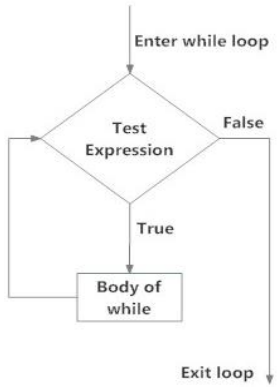
WHILE REPEAT LOOP – PPT 211

A while loop in R is a close cousin of the for loop in R. However, a while loop will check a logical condition, and keep running the loop as long as the condition is true. Here's what the syntax of a while loop looks like:

```
while (condition) {
        expression
}
```

Flowchart of while Loop



If the condition in the while loop in R is *always* true, the while loop will be an infinite loop, and our program will never stop running.

SAMPLE APPLICATION OF WHILE LOOPING STATEMENT IN RSTUDIO

1.
```
i <- 1
while (i < 6) {
print(i)
i = i+1
}
```

2. val **=** 1
   # using while loop
   while (val <**=** 5)
   {
   # statements
   print(val)
   val **=** val **+** 1
   }

3. v <- c("Hello","while loop")
   count <- 1
   while (count < 7) {
   print(v)
   count = count + 1
   }

This sample will create a loop and after each run add 1 to the stored variable. You need to close the loop, therefore it explicitly tells R to stop looping when the variable reached 10.

Note: If you want to see current loop value, you need to wrap the variable inside the function print().

Example:

begin <- 1

#Create the loop

while (begin <= 10){

```
  #See which we are
  cat('This is loop number',begin)

  begin <- begin+1
  print(begin)
}
```

An Armstrong number, also known as narcissistic number, is a number that is equal to the sum of the cubes of its own digits.

Example:

```
# take input from the user
num = as.integer(readline(prompt="Enter a number: "))
# initialize sum
sum = 0
# find the sum of the cube of each digit
temp = num
while(temp > 0) {
  digit = temp %% 10
  sum = sum + (digit ^ 3)
  temp = floor(temp / 10)
}
# display the result
if(num == sum) {
  print(paste(num, "is an Armstrong number"))
} else {
  print(paste(num, "is not an Armstrong number"))
}
```

Using an if-else Statement within a while loop in R

Example:

```
wins <-0
while (wins <=10){
  if (wins < 10){
    print("does not make playoffs")
  } else {
    print("Makes Playoffs")
  }
  wins <- wins +1
}
```

Program to calculate factorial of a number using WHILE STATEMENT.

Example:

```
n <- 5
# assigning the factorial variable
# and iteration variable to 1
factorial <- 1
i <- 1
```

Example 1:

```
# using while loop
while (i <= n)
{
    # multiplying the factorial variable
    # with the iteration variable
    factorial = factorial * i
```

Example 2:

```
    # incrementing the iteration variable
    i = i + 1
}
# displaying the factorial
print(factorial)
```

The **Repeat loop** executes the same code again and again until a stop condition is met.

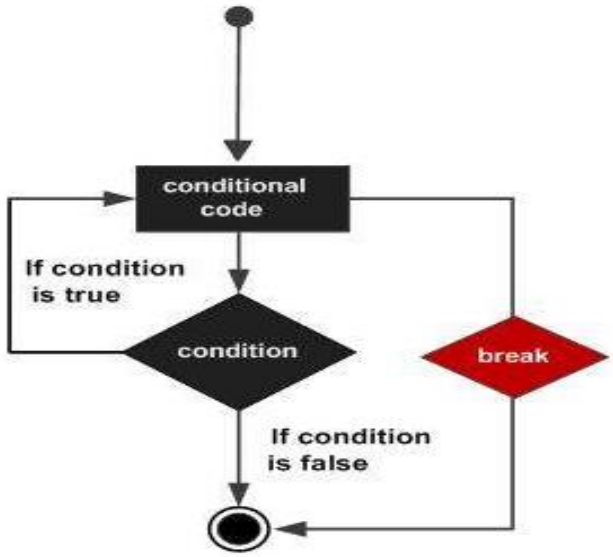
It is a simple loop that will run the same statement or a group of statements repeatedly until the stop condition has been encountered. Repeat loop does not have any condition to terminate the loop, a programmer must specifically place a condition within the loop's body and use the declaration of a break statement to terminate this loop.

The basic syntax for creating a repeat loop in R:

```
repeat {
        commands
        if(condition) {
          break
```

```
        }
}
```

FLOW DIAGRAM (REPEAT LOOP)



Sample code applying Repeat Loop

```
v <- c("Hello","loop")
rpt <- 2
repeat {
   print(v)
   rpt <- rpt+1
   if(rpt > 5) {
         break
   }
}
```

Example 1:

```
val = 1
repeat
{
    print(val)
    val = val + 1

    # checking stop condition
    if(val > 5)
    {
        break
    }
}
```

Example 2:

```
i <- 0
# using repeat loop
repeat
{
    print("Go RTU!")
    i = i + 1
    # checking the stop condition
    if (i == 5)
    {
        break
    }
}
```

FUNCTIONS IN R – PPT 212

A **function** is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a **function** is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

R has Two (2) types of Function

1) **in-built** functions which can be directly called in the program without defining them first.
2) **user defined** function. - We can also create and use our own functions referred as Built-in Function

Simple examples of in-built functions are

seq(), **mean()**,

max(), **sum(x)** and **paste(...)** etc.

They are directly called by user written programs. You can refer most widely used R functions.

Example:

```
# Create a sequence of numbers from 32 to 44.

print(seq(32,44))

# Find mean of numbers from 25 to 82.

print(mean(25:82))

# Find sum of numbers frm 41 to 68.

print(sum(41:68))
```

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions.

Example:

```
# Create a function to print squares of numbers in sequence.

new.function <- function(a) {
        for(i in 1:a) {
                b <- i^2
                print(b)
        }
}

#calling the function

new.function(6)
```

Function Components

The different parts of a function are:

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

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as

defined in the function or they can be supplied in a different sequence

but assigned to the names of the arguments.

Example:

```
# Create a function with arguments.
new.function <- function(r,t,u) {
    result <- r * t + u
    print(result)
}
# Call the function by position of arguments.
new.function(5,4,12)
# Call the function by names of the arguments.
new.function(r = 12, t = 5, u = 4)
```

Calling a Function with Default Argument

R can also call such functions by supplying new values of the argument and get non default result.

Example:

```
# Create a function with arguments.
new.function <- function(a = 3, b = 6) {
    result <- a * b
    print(result)
}
# Call the function without giving any argument.
new.function()
# Call the function with giving new values of the argument.
new.function(9,5)
```

Example 1:

```
readinteger <- function()
{
        n <- readline(prompt="Enter an integer: ")
        return(as.integer(n))
}
print(readinteger())
```

Example 2:

```
countdown <- function(from)
{
        print(from)
        while(from!=0)
        {
                Sys.sleep(1)
                 from <- from - 1
                print(from)
        }
}
countdown(5)
```

Celsius into Kelvin:

```
celsius_to_kelvin <- function(temp_C) {
        temp_K <- temp_C + 273.15        return(temp_K)
}
celsius_to_kelvin(0)
```

Fahrenheit_to_celsius

```
fahrenheit_to_celsius <- function(temp_F) {
  temp_C <- (temp_F - 32) * 5 / 9
  return(temp_C)
}
#fahrenheit_to_celsius(6)
```

Fahrenheit_to_kelvin

```
fahrenheit_to_kelvin <- function(temp_F) {
  temp_C <- fahrenheit_to_celsius(temp_F)
  temp_K <- celsius_to_kelvin(temp_C)
  return(temp_K)
}
# freezing point of water in Kelvin
fahrenheit_to_kelvin(32.0)
```

DATA INTERFACES IN R - PPT 213

Introduction & CSV Files

Flat files

states.csv

```
state,capital,pop_mill,area_sqm
South Dakota,Pierre,0.853,77116
New York,Albany,19.746,54555
Oregon,Salem,3.970,98381
Vermont,Montpelier,0.627,9616
Hawaii,Honolulu,1.420,10931
```

	state	capital	pop_mill	area_sqm
1	South Dakota	Pierre	0.853	77116
2	New York	Albany	19.746	54555
3	Oregon	Salem	3.970	98381
4	Vermont	Montpelier	0.627	9616
5	Hawaii	Honolulu	1.420	10931

CSV Files

R can read and write data from files stored outside the R environment. We can also store and accessed by the operating system. R can read and write into various file formats like csv, excel, xml etc.

Getting and Setting the Working Directory

 # Get and print current working directory.

print(getwd())

 # Set current working directory.

setwd(“C:/web/Sample")

 # Get and print current working directory.

print(getwd())

Input as CSV File

You can create this file using windows notepad by copying and pasting this data.

Save the file as **employee3.csv** using the save As All files(*.*) option in notepad.

id,name,salary,start_date,dept

1,Rick,623.3,2012-01-01,IT

2,Dan,515.2,2013-09-23,Operations

3,Michelle,611,2014-11-15,IT

4,Ryan,729,2014-05-11,HR

5,Gary,843.25,2015-03-27,Finance

6,Nina,578,2013-05-21,IT

7,Simon,632.8,2013-07-30,Operations

8,Guru,722.5,2014-06-17,Finance

Reading a CSV File

Following is a simple example of **read.csv()** function

to read a CSV file available in your current working directory −

```
data <-read.csv("employee2.csv")
print(data)
```

Output:

```
      id,   name,    salary,   start_date,     dept
1      1    Rick     623.30    2012-01-01      IT
2      2    Dan      515.20    2013-09-23      Operations
3      3    Michelle 611.00    2014-11-15      IT
4      4    Ryan     729.00    2014-05-11      HR
5     NA    Gary     843.25    2015-03-27      Finance
6      6    Nina     578.00    2013-05-21      IT
7      7    Simon    632.80    2013-07-30      Operations
8      8    Guru     722.50    2014-06-17      Finance
```

Analyzing the CSV File

By default the **read.csv()** function gives the output as a data frame. This can be easily checked as follows. Also we can check the number of columns and rows.

```
data <- read.csv("employee.csv")
print(is.data.frame(data))
print(ncol(data))
print(nrow(data))
```

Get the maximum salary

```
# Create a data frame.
data <- read.csv("employee.csv")
# Get the max salary from data frame.
sal <- max(data$salary)
print(sal)
```

Get the details of the person with max salary

We can fetch rows meeting specific filter criteria similar to a SQL where clause.

```
# Create a data frame.
data <- read.csv(“employee.csv")
# Get the max salary from data frame.
sal <- max(data$salary)
# Get the person detail having max salary.
retval <- subset(data, salary == max(salary))
print(retval)
```

Get the people who joined on or after 2014

```
# Create a data frame.
data <- read.csv(“employee.csv")
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))
print(retval)
```

Writing into a CSV File

R can create csv file form existing data frame. The **write.csv()** function is used to create the csv file. This file gets created in the working directory.

```
# Create a data frame.
data <- read.csv(“employee.csv")
```

```
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))

# Write filtered data into a new file. write.csv(retval,"output.csv")

newdata <- read.csv("output.csv")

print(newdata)
```

MORE DATA INTERFACES IN R (XLSX & XML) - PPT 214

Microsoft Excel Files (.xlsx)

Microsoft Excel is the most widely used spreadsheet program which stores data in the .xls or .xlsx format. R can read directly from these files using some excel specific packages. Few such packages are - XLConnect, xlsx, gdata etc. We will be using xlsx package. R can also write into excel file using this package.

Install xlsx Package

You can use the following command in the R console to install the "xlsx" package. It may ask to install some additional packages on which this package is dependent. Follow the same command with required package name to install the additional packages.

```
install.packages("xlsx")
```

Verify and Load the "xlsx" Package

Use the following command to verify and load the "xlsx" package.

```
# Verify the package is installed.

any(grepl("xlsx",installed.packages()))

# Load the library into R workspace.

library("xlsx")
```

Output:

When the script is run we get the following output.

```
[1] TRUE

Loading required package: rJava
```

Loading required package: methods

Loading required package: xlsxjars

Reading the Excel File

The input.xlsx is read by using the **read.xlsx()** function as shown below.

The result is stored as a data frame in the R environment.

```
# Read the first worksheet in the file input.xlsx.
library(readxl)
input <- read_excel("input.xlsx")
View(input)
```

XML File (.xml)

XML is a file format which shares both the file format and the data on the World Wide Web, intranets, and elsewhere using standard ASCII text.

It stands for Extensible Markup Language (XML).

Similar to HTML it contains markup tags. But unlike HTML where the markup tag describes structure of the page, in xml the markup tags describe the meaning of the data contained into the file.

You can read a xml file in R using the "XML" package.

This package can be installed using following command.

```
install.packages("XML")
```

Input Data

Create a XMl file by copying the below data into a text editor like notepad. Save the file with a **record.xml** extension and choosing the file type as **all files(*.*)**.

Example:

```
<RECORDS>
   <EMPLOYEE>
      <ID>1</ID>
      <NAME>Rick</NAME>
      <SALARY>623.3</SALARY>
      <STARTDATE>1/1/2012</STARTDATE>
      <DEPT>IT</DEPT>
   </EMPLOYEE>

   <EMPLOYEE>
      <ID>2</ID>
      <NAME>Dan</NAME>
      <SALARY>515.2</SALARY>
      <STARTDATE>9/23/2013</STARTDATE>
      <DEPT>Operations</DEPT>
   </EMPLOYEE>

   <EMPLOYEE>
      <ID>3</ID>
      <NAME>Michelle</NAME>
      <SALARY>611</SALARY>
      <STARTDATE>11/15/2014</STARTDATE>
      <DEPT>IT</DEPT>
   </EMPLOYEE>
```

```
   <EMPLOYEE>
      <ID>4</ID>
      <NAME>Ryan</NAME>
      <SALARY>729</SALARY>
      <STARTDATE>5/11/2014</STARTDATE>
      <DEPT>HR</DEPT>
   </EMPLOYEE>

   <EMPLOYEE>
      <ID>5</ID>
      <NAME>Gary</NAME>
      <SALARY>843.25</SALARY>
      <STARTDATE>3/27/2015</STARTDATE>
      <DEPT>Finance</DEPT>
   </EMPLOYEE>

   <EMPLOYEE>
      <ID>6</ID>
      <NAME>Nina</NAME>
      <SALARY>578</SALARY>
      <STARTDATE>5/21/2013</STARTDATE>
      <DEPT>IT</DEPT>
   </EMPLOYEE>
```

```
   <EMPLOYEE>
      <ID>7</ID>
      <NAME>Simon</NAME>
      <SALARY>632.8</SALARY>
      <STARTDATE>7/30/2013</STARTDATE>
      <DEPT>Operations</DEPT>
   </EMPLOYEE>

   <EMPLOYEE>
      <ID>8</ID>
      <NAME>Guru</NAME>
      <SALARY>722.5</SALARY>
      <STARTDATE>6/17/2014</STARTDATE>
      <DEPT>Finance</DEPT>
   </EMPLOYEE>

</RECORDS>
```

Reading XML File

The xml file is read by R using the function **xmlParse()**. It is stored as a list in R.

```
# Load the package required to read XML files.
library("XML")
# Also load the other required package.
library("methods")
# Give the input file name to the function.
result <- xmlParse(file = “record.xml")
# Print the result.
print(result)
```

Get Number of Nodes Present in XML File

```
# Load the packages required to read XML files.
library("XML") library("methods")
# Give the input file name to the function.
```

```
result <- xmlParse(file = “record.xml")

# Exract the root node form the xml file.

rootnode <- xmlRoot(result)

# Find number of nodes in the root.

rootsize <- xmlSize(rootnode)

# Print the result.

print(rootsize)
```

Details of the First Node

Let's look at the first record of the parsed file. It will give us an idea of the various elements present in the top level node.

```
# Load the packages required to read XML files.

library("XML") library("methods")

# Give the input file name to the function.

result <- xmlParse(file = “record.xml")

# Exract the root node form the xml file.

rootnode <- xmlRoot(result)

# Print the result.

print(rootnode[1])
```

XML to Data Frame

To handle the data effectively in large files we read the data in the xml file as a data frame.

Then process the data frame for data analysis.

```
# Load the packages required to read XML files.

library("XML")

library("methods")

# Convert the input xml file to a data frame.

xmldataframe <- xmlToDataFrame(“record.xml")

print(xmldataframe)
```

GRAPHS IN R – PPT 215

R Programming language has numerous libraries to create charts and graphs.

A pie-chart is a representation of values as slices of a circle with different colors.

The slices are labeled and the numbers corresponding to each slice is also represented in the chart.

In R the pie chart is created using the **pie()** function which takes positive numbers

as a vector input. The additional parameters are used to control labels, color, title etc.

Syntax

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

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

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

Following is the description of the parameters used −

- **x** is a vector containing the numeric values used in the pie chart.
- **labels** is used to give description to the slices.
- **radius** indicates the radius of the circle of the pie chart.

  (value between −1 and +1).
- **main** indicates the title of the chart.
- **col** indicates the color palette.
- **clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.

SAMPLE PIE CHART

```
# Create data for the graph.
x <- c(21, 62, 10, 53)
labels <- c(“Taguig", “Mandaluyong", “Pasig", "Makati")
# Give the chart file a name.
png(file = "city.png")
# Plot the chart.
```

```
pie(x,labels)
# Save the file.
dev.off()
```

Slice Percentages and Chart Legend

We can add slice percentage and a chart legend by creating additional chart variables.

```
# Create data for the graph.
x <-  c(21, 62, 10,53)
labels <-  c("London","New York","Singapore","Mumbai")
piepercent<- round(100*x/sum(x), 1)
# Give the chart file a name.
png(file = "city_percentage_legends.jpg")
# Plot the chart.
pie(x, labels = piepercent, main = "City pie chart",col = rainbow(length(x)))
legend("topright", c("London","New York","Singapore","Mumbai"), cex = 0.8,
       fill = rainbow(length(x)))
# Save the file.
dev.off()
```

```
# Get the library.
library(plotrix)
 # Create data for the graph.
 x <- c(21, 62, 10,53)
 lbl <- c("London","New York","Singapore","Mumbai")
# Give the chart file a name.
png(file = "3d_pie_chart.jpg")
# Plot the chart.
pie3D(x,labels = lbl,explode = 0.1, main = "Pie Chart of Countries ")
```

 # Save the file.

dev.off()

A **bar chart** represents data in rectangular bars with length of the bar proportional to the value of the variable.

R uses the function **barplot()** to create bar charts. R can draw both vertical and Horizontal bars in the bar chart. In bar chart each of the bars can be given different colors.

Syntax

The basic syntax to create a bar-chart in R is −

barplot(H,xlab,ylab,main, names.arg,col)

Following is the description of the parameters used −

- **H** is a vector or matrix containing numeric values used in bar chart.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the title of the bar chart.
- **names.arg** is a vector of names appearing under each bar.
- **col** is used to give colors to the bars in the graph.

Create the data for the chart

H <- c(7,12,28,3,41)

Give the chart file a name

png(file = "barchart.png")

Plot the bar chart

barplot(H)

Save the file

dev.off()

Group Bar Chart and Stacked Bar Chart

```
# Create the input vectors.

colors = c("green","orange","brown")

months <- c("Mar","Apr","May","Jun","Jul")

regions <- c("East","West","North")

# Create the matrix of the values.

Values <- matrix(c(2,9,3,11,9,4,8,7,3,12,5,2,8,10,11), nrow = 3, ncol = 5, byrow = TRUE)

# Give the chart file a name

png(file = "barchart_stacked.png")

# Create the bar chart

barplot(Values, main = "total revenue", names.arg = months, xlab = "month", ylab = "revenue", col = colors)

# Add the legend to the chart

legend("topleft", regions, cex = 1.3, fill = colors)

# Save the file

dev.off()
```

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

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

Syntax, the basic syntax to create a line chart in R is −

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

Following is the description of the parameters used −

- **v** is a vector containing the numeric values.
- **type** takes the value "p" to draw only the points, "l" to draw only the lines and "o" to draw both points and lines.
- **xlab** is the label for x axis.

- **ylab** is the label for y axis.
- **main** is the Title of the chart.
- **col** is used to give colors to both the points and lines.

Example:

A simple line chart is created using the input vector and the type parameter as "O". The below script will create and save a line chart in the current R working directory.

```
# Create the data for the chart.
v <- c(8,12,28,3,52)
# Give the chart file a name.
png(file = "line_chart.jpg")
# Plot the bar chart.
plot(v,type = "o")
# Save the file.
dev.off()
```

```
# Create the data for the chart.
v <- c(7,12,28,3,41)
t <- c(14,7,6,19,3)
# Give the chart file a name.
png(file = "line_chart_2_lines.jpg")
# Plot the bar chart.
plot(v,type = "o",col = "red", xlab = "Month", ylab = "Rain fall", main = "Rain fall chart")
lines(t, type = "o", col = "blue")
# Save the file.
dev.off()
```

XY SCATTER

```
x <- mtcars$wt
y <- mtcars$mpg
# Plot with main and axis titles
# Change point shape (pch = 19) and remove frame.
plot(x, y, main = "Main title",
     xlab = "X axis title", ylab = "Y axis title",
     pch = 19, frame = FALSE)
# Add regression line
plot(x, y, main = "Main title",
     xlab = "X axis title", ylab = "Y axis title",
     pch = 19, frame = FALSE)
abline(lm(y ~ x, data = mtcars), col = "blue")
```

DATA SCIENCE IN R – PPT 216

Data Science is a blend of various tools, algorithms, and machine learning principles with the goal to discover hidden patterns from the raw data.

Data science is a discipline that allows you to turn raw data into understanding, insight, and knowledge.

Data Science is primarily used to make decisions and predictions making use of predictive causal analytics, prescriptive analytics (predictive plus decision science) and machine learning.

Predictive causal analytics – If you want a model that can predict the possibilities of a particular event in the future, you need to apply predictive causal analytics.

Prescriptive analytics: If you want a model that has the intelligence of taking its own decisions and the ability to modify it with dynamic parameters, you certainly need prescriptive analytics for it. This relatively new field is all about providing advice. In other terms, it not only predicts but suggests a range of prescribed actions and associated outcomes.

Predictive analytics - Say, if you are providing money on credit, then the probability of customers making future credit payments on time is a matter of concern for you. Here, you can build a model that can perform predictive analytics on the payment history of the customer to predict if the future payments will be on time or not.

Prescriptive - The best example for this is Google's self-driving car You can run algorithms on this data to bring intelligence to it. This will enable your car to take decisions like when to turn, which path to take, when to slow down or speed up.

Machine learning for making predictions — If you have transactional data of a finance company and need to build a model to determine the future trend, then machine learning algorithms are the best bet. This falls under the paradigm of supervised learning. It is called supervised because you already have the data based on which you can train your machines.

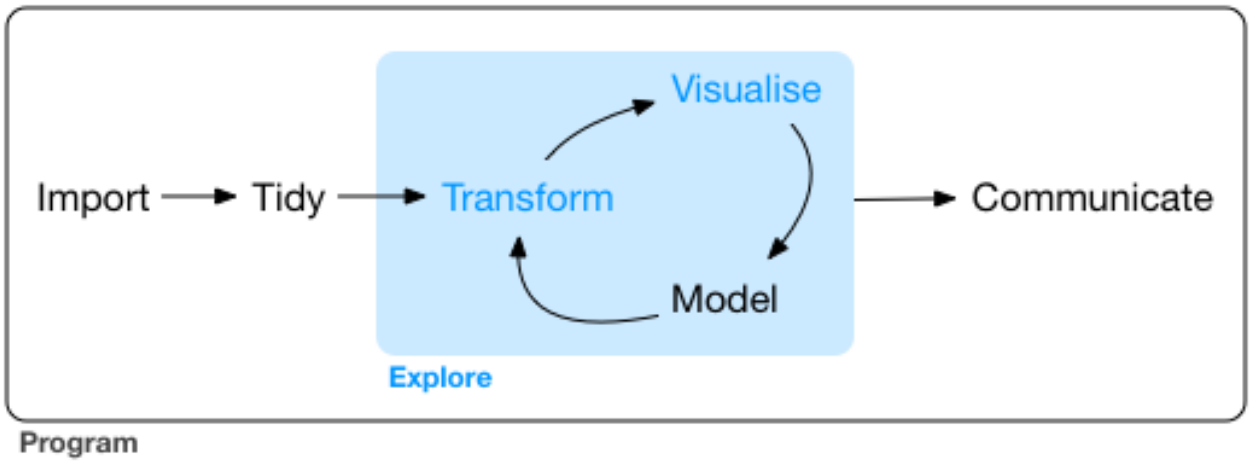
Machine learning for pattern discovery — If you don't have the parameters based on which you can make predictions, then you need to find out the hidden patterns within the dataset to be able to make meaningful predictions. This is nothing but the unsupervised model as you don't have any predefined labels for grouping. The most common algorithm used for pattern discovery is Clustering.

Machine learning for making predictions — For example, a fraud detection model can be trained using a historical record of fraudulent purchases.

Machine learning for pattern discovery — Let's say you are working in a telephone company and you need to establish a network by putting towers in a region. Then, you can use the clustering technique to find those tower locations which will ensure that all the users receive optimum signal strength.

Machine learning for making predictions —For example, a fraud detection model can be trained using a historical record of fraudulent purchases.

Machine learning for pattern discovery —
Let's say you are working in a telephone company and you need to establish a network by putting towers in a region. Then, you can use the clustering technique to find those tower locations which will ensure that all the users receive optimum signal strength.



Tidyverse

install.packages("tidyverse")

library(tidyverse)

```
The downloaded binary packages are in
        C:\Users\rey\AppData\Local\Temp\Rtmpe6EA0t\downloaded_packages
> library(tidyverse)
-- Attaching packages --------------------------------------------------------- tidyverse 1.3.1 --
v ggplot2 3.3.3     v purrr   0.3.4
v tibble  3.1.1     v dplyr   1.0.6
v tidyr   1.1.3     v stringr 1.4.0
v readr   1.4.0     v forcats 0.5.1
-- Conflicts ------------------------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

A **tibble,** or tbl_df, (pronounced as tibble diff) is a modern reimagining of the data.

tbl_df object is a data frame providing a nicer printing method, useful when working with large data sets.

SAMPLE:

```
friends_data <- data_frame(
  name = c("Nicolas", "Thierry", "Bernard", "Jerome"),
  age = c(27, 25, 29, 26),
  height = c(180, 170, 185, 169),
  married = c(TRUE, FALSE, TRUE, TRUE)
)
# Print
friends_data
```

OUTPUT:

```
> # Print
> friends_data
# A tibble: 4 x 4
  name      age height married
  <chr>   <dbl>  <dbl> <lgl>
1 Nicolas    27    180 TRUE
2 Thierry    25    170 FALSE
3 Bernard    29    185 TRUE
4 Jerome     26    169 TRUE
```

deprecation is the discouragement of use of some terminology, feature, design, or practice, typically because it has been superseded or is no longer considered efficient or safe, without completely removing it or prohibiting its use.

The mpg data frame

A data frame is a rectangular collection of variables (in the columns) and observations (in the rows). mpg contains observations collected by the US Environmental Protection Agency on 38 models of car.

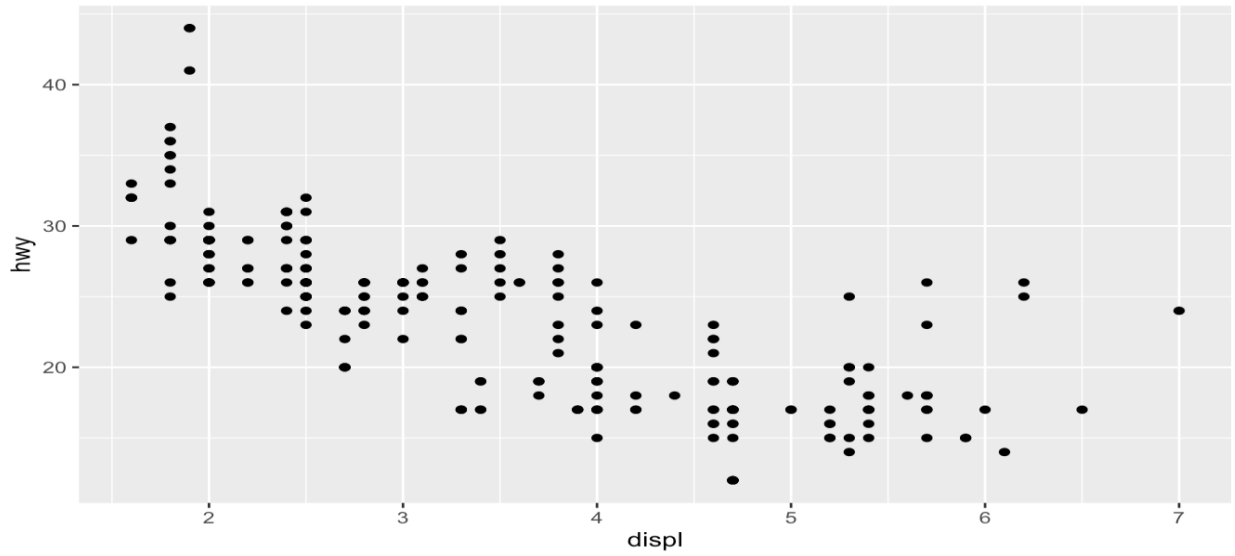
>mpg

```
# A tibble: 234 x 11
   manufacturer model      displ  year   cyl trans      drv     cty   hwy fl    class
   <chr>        <chr>      <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
 1 audi         a4           1.8  1999     4 auto(l5)   f        18    29 p     compact
 2 audi         a4           1.8  1999     4 manual(m5) f        21    29 p     compact
 3 audi         a4           2    2008     4 manual(m6) f        20    31 p     compact
 4 audi         a4           2    2008     4 auto(av)   f        21    30 p     compact
 5 audi         a4           2.8  1999     6 auto(l5)   f        16    26 p     compact
 6 audi         a4           2.8  1999     6 manual(m5) f        18    26 p     compact
 7 audi         a4           3.1  2008     6 auto(av)   f        18    27 p     compact
 8 audi         a4 quattro   1.8  1999     4 manual(m5) 4        18    26 p     compact
 9 audi         a4 quattro   1.8  1999     4 auto(l5)   4        16    25 p     compact
10 audi         a4 quattro   2    2008     4 manual(m6) 4        20    28 p     compact
# ... with 224 more rows
```

The plot shows a negative relationship between engine size (displ) and fuel efficiency (hwy). In other words, cars with big engines use more fuel.

GGPLOT -ggplot is a system for declaratively creating graphics. Provide the data, tell ggplot how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

```
ggplot(data = mpg) +
    geom_point(mapping = aes(x = displ, y = hwy))
```



The plot shows a negative relationship between engine size (displ) and fuel efficiency (hwy). In other words, cars with big engines use more fuel.

Negative correlation is a relationship between two variables in which one variable increases as the other decreases, and vice versa.

Aesthetic mappings

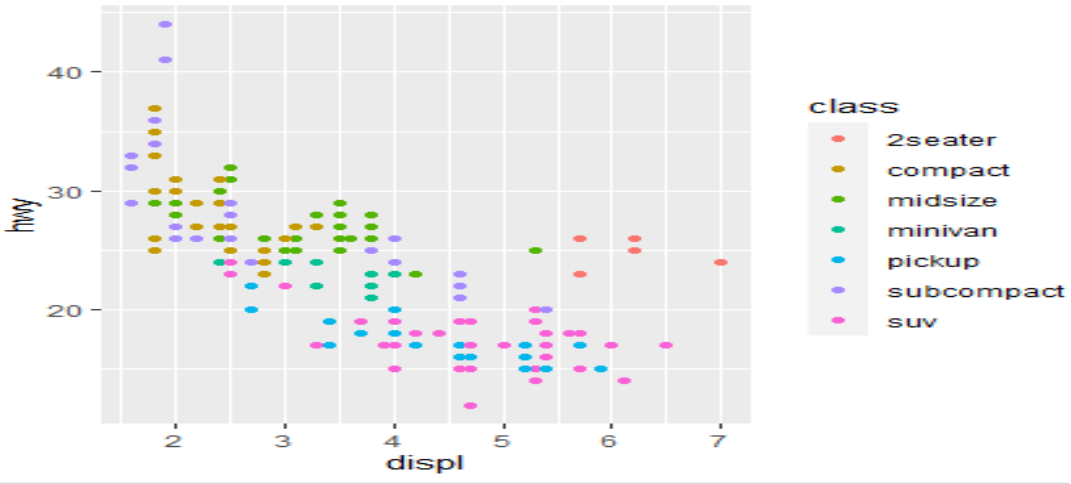
An aesthetic is a visual property of the objects in your plot. Aesthetics include things like the size, the shape, or the color of your points. You can display a point (like the one below) in different ways by changing the values of its aesthetic properties. Since we already use the word “value” to describe data, let’s use the word “level” to describe aesthetic properties

#PRIOR SCRIPT

```
ggplot(data = mpg) +
        geom_point(mapping = aes(x = displ, y = hwy))
```

#Aesthetic Mapping (color)

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



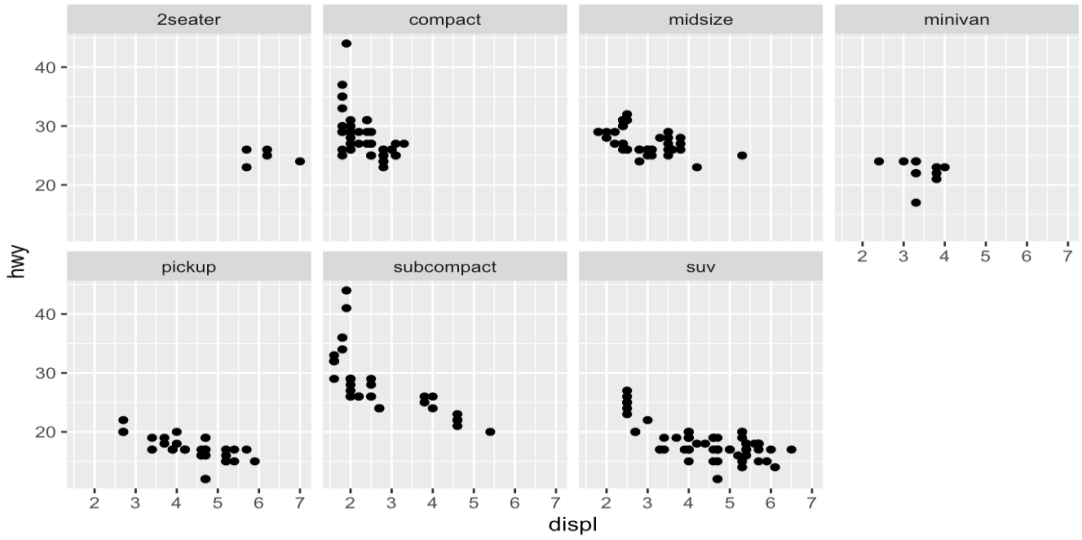
The colors reveal that many of the unusual points are two-seater cars. These cars don't seem like hybrids, and are, in fact, sports cars! Sports cars have large engines like SUVs and pickup trucks, but small bodies like midsize and compact cars, which improves their gas mileage. In hindsight, these cars were unlikely to be hybrids since they have large engines.

Facets

One way to add additional variables is with aesthetics. Another way, particularly useful for categorical variables, is to split your plot into **facets**, subplots that each display one subset of the data.

To facet your plot by a single variable, use facet_wrap(). The first argument of facet_wrap() should be a formula, which you create with ~ followed by a variable name

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```



Geometric objects that a plot uses to represent data.

A **geom** is the geometrical object

To change the geom in your plot, change the geom function that you add to ggplot().

For instance, to make the plots above, you can use this code:

```
# left
ggplot(data = mpg) +
        geom_point(mapping = aes(x = displ, y = hwy))
# right
ggplot(data = mpg) +
        geom_smooth(mapping = aes(x = displ, y = hwy))
```

OTHER DATA SETS

```
> data()
```

```
library(tidyverse)
data()
?BOD
 ggplot(data=BOD,
        mapping = aes(x = Time,
                      y = demand))+
   geom_point(size = 5)+
   geom_line(color = "red")
```

DATA VISUALIZATION – PPT 217

Data Visualization helps the organizations unleash power of their most valuable assets: their data and their people.

Pie Chart are best to use when you are trying to compare parts of a whole.

```
vtr<- c(14,28, 11 ,30,17)
name<- c("R&D","Marketing", "Corparate","Sales","Suopport")
pie(vtr,name)
```

```
vtr<- c(14,28, 11 ,30,17)
name<- c("R&D","Marketing", "Corparate","Sales","Suopport")
pie(vtr,name, col = rainbow(length(vtr)))
```

Bar Graphs are used to compare things between different groups or to track changes over time.

```
vtr<- c(14,28, 11 ,30,17)
name<- c("R&D","Marketing", "Corparate","Sales","Suopport")
barplot(vtr)
```

Boxplot are used summarize data from multiple sources and display the results in a single graph.

```
boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon", main = "Mileage Data")
```

```
main = "Mileage Data", col = c("Red", "yellow","Blue"))
```

Histogram are used to plot the frequency of score occurrences in a continuous data set that has been divided into classes called bins.

```
vtr1=c(9,11,25,70,11,3,4,5,9)
hist(vtr1)
```

Line Graph are used to track changes over short and long periods of time.

```
1  vtr1=c(9,11,25,70,11,3,4,5,9)
2
3  plot(vtr1, type ="o" )
```

P – Points

I – Lines

O – Points and Lines

Scatter plots show how much one variable is affected by another.