**Setting a new working directory in R:**

Open R by typing R in command in terminal.

getwd()

setwd(“/mnt/c/Users/Hp/Desktop/r-novice-inflammation”)

**Loading the data into R**

Read.csv(file = “data/inflammation-01.csv”, header = FALSE)

Read.csv(…) is a function that asks R to run the function read.csv. read.csv has two arguments: the name of the file we want to read, and whether the first line of the file contains names for the columns of data. The filename needs to be a string, so we put it in quotes. Assigning the second argument, header, to be false indicates that the data file does not have column headers. The default value is header = TRUE.

**Assigning a number to a newly created variable**

weight\_kg (creating a new variable)

weight\_kg <- 55 ( assigning a number to this variable )

weight\_kg ( printing output)

**Commenting**

We can add comments to our code using the # character.

**Note:** We can also load the data into the variable.

dat <- read.csv(file = “data/inflammation-01.csv”, header = FALSE)

head(variable\_name) : display only the first few rows of data.

class(dat) : output gives us the structure which in this case is data frame, this structure is like the spreadsheet in MS Excel.

Dim(dat): gives the number of rows and columns.

c <- array(dim=c(3,2,2)) # This command means that c will have 3 rows, 2 columns and 2 tables.

To get a single value from the data frame, we can provide an index in square brackets. The first number specifies the row and the second the column:

Dat[30, 20], row 30, column 20.

If we want to select more than one row or column, we can use the function c, which stands for combine.

Dat[c(1, 3, 5), c(10, 20)], (1, 3, 5) is representing the rows and (10,20) is representing the columns.

: operator – This special function generates sequences of numbers.

n:m, output gives number from n to m.

dat[1:4, 1:10], we can select the first ten columns of values for the first four rows.

**If we want to select all rows or all columns, leave that index value empty:**

Dat[5, ], outputs all columns from row 5.

Dat[ , ], we get the entire data frame.

Max(dat[2, ]), gives the max number from the entire row.

Min(dat[2, ], gives the min number from the entire row.

Mean(dat[ , 7]), mean inflammation on day 7.

Median(dat[ ,7]), median inflammation on day 7.

Sd(dat[ ,7]), standard deviation of inflammation on day 7.

Summary(dat[ , 1:4]). Summarize the previous common calculations. For every column in the data frame, the function “summary” calculates: the minimun value, the first quartile, the median, the mean, the third quartile and the max value.

R may return an error when you attempt to perform similar calculations on subsetted rows of data frames. This is because some functions in R automatically convert the object type to a numeric vector, while others do not. We get the expected output by including an explicit call to as.numeric(), e.g. mean(as.numeric(dat[1,])). Columns of data frames are already defined as vectors.

**Apply function:** It allows us to repeat a function on all the rows or columns of a data frame.

apply(X, MARGIN, FUN, ...)

X: an array, including a matrix.

Margin: A vector giving the subscripts which the function will be applied over. E.g., for a matrix ‘1’ indicates rows, ‘2’ indicates columns, ‘c(1, 2)’ indicates rows and columns.’

FUN: the function to be applied: see ‘Details’. In the case of functions like ‘+’, ‘%\*%’, etc., the function name must be backquoted or quoted.

…. : optional arguments to ‘FUN’.

avg\_patient\_inflammation <- apply(dat, 1, mean) : To obtain the average inflammation of each patient we will need to calculate the mean of all the rows (MARGIN = 1) of the data frame.

**Lapply:** lapply is used to apply a function over a list. Lapply always returns a list of the same length as the input list.

Syntax: lapply(list, function, …)

Example: A <- matrix(1:9,3,3)

B <- matrix(10:18,3,3)

Mylist <- list(A,B)

Determinant <- lapply(Mylist,det)

**Mapply:** mapply is aa multivariative version of lapply. This function can be applied over several lists simultaneously.

Syntax: mapply(fun, list1, list2, ….)

Example: m <- c(15,40,85,100,75,24)

mw <- c(40,54,48,75,65,98)

n <- mapply(number\_of\_molecules,m,mw)

**Note:** A list cannot be handled by cat function.

**tapply function:** tapply is used to apply a function over a subset of vectors given by a combination of factors.

Syntax: tapply(vector, factors, function, …)

Example: Id <- c(1,1,1,1,2,2,2,3,3)

Values <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)

tapply (Values,Id,sum)

Output: 1 2 3

10 18 17 (1+2+3+4 5+6+7 8+9)

**Plotting:** Plotting the values is done with the function plot.

plot(avg\_day\_inflammation)

**Let suppose we have a dataset called Iris and from which I want to plot the Petal.length data, so for this we will write:**

plot(Iris$Petal.length)

**If we want to plot a graph with Petal.width in the Y axis and Petal.length in X axis, we will write the following command:**

plot(Iris$Petal.length, Iris$Petal.width)

**Plot with options->**

plot(Iris$Petal.length, Irsi$Petal.width,

col = “Hex code of the color”, => Decides the color of the graph point.

pch = numeric value, => Decides the point character shape, 19 is for solid circle

main = “ ”, => Title of the plot

xlab = “ ”, => Labeling of x axis

ylab = “ ”, => Labeling of Y axis

**Formula Plot->**

plot(cos,0,2\*pi) => This will plot cosine function from 0 to 2\*pi limit.

plot(exp,1,5) => This will plot exponential function from 1 to 5 limit.

**BAR CHART:** We will not be able to make a bar chart by simply using raw data. So, to make a bar chart **we will first have to create a summary table** and to do so we will write the following instructions:

**Let us suppose we want to draw a bar chart of cyl data points from the mtcar dataset. So, to make a summary table the piece of code will be:**

cylinder <- table(mtcar$cyl) => The summary table will be stored to the cylinder matrix.

Now we can plot the bar chart using:

barplot(cylinder)

**HISTOGRAMS:** Histograms are used when data is quantitative, scaled, measured, interval or ratio level.

**Syntax:** hist(data\_points). Example: **hist(Iris$Petal.length) => gives the histogram of the petal.length datapoints.**

**Histograms by groups:**

In the Iris datasets, we have three species, and, in this example, we are trying to make individual histograms for individual species of Iris. So, to do so we will write following piece of code:

par(mfrow = c(3,1)) => par stands for parameter and mfrow is defining the number of rows and columns. This code will put the graph in 3 rows and 1 column.

hist(Iris$Petal.width [Iris$Species == “setosa”], => Selecting the setosa species

xlim = c(0,3),=> xlim stands for the limit of x and this will make sure the all the datapoints have same x scale.

breaks = 9, => Not so important

main = “Petal Width for Setosa”, => Title

xlab = “”, => X axis title

col = “red”) => Color of histogram

hist(Iris$Petal.width [Iris$Species == “versicolor”],

xlim = c(0,3),

breaks = 9,

main = “Petal Width for versicolor”,

xlab = “”

col = “purple”)

hist(Iris$Petal.width [Iris$Species == “virginica”],

xlim = c(0,3),

breaks = 9,

main = “Petal Width for virginica”,

xlab = “”

col = “blue”)

**This piece of code will create three histograms for three different species.**

# Restoring the graphic parameter

par(mfrow=c(1,1)

**SCATTERPLOTS:** We will use scatterplots for visualizing the association between two quantitative variables.

**Syntax:** plot(mtcars$wt, mtcars$mpg) => We can also add some options to the plot command.

Example: plot(mtcars$wt, mtcars$mpg,

pch = 19, => Solid circle

cex = 1.5, => Make 150% size

col = “hex code”,

main = “Title”,

xlab = “x-axis label”,

ylab = “y-axis label”)

**OVERLAYING PLOTS:** Overlay plots are useful for displaying data from both output databases in a co-simulation in the same viewport. Overlay plots are composed of layers; each layer contains one plot, and the layers are stacked on top of each other to create the combined plot.

**Creating a Histogram:** hist(lynx,

breaks = 14,

fre = FALSE,

col = “thistle1”

main = “title”

xlab = “x-axis label”)

**#Add a normal Distribution**

curve(dnorm(x, mean = mean(lynx), sd = sd(lynx)), => dnorm stands for density of normal distribution

col = “thistle4”,

lwd = 2, => Line width of 2 pixels

add = TRUE) => Superimpose on previous graph

**# Add two kernel density estimators, sort of like bell curve but they follow the distribution of the data.**

lines(density(lynx), col = “blue”, lwd = 2)

lines(density(lynx, adjust = 3), col = “purple”, lwd = 2) => adjust value is the moving average.

**# Add a rug plot, rug plot is a plot of data for a single quantitative variable, displayed as marks along an axis. It is analogous to a one-dimensional scatter plot.**

R rug(lynx, lwd = 2, col = “gray”)

We will get an overlaying plot.

**Describe():** It gets more detail. It is not included with the base functionality of R, instead it comes from the psych package. Using this function, we will get n(sample size), mean, S.D., median, 10% trimmed mean, mean absolute deviation(MAD), min/max, range, skewness, kurtosis, and standard errors.

Syntax: describe(iris$sepal.Length)

**Pacman Package:** The [pacman](http://trinker.github.io/pacman_dev/) package is an R package management tool that combines the functionality of base library related functions into intuitively named functions.

Function names in the pacman package follow the format of p\_xxx where 'xxx' is the task the function performs. For instance, the p\_load function allows the user to load one or more packages as a more generic substitute for the library or require functions and if the package isn't available locally it will install it for you.

**Installation:** install.packages(“pacman”)

| **pacman Function** | **Base Equivalent** | **Description** |
| --- | --- | --- |
| p\_load | install.packages + library | Load and Install Packages |
| p\_install | install.packages | Install Packages from CRAN |
| p\_load\_gh | NONE | Load and Install GitHub Packages |
| p\_install\_gh | NONE | Install Packages from GitHub |
| p\_install\_version | install.packages & packageVersion | Install Minimum Version of Packages |
| p\_temp | NONE | Install a Package Temporarily |
| p\_unload | detach | Unload Packages from the Search Path |
| p\_update | update.packages | Update Out-of-Date Packages |

We will have to install pacman to use the describe() function. After installing the package, we will load it by using the syntax:

p\_load(pacman)

To load describe() function, we will first load the psych package using:

p\_load(psych)

After loading psych package, we can use the describe() function.

**Example:** hist(iris$petal.Lenght[iris$species == “virginica” & iris$Petal.length < 5.5], main = “ Title”) #This command will make a histogram of petal length values less than 5.5 of virginica species.

**Defining a Function:**

We define the function (ex: moles) by assigning it to the output of function. The list of argument names is contained within parentheses. Next, the body of the function – the statements that are executed when it runs – is contained within curly braces {}. The statement in the body is indented by two spaces, which makes the code easier to read but does not affect how the code operates.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. Inside the function, we use a return statement to send a result back to whoever asked for it.

Example:

Mole\_conversion <- function(mass , mol\_weight)

{

moles <- mass/mol\_weight

Return(moles)

}

Note: In R, it is not necessary to include the return statement. R automatically returns whichever variable is on the last line of the body of the function.

**Composing Functions:**

We define basic operations, then combine them in ever-larger chunks to get the effect we want.

Example: Finding the molarity:

Mole\_conversion <- function(mass , mol\_weight) #defining one function.

{ moles <- mass/mol\_weight

Return(moles)

}

molarity <- function(moles , volume) #defining second function.

{ M = moles/volume

Return(M)

}

Molarity\_1 <- function(mass , mol\_weight, volume)

# composing function by combining the predefined functions.

{ moles <- mole\_conversion(mass, mol\_weight)

Answer <-molarity(moles, volume)

Return(answer)

}

**Nesting Functions:**

Nesting one function inside another.

time\_in\_hour(time\_in\_minute(55)): the output of the function time\_in\_minute will be served as the input for the function time\_in\_hour.

Note: If the variable v refers to a vector, then v[1] is the vector’s first element and v[length(v)] is its last (the function length returns the number of elements in a vector).

Functions can accept arguments explicitly assigned to a variable name in the function call functionName(variable = value), as well as arguments by order.

**all.equal function:** For comparing two objects allowing for rounding errors. If the two variables are equal, the output will be true else the output will be false.

**na.rm function:** na.rm = TRUE, will remove the NA values from the calculations.

**As.factor:** Converts a column from numeric to factor.

**As.character:** attempts to coerce (using force) its argument to **character** type.

**sample():**sample takes a sample of the specified size from the elements of x using either with or without replacement.

sample(x, size, replace = FALSE, prob = NULL)

sample.int(n, size = n, replace = FALSE, prob = NULL, useHash = (!replace && is.null(prob) && size <= 2="" n="" &&=""> 1e7))

##### Arguments

**x:** either a vector of one or more elements from which to choose, or a positive integer. See ‘Details.’

**n:** a positive number, the number of items to choose from. See ‘Details.’

**size:** a non-negative integer giving the number of items to choose.

**replace:** should sampling be with replacement?

**prob:** a vector of probability weights for obtaining the elements of the vector being sampled

**useHash**: [logical](https://www.rdocumentation.org/link/logical?package=base&version=3.6.2) indicating if the hash-version of the algorithm should be used. Can only be used for replace = FALSE, prob = NULL, and size <= n/2, and really should be used for large n, as useHash=FALSE will use memory proportional to n.

**Intersect():** Calculates the intersection of subsets of a probability space. Comparisons are made row-wise, so that in the data frame case, intersect(A,B) is a data frame with those rows that are both in A and in B.

Syntax: intersect(x,y,…)

**x, y**

vectors, data frames, or ps objects containing a sequence of elements (conceptually).

**…**

further arguments to be passed to or from other methods.

**Save():** The function save() can be used to save one or more R objects to a specified file (in .RData or .rda file formats). The function can be read back from the file using the function load().

**Syntax:** save(data\_frame,file = “data.rda”

**Load():** Reload datasets written with the function save.

Syntax: load(file, envir = parent.frame(), verbose = FALSE)

**union(): union()**function in [R Language](https://www.geeksforgeeks.org/introduction-to-r-programming-language/) is used to combine the data of two objects. This function takes two objects like Vectors, dataframes, etc. as arguments and results in a third object with the combination of the data of both the objects.

**duplicated():** 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.

**Syntax**: duplicated(dataframe)

**ls():** To know what objects are currently defined in the workspace, you can use the ls() function.

**DATAFRAMES:** Data frames are generic data objects of R, used to store tabular data.

**Creating dataframes :**

Piece of code:

vec1 <- c(1,2,3)

vec2 <- c(4,5,6)

vec3 <- c(7,8,9)

df <- data.frame(vec1,vec2,vec3)

print(df)

data.frame is the key function for the dataframe creation.

A dataframe can also be created by reading data from a file using the following command:

newDf <- read.table(path=”path of the file”)

In the path, use “/” instead of “\”. A separator can also be used to distinguish between entries. Default separator is space, ’ ’.

newDf <- read.table(file=”Path of the file”, sep =” ”)

**Accessing rows and columns:** df[val1,val2] refers to row “val1”, column”val2”. Can be number or string.

“val1” or “val2” can also be array of values like “1:2” or “c(1,3)”

df[val2] (no commas) – just refers to column “val2” only.

**Subset()** which extracts subset of data based on conditions.

Example:

pd <- data.frame(“Name”=c(“Senthil”,”Senthil”,”Sam”,”Sam”),”Month”=c(“Jan”,”Feb”,”Jan”,”Feb”),”BS”=c(141.2,139.3,135.2,160.1), “BP”=c(90,78,80,81))

pd2 <- subset(pd,Name==”Senthil”| BS>150)

print(“new subset pd2”)

print(pd2)

**Editing dataframes:** Dataframes can be edited by direct assignment.

Example:

vec1 <- c(1,2,3)

vec2 <- c(4,5,6)

vec3 <- c(7,8,9)

df <- data.frame(vec1,vec2,vec3)

print(df)

df[[2]][3] <- 10 #(The second column of the third row will be changed to 10)

print(df)

A dataframe can also be edited using the edit() command.

Example:

vec1 <- c(1,2,3)

vec2 <- c(4,5,6)

vec3 <- c(7,8,9)

df <- data.frame(vec1,vec2,vec3)

print(df)

df <- edit(df) # A table will be open and we can replace any values we want.

print(df)

**Adding extra rows and columns:**

Extra row can be added with “rbind” function and extra column with “cbind”.

Example:

vec1 <- c(1,2,3)

vec2 <- c(4,5,6)

vec3 <- c(7,8,9)

df <- data.frame(vec1,vec2,vec3)

df <- rbind(df,data.frame(vec1 = 10, vec2 = 11, vec3 = 12))

df <- cbind(df,vec4=c(13,14,15,16))

print(df)

**Deleting Rows and Columns:** Examples:

vec1 <- c(1,2,3)

vec2 <- c(4,5,6)

vec3 <- c(7,8,9)

df <- data.frame(vec1,vec2,vec3)

df <- df[-3,-1] # Delete the 3rd row and 1st column.

print(df)

# Conditional deletion:

vec1 <- c(1,2,3)

vec2 <- c(4,5,6)

vec3 <- c(7,8,9)

df <- data.frame(vec1,vec2,vec3)

df <- df[!df$vec1==3,] # This command will delete the entire row vec1 carrying 3.

print(df)

**Manipulating Rows ­­­­­­­­– the factor issue:** When character columns are created in a data.frame, they become factors. Factor variables are those where the character column is split into categories or factor levels.

Example:

df[3,3] = “Others” # This will give us the NA value as currently “Others” is invalid factor level.

print(df)

To resolve this issue, we must false the stringAsFactors value at the time of the formation of dataframe.

Example:

vec1 <- c(1,2,3)

vec2 <- c(4,5,6)

vec3 <- c(7,8,9)

df <- data.frame(vec1, vec2, vec3, stringsAsFactors = FALSE)

df[3,3] = “Others”

print(df)

**Recasting dataframes:** Recasting is the process of manipulating a data frame in terms of its variable. Recasting is used for the reshaping of the data which could bring more insight on the data when seen from different perspective.

Example:

#Creating of a dataframe:

**pd = data.frame("Name"=c("Senthil","Senthil","Sam","Sam"),**

**"Month"=c("Jan","Feb","Jan","Feb"),**

**"BS"=c(141.2,139.3,135.2,160.1),**

**"BP"=c(90,78,80,81))**

# Recast the dataframes through two steps: Melt and Cast. To use the Melt and Cast commands, firstly we must identify the identifier variables (Discrete type variables) and measurements variables (numeric variables). Here in the given dataframe, the identifier variables are Name and month, and the measurement variables are BS and BP. Categorical and Date variables cannot be measurements variables.

# Melt Command

**library(reshape2)** # reshape2 library should be preinstalled using install.packages(“reshape2”)

# Syntax of melt function is **melt(data, id.vars, measure.vars, variable.name = “variable”, value.name = “value”)**

**df = melt(pd, id.vars = c("Name","Month"), measure.vars = c("BS","BP"))**

**print(df)** # melt function keeps the identifier variables as it is and stores the measurement variables under a separate column variable and stores their values under a separate column value.

Name Month variable value

1 Senthil Jan BS 141.2

2 Senthil Feb BS 139.3

3 Sam Jan BS 135.2

4 Sam Feb BS 160.1

5 Senthil Jan BP 90.0

6 Senthil Feb BP 78.0

7 Sam Jan BP 80.0

8 Sam Feb BP 81.0 (Output of melt function)

# Cast step: We will be applying the dcast() function and its syntax is **dcast(data,formula,value.var=col.with values)**

**df2= dcast(df,variable+Month~Name,value.var="value")**

**print(df2)** # variable+month~Name means that columns “variable” and “month” will remain as is and categories in column “Name” become new variables.

# value.var=”value”, column of df from which the values are to be taken from.

variable Month Sam Senthil

1 BS Feb 160.1 139.3

2 BS Jan 135.2 141.2

3 BP Feb 81.0 78.0

4 BP Jan 80.0 90.0 (output of cast function)

**Recasting in single step:** Applying the recast() function performs melt and cast in one command.

Syntax: **recast(data,formula,…,id.var,measure.var)**

Example:

**recast(pd,variable+Month~Name,id.var=c("NAME","Month"))**

# variable+Month~Name (Parameter refers to the “cast” section of the command)

# id.var=c(**"**Name**"**,"Month") (Parameter refers to the "melt" section of the command.

variable Month Sam Senthil

1 BS Feb 160.1 139.3

2 BS Jan 135.2 141.2

3 BP Feb 81.0 78.0

4 BP Jan 80.0 90.0 (Output of recast function)

**Add new variable to dataframe based on existing ones:**

Call the library ‘dplyr’ command using the library() command.

mutate() command will add extra variable columns based on existing ones.

Example:

dat <- data.frame("Name"= c("Ram","Shayam","Sita","Amar"),

"Maths\_Marks"=c(84,75,86,67),

"Science\_Marks"=c(87,56,79,82),

"English\_Marks"=c(88,85,90,95))

library(dplyr)

dat2 <- mutate(dat,Percentage=((Maths\_Marks+Science\_Marks+English\_Marks)/3))

print(dat2)

# Multiple variables can be created as transformation of old variable.

# Here Percentage is the new variable column.

**Joining of two frames:** The common syntax for “dplyr” functions used to combine dataframes:

**function(dataframe1,dataframe2,by=id.variable)**

# The “id.variable” is common to both dataframes.

# This variable provides the identifiers for combining the 2 dataframes.

# The nature of combination depends on the function to be used.

# Call the library ‘dplyr’ command using the library() command.

# The following commands would be used to combine datasets:

1. left\_join() b) right\_join() c) inner\_join() d) full\_join() e) semi\_join() f) anti\_join()

**left\_join():** Joins matching rows of dataframe2 to dataframe1 based on the id.variable.

Example:

# Creating one dataframe

dat <- data.frame("Name"= c("Ram","Shayam","Sita","Amar"),

"Maths\_Marks"=c(84,75,86,67),

"Science\_Marks"=c(87,56,79,82),

"English\_Marks"=c(88,85,90,95))

# Creating second dataframe

dat2 <- data.frame("Name"= c("Ram","Shayam","Amar"),

"School"=c("D.P.S.", "J.N.V.", "A.P.S."))

# In the example, only “Ram”, “Shayam” and “Amar” from id.variable “Name” are present in “dat” which is dataframe1.

# Only these three IDs and corresponding values in “dat2” will be merged with “dat”.

# The variable “School” from “dat2” would be merged to “dat”.

library(dplyr)

dat\_left\_join <- left\_join(dat,dat2,by="Name")

print(dat\_left\_join)

Name Maths\_Marks Science\_Marks English\_Marks School

1 Ram 84 87 88 D.P.S.

2 Shayam 75 56 85 J.N.V.

3 Sita 86 79 90 <NA>

4 Amar 67 82 95 A.P.S. (Output)

**right\_join():** Joins matching rows of dataframe1 to dataframe2 based on the id.variable.

Example:

dat <- data.frame("Name"= c("Ram","Shayam","Sita","Amar"),

"Maths\_Marks"=c(84,75,86,67),

"Science\_Marks"=c(87,56,79,82),

"English\_Marks"=c(88,85,90,95))

dat2 <- data.frame("Name"= c("Ram","Shayam","Amar"),

"School"=c("D.P.S.", "J.N.V.", "A.P.S."))

library(dplyr)

dat\_right\_join <- right\_join(dat,dat2,by="Name")

print(dat\_right\_join)

Name Maths\_Marks Science\_Marks English\_Marks School

1 Ram 84 87 88 D.P.S.

2 Shayam 75 56 85 J.N.V.

3 Amar 67 82 95 A.P.S. (Output)

**inner\_join():** Merges and retains those rows with IDs present in both dataframes.

Example:

dat <- data.frame("Name"= c("Ram","Shayam","Sita","Amar"),

"Maths\_Marks"=c(84,75,86,67),

"Science\_Marks"=c(87,56,79,82),

"English\_Marks"=c(88,85,90,95))

dat2 <- data.frame("Name"= c("Ram","Shayam","Amar"),

"School"=c("D.P.S.", "J.N.V.", "A.P.S."))

library(dplyr)

dat\_inner\_join <- inner\_join(dat,dat2,by="Name")

print(dat\_inner\_join)

Name Maths\_Marks Science\_Marks English\_Marks School

1 Ram 84 87 88 D.P.S.

2 Shayam 75 56 85 J.N.V.

3 Amar 67 82 95 A.P.S. (Output)

**full\_join():** Returns all rows and all columns from both dataframe1 and dataframe2. Where there are not matching values, returns NA for the one missing.

Example:

dat <- data.frame("Name"= c("Ram","Shayam","Sita","Amar"),

"Maths\_Marks"=c(84,75,86,67),

"Science\_Marks"=c(87,56,79,82),

"English\_Marks"=c(88,85,90,95))

dat2 <- data.frame("Name"= c("Ram","Shayam","Amar"),

"School"=c("D.P.S.", "J.N.V.", "A.P.S."))

library(dplyr)

dat\_full\_join <- full\_join(dat,dat2,by="Name")

print(dat\_full\_join)

Name Maths\_Marks Science\_Marks English\_Marks School

1 Ram 84 87 88 D.P.S.

2 Shayam 75 56 85 J.N.V.

3 Sita 86 79 90 <NA>

4 Amar 67 82 95 A.P.S. (Output)

**semi\_join:** Returns all rows from dataframe1 where there are matching values in dataframe2, keeping just columns from dataframe1.

A semi join differs from an inner join because an inner join will return one row of dataframe1 for each matching row of dataframe2, where a semi join will never duplicate rows of dataframe1.

Example:

dat <- data.frame("Name"= c("Ram","Shayam","Sita","Amar"),

"Maths\_Marks"=c(84,75,86,67),

"Science\_Marks"=c(87,56,79,82),

"English\_Marks"=c(88,85,90,95))

dat2 <- data.frame("Name"= c("Ram","Shayam","Amar"),

"School"=c("D.P.S.", "J.N.V.", "A.P.S."))

library(dplyr)

dat\_semi\_join <- semi\_join(dat,dat2,by="Name")

print(dat\_semi\_join)

Name Maths\_Marks Science\_Marks English\_Marks

1 Ram 84 87 88

2 Shayam 75 56 85

3 Amar 67 82 95 (Output)

**anti\_join():** Return all rows from dataframe1 where there are not matching values in dataframe2, keeping just columns from dataframe1.

Example:

dat <- data.frame("Name"= c("Ram","Shayam","Sita","Amar"),

"Maths\_Marks"=c(84,75,86,67),

"Science\_Marks"=c(87,56,79,82),

"English\_Marks"=c(88,85,90,95))

dat2 <- data.frame("Name"= c("Ram","Shayam","Amar"),

"School"=c("D.P.S.", "J.N.V.", "A.P.S."))

library(dplyr)

dat\_anti\_join <- anti\_join(dat,dat2,by="Name")

print(dat\_anti\_join)