample(1:100,100), X2=runif(100))
summarise_at(mydata,vars(X1,X2), function(x) var(x - mean(x)))
X1 X2
1 841.6667 0.08142161
```

Example 21 : Summarize all Numeric Variables

The summarise_if function allows you to summarise conditionally.

```
summarise_if(mydata, is.numeric, funs(n(),mean,median))
```

Alternative Method :

First, store data for all the numeric variables

```
numdata = mydata[sapply(mydata,is.numeric)]
```

Second, the summarise_all function calculates summary statistics for all the columns in a data frame

```
summarise_all(numdata, funs(n(),mean,median))
```

Example 22 : Summarize Factor Variable

We are checking the number of levels/categories and count of missing observations in a categorical (factor) variable.

```
summarise_all(mydata["Index"], funs(nlevels(.), nmiss=sum(is.na(.))))
nlevels nmiss
1 19 0
```

arrange() function :

Use : Sort data

Syntax

```
arrange(data_frame, variable(s)_to_sort)
```

```
or data_frame %>% arrange(variable(s)_to_sort)
```

To sort a variable in descending order, use desc(x).

Example 23 : Sort Data by Multiple Variables

The default sorting order of arrange() function is ascending. In this example, we are sorting data by multiple variables.

```
arrange(mydata, Index, Y2011)
```

Suppose you need to sort one variable by descending order and other variable by ascending order.

```
arrange(mydata, desc(Index), Y2011)
```

Pipe Operator %>%

It is important to understand the pipe (%>%) operator before knowing the other functions of dplyr package. dplyr utilizes pipe operator from another package (magrittr).

It allows you to write sub-queries like we do it in sql.

Note : All the functions in dplyr package can be used without the pipe operator. The question arises "Why to use pipe operator %>%". The answer is it lets to wrap multiple functions together with the use of %>%.

Syntax :

```
filter(data_frame, variable == value)
```

or

```
data_frame %>% filter(variable == value)
```

The %>% is NOT restricted to filter function. It can be used with any function.

Example :

The code below demonstrates the usage of pipe %>% operator. In this example, we are selecting 10 random observations of two variables "Index" "State" from the data frame "mydata".

```
dt = sample_n(select(mydata, Index, State),10)
```

or

```
dt = mydata %>% select(Index, State) %>% sample_n(10)
```

group_by() function :

Use : Group data by categorical variable

Syntax :

```
group_by(data, variables)
```

or

```
data %>% group_by(variables)
```

Example 24 : Summarise Data by

Categorical Variable

We are calculating count and mean of variables Y2011 and Y2012 by variable Index.

```
t = summarise_at(group_by(mydata, Index), vars(Y2011, Y2012), funs(n(), mean(., na.rm = TRUE)))
```

The above code can also be written like

```
t = mydata %>% group_by(Index) %>%  
summarise_at(vars(Y2011:Y2015), funs(n(), mean(., na.rm = TRUE)))
```

Index Y2011_n Y2012_n Y2013_n Y2014_n Y2015_n Y2011_mean Y2012_mean

A 4 4 4 4 4 1432642 1455876

C 3 3 3 3 3 1750357 1547326

D 2 2 2 2 2 1336059 1981868

F 1 1 1 1 1 1497051 1131928

G 1 1 1 1 1 1851245 1850111

H 1 1 1 1 1 1902816 1695126

I 4 4 4 4 4 1690171 1687056

K 2 2 2 2 2 1489353 1899773

L 1 1 1 1 1 1210385 1234234

M 8 8 8 8 8 1582714 1586091

N 8 8 8 8 8 1448351 1470316

O 3 3 3 3 3 1882111 1602463

P 1 1 1 1 1 1483292 1290329

R 1 1 1 1 1 1781016 1909119

S 2 2 2 2 2 1381724 1671744

T 2 2 2 2 2 1724080 1865787

U 1 1 1 1 1 1288285 1108281

V 2 2 2 2 2 1482143 1488651

W 4 4 4 4 4 1711341 1660192

do() function :

Use : Compute within groups

Syntax :

```
do(data_frame, expressions_to_apply_to_each_group)
```

Note : The dot (.) is required to refer to a data frame.

Example 25 : Filter Data within a Categorical Variable

Suppose you need to pull top 2 rows from 'A', 'C' and 'I' categories of variable Index.

```
t = mydata %>% filter(Index %in% c("A", "C", "I")) %>% group_by(Index)
%>%
```

```
do(head( . , 2))
```

Output : do() function

Example 26 : Selecting 3rd Maximum Value by Categorical Variable We are calculating third maximum value of variable Y2015 by variable Index. The following code first selects only two variables Index and Y2015.

Then it filters the variable Index with 'A', 'C' and 'I' and then it groups the same variable and sorts the variable Y2015 in descending order. At last, it selects the third row.

```
t = mydata %>% select(Index, Y2015) %>%
filter(Index %in% c("A", "C", "I")) %>%
group_by(Index) %>%
do(arrange(.,desc(Y2015))) %>% slice(3)
```

The slice() function is used to select rows by position.

Using Window Functions

Like SQL, dplyr uses window functions that are used to subset data within a group. It returns a vector of values. We could use `min_rank()` function that calculates rank in the preceding example,

```
t = mydata %>% select(Index, Y2015) %>%  
filter(Index %in% c("A", "C", "I")) %>%  
group_by(Index) %>%  
filter(min_rank(desc(Y2015)) == 3)  
Index Y2015  
1 A 1647724  
2 C 1330736  
3 I 1583516
```

Example 27 : Summarize, Group and Sort Together

In this case, we are computing mean of variables Y2014 and Y2015 by variable Index. Then sort the result by calculated mean variable Y2015.

```
t = mydata %>%  
group_by(Index)%>%  
summarise(Mean_2014 = mean(Y2014, na.rm=TRUE),  
Mean_2015 = mean(Y2015, na.rm=TRUE)) %>%  
arrange(desc(Mean_2015))
```

mutate() function :

Use : Creates new variables

Syntax :

```
mutate(data_frame, expression(s) )
```

or

```
data_frame %>% mutate(expression(s))
```


Example 28 : Create a new variable The following code calculates division of Y2015 by Y2014 and name it "change".

```
mydata1 = mutate(mydata, change=Y2015/Y2014)
```

Example 29 : Multiply all the variables by 1000

It creates new variables and name them with suffix "_new".

```
mydata11 = mutate_all(mydata, funs("new" = .* 1000))
```

The output shown in the image above is truncated due to high number of variables.

Note - The above code returns the following error messages -

Warning messages:

1: In Ops.factor(c(1L, 1L, 1L, 1L, 2L, 2L, 2L, 3L, 3L, 4L, 5L, 6L, :

‘*’ not meaningful for factors

2: In Ops.factor(1:51, 1000) : ‘*’ not meaningful for factors

It implies you are multiplying 1000 to string(character) values which are stored as factor variables. These variables are 'Index', 'State'. It does not make sense to apply multiplication operation on character variables. For these two variables, it creates newly created variables which contain only

NA.

Solution :See Example 45 -Apply multiplication on only numeric variables

Example 30 : Calculate Rank for Variables

Suppose you need to calculate rank for variables Y2008 to Y2010.

```
mydata12 = mutate_at(mydata, vars(Y2008:Y2010), funs(Rank=min_rank(.)))
```

By default, min_rank() assigns 1 to the smallest value and high number to the largest value.

In case, you need to assign rank 1 to the largest value of a variable, use min_rank(desc(.))

```
mydata13 = mutate_at(mydata,
```

```
vars(Y2008:Y2010),
```

```
funs(Rank=min_rank(desc(.))))
```

Example 31 : Select State that generated highest income among the variable 'Index'

```
out = mydata %>% group_by(Index) %>% filter(min_rank(desc(Y2015)) == 1) %>%  
select(Index, State, Y2015)
```

Index State Y2015

1	A	Alaska	1979143
2	C	Connecticut	1718072
3	D	Delaware	1627508
4	F	Florida	1170389
5	G	Georgia	1725470
6	H	Hawaii	1150882
7	I	Idaho	1757171
8	K	Kentucky	1913350
9	L	Louisiana	1403857
10	M	Missouri	1996005
11	N	New Hampshire	1963313
12	O	Oregon	1893515
13	P	Pennsylvania	1668232
14	R	Rhode Island	1611730
15	S	South Dakota	1136443
16	T	Texas	1705322
17	U	Utah	1729273
18	V	Virginia	1850394
19	W	Wyoming	1853858

Example 32 : Cumulative Income of 'Index' variable

The cumsum function calculates cumulative sum of a variable. With mutate function, we insert a new variable called 'Total' which contains values of cumulative income of variable Index.

```
out2 = mydata %>% group_by(Index) %>% mutate(Total=cumsum(Y2015)) %>%  
select(Index, Y2015, Total)
```

join() function :

Use : Join two datasets

Syntax :

Output : INNER JOIN

`inner_join(x, y, by =)`

`left_join(x, y, by =)`

`right_join(x, y, by =)`

`full_join(x, y, by =)`

`semi_join(x, y, by =)`

`anti_join(x, y, by =)`

x, y - datasets (or tables) to merge / join

by - common variable (primary key) to join by.

Example 33 : Common rows in both the tables

```
df1 = data.frame(ID = c(1, 2, 3, 4, 5),
```

```
w = c('a', 'b', 'c', 'd', 'e'),
```

```
x = c(1, 1, 0, 0, 1),
```

```
y=rnorm(5),
```

```
z=letters[1:5])
```

```
df2 = data.frame(ID = c(1, 7, 3, 6, 8),
```

```
a = c('z', 'b', 'k', 'd', 'l'),
```

```
b = c(1, 2, 3, 0, 4),
```

```
c =rnorm(5),
```

```
d =letters[2:6])
```

INNER JOIN returns rows when there is a match in both tables. In this example, we are merging df1 and df2 with ID as common variable (primary key).

```
df3 = inner_join(df1, df2, by = "ID")
```

If the primary key does not have same name in both the tables, try the following way:

```
inner_join(df1, df2, by = c("ID"="ID1"))
```

Example 34 : Applying LEFT JOIN

LEFT JOIN : It returns all rows from the left table, even if there are no matches in the right table.

```
left_join(df1, df2, by = "ID")
```

Output : LEFT JOIN

Combine Data Vertically

intersect(x, y)

Rows that appear in both x and y.

union(x, y)

Rows that appear in either or both x and y.

setdiff(x, y)

Rows that appear in x but not y.

Example 35 : Applying INTERSECT

Prepare Sample Data for Demonstration

```
mtcars$model <- rownames(mtcars)
```

```
first <- mtcars[1:20, ]
```

```
second <- mtcars[10:32, ]
```

INTERSECT selects unique rows that are common to both the data frames.

```
intersect(first, second)
```

Example 36 : Applying UNION

UNION displays all rows from both the tables and removes duplicate records from the combined dataset. By using union_all function, it allows duplicate rows in the combined dataset.

```
x=data.frame(ID = 1:6, ID1= 1:6)
```

```
y=data.frame(ID = 1:6, ID1 = 1:6)
```

```
union(x,y)
```

```
union_all(x,y)
```

Example 37 : Rows appear in one table but not in other table

```
setdiff(first, second)
```

Example 38 : IF ELSE Statement

Syntax :

```
if_else(condition, true, false, missing = NULL)
```

true : Value if condition meets

false : Value if condition does not meet

missing : Value if missing cases. It will be used to replace missing values (Default : NULL)

```
df <- c(-10,2, NA)
```

```
if_else(df < 0, "negative", "positive", missing = "missing value")
```

Create a new variable with IF_ELSE

If a value is less than 5, add it to 1 and if it is greater than or equal to 5, add it to 2. Otherwise 0.

```
df =data.frame(x = c(1,5,6,NA))
```

```
df %>% mutate(newvar=if_else(x<5, x+1, x+2,0))
```

Nested IF ELSE

Multiple IF ELSE statement can be written using if_else() function. See the example below

-

```
mydf =data.frame(x = c(1:5,NA))  
mydf %>% mutate(newvar= if_else(is.na(x),"I am  
missing",  
if_else(x==1,"I am one",  
if_else(x==2,"I am two",  
if_else(x==3,"I am three","Others")))))
```

Output

x flag

1 1 I am one

2 2 I am two

3 3 I am three

4 4 Others

5 5 Others