INTRODUCTION TO DATA ANALYSIS

Partha Padmanabhan

UCSC Silicon Valley Extension
YOUR UC IN SILICON VALLEY

>summarize/summarise: Reduces variables to value
Primarily useful with data that has been grouped by one or more variable group_by
creates the group that will be operated on
summarise uses the provided aggregation function to summarise each group

```
# group the data into daily flights >Dest<-group_by(hflights, Dest)
# group the data into daily flights
>daily <- group_by(hflights, Year, Month, DayofMonth)
# to get the number of flights per day
per_day <- summarize(daily, number_flights = n())
head(per_day)

# We have access to each of the grouping variables.
# Notice that in the summary data.frame,
# we have Year and Month as grouping variables.
# We can get the number of flights per month by summarizing as follows per_month <-
summarize(per_day, number_flights = sum(number_flights)) head(per_month)
```

```
#  Chaining #
#  There is a nice way to pass the result of one function to another.
#  This is possible because so many dplyr functions take a data
#  table as input and output another data table.
#  For example:

a1 <- group_by(hflights, Year, Month, DayofMonth)
a2 <- select(a1, Year:DayofMonth, ArrDelay, DepDelay)
a3 <- summarise(a2, arr = mean(ArrDelay, na.rm = TRUE), dep = mean(DepDelay,na.rm =
TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)
a4<-hflights %>% group_by(Year, Month, DayofMonth) %>%
select(Year:DayofMonth, ArrDelay, DepDelay) %>%
summarise(arr = mean(ArrDelay, na.rm = TRUE), dep = mean(DepDelay, na.rm = TRUE)) %>%
filter(arr > 30 | dep > 30)
```


Rounding Functions

The following functions are available for rounding numerical values:

- round() - uses IEEE standard to round up or down; optional digits= argument controls precision
- signif() - rounds numerical values to the specified digits= number of significant digits
- trunc() - rounds by removing non-integer part of number
- floor(), ceiling() - rounds to integers not greater or not less than their arguments, respectively
- zapsmall() - accepts a vector or array, and makes numbers close to zero (compared to others in the input) zero. digits= argument controls the rounding.

Summary

Data preparation is a big issue for data mining

- Data preparation includes
  - Data cleaning and data integration
  - Data reduction and feature selection
  - Discretization
- Many methods have been proposed but still an active area of research

Summarize Apply functions

We have discussed about loops. We introduced vectorized function. We will discuss some of the most used vectorized functions: the **apply** functions.

The apply family pertains to the R base package and is populated with functions to manipulate slices of data from matrices, arrays, lists and data frames in a repetitive way. These functions avoid explicit use of loop.

They act on an input list, matrix or array and apply a named function with one or several optional arguments. The called function could be:

- An aggregation function, like for example the mean, or the sum
- Other transforming or sub-setting functions
- And other vectorized functions, which return more complex strictures like list, vectors, matrices and arrays

The apply functions help to perform operations with very few lines of code. The family comprises:

apply: Apply functions over array margins
by: Apply a function to a data frame split by factors

lapply: Apply a function over list or vector

mapply: Apply a function to multiple list or vector arguments

rapply: Recursively apply a function to a list

tapply: Apply a function over a ragged array

apply: returns a vector or array or list of values obtained by applying a function to margins of an array or matrix

```
x<-matrix(c(11:30), nrow = 10, ncol =2)

> # means of the rows
> apply(x,1,mean)
[1] 16 17 18 19 20 21 22 23 24 25

> # means of the columns
> apply(x,2,mean)
[1] 15.5 25.5
```



```
> apply(x,1:2, function(x) x/2)
     [,1] [,2]
[1,] 5.5 10.5
[2,] 6.0 11.0
[3,] 6.5 11.5
[4,] 7.0 12.0
[5,] 7.5 12.5
[6,] 8.0 13.0
[7,] 8.5 13.5
[8,] 9.0 14.0
[9,] 9.5 14.5
[10,] 10.0 15.0
```

lapply: returns a list of the same length as x, each element of which is the result of applying FUN to the corresponding element of x

```
> lst <- list(x1 = 1:10, x2 = 11:20)
> lapply(lst,mean)
$x1
[1] 5.5
$x2
[1] 15.5
> lapply(lst,sum)
 $x1
[1] 55
$x2
[1] 155
```


sapply: is a user-friendly version of lapply by default returning a vector or matrix if approriate, i.e., if lapply would have returned a list with elements $x1, an d$x2, sapply will return either a vector with elements [[‘x1’]] and [[‘x2’]], or matrix with column names “x1” and “x2”

```
lst <- list(x1 = 1:10, x2 = 11:20)
> lst_mean<-sapply(lst,mean)
> lst_mean
x1 x2
5.5 15.5
> class(lst_mean)
 [1] "numeric"
```

```
data <-mtcars
mpg_category <- function(mpg){
 if (mpg >30) {
    return ("High")
}else if (mpg > 20){
   return ("Medium")
}
return ("Low")
}

lapply(X = data$mpg, FUN = mpg_category)

sapply(X = data$mpg, FUN = mpg_category)
```



```
within_range <- function(mpg, low,high){
   if (mpg >= low & mpg <= high){
    return(TRUE)
   }
    return(FALSE)
}

index <- sapply(X = data$mpg, FUN = within_range, low =
15, high = 20)
index
data[index,]
```

```
mpg_within_std_range <- function(mpg,cyl){
  if ( cyl == 4 ){
  return(within_range(mpg,low = 23, high = 31))
  }
  else if (cyl == 6){
  return(within_range(mpg, low = 18, high = 23))
  }
  return(within_range(mpg, low = 13, high = 18))
}

index <- mapply(FUN = mpg_within_std_range, mpg = data$mpg, cyl = data$cyl)
index
data[!index,]
sapply(data, FUN = median)
```


vapply: is similar to sapply, but has a pre-specified type of return, o it can be safer to use. A third argument if supplied to vapply

```
lst <- list(x1 = 1:10, x2 = 11:20)
> lst.fivenum <- vapply(lst, fivenum, c(Min.=0, "1st Qu."=0, Median=0, "3rd Qu."=0, Max.=0))

> lst.fivenum
x1 x2
Min. 1.0 11.0
1st Qu. 3.0 13.0
Median 5.5 15.5
3rd Qu. 8.0 18.0
Max. 10.0 20.0

> class(lst.fivenum)
 [1] "matrix"
```

mapply: is a multivariate version of sapply. It applies FUN to the first elements of each (...) argument, the second elements, the third elements, and so on.

```
>lst <- list(x1 = 1:10, x2 = 11:20)
> lst2 <- list(x3 = c(21:30), x4 = c(31:40))
> mapply(sum, lst$x1, lst$x2, lst2$x3, lst2$x4)
[1] 64 68 72 76 80 84 88 92 96100
```

tapply: apply a function to each cell.

Mean of Sepal length by species

```
> tapply(iris$Sepal.Length, iris$Species, mean)
 setosa versicolor virginica
5.006      5.936      6.588
```


Data Visualization

With ever increasing volume of data, it is impossible to tell stories without visualization. Data Visualization is an art of how to turn numbers into useful knowledge.

- R programming lets you learn this art by offering a set of inbuilt functions and libraries to build visualizations and present data.

- We will study the application of primary drawing functions and advanced drawing functions and will focus on understanding the methods of data exploration by visualization

– The related functions in R
– The properties of single variable
– The relationship between variables – Geographic information

Why use visualization?

•A figure is worth a thousand words

```
>df <- read.csv("CAR_DATA.csv", stringsAsFactors = FALSE)
>head(df,4)
```

```
  mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

Why Data Visualization?

Possible to make sense of large amount of data efficiently and in time
- Easy to communicate and share the insights from the data

- To suggest modeling strategies

- get to know your data!
– distributions (symmetric, normal, skewed)
– data quality problems
– outliers
– correlations and inter-relationships
– subsets of interest
– suggest functional relationships

Some basic principles

- Determine the target of visualization from the beginning
– Exploratory visualization
– Explanatory visualization

- Understanding the characteristics of the data and the audience
– Which variables are important and interesting
– Consider the role and background of the audience
– Select a proper mapping

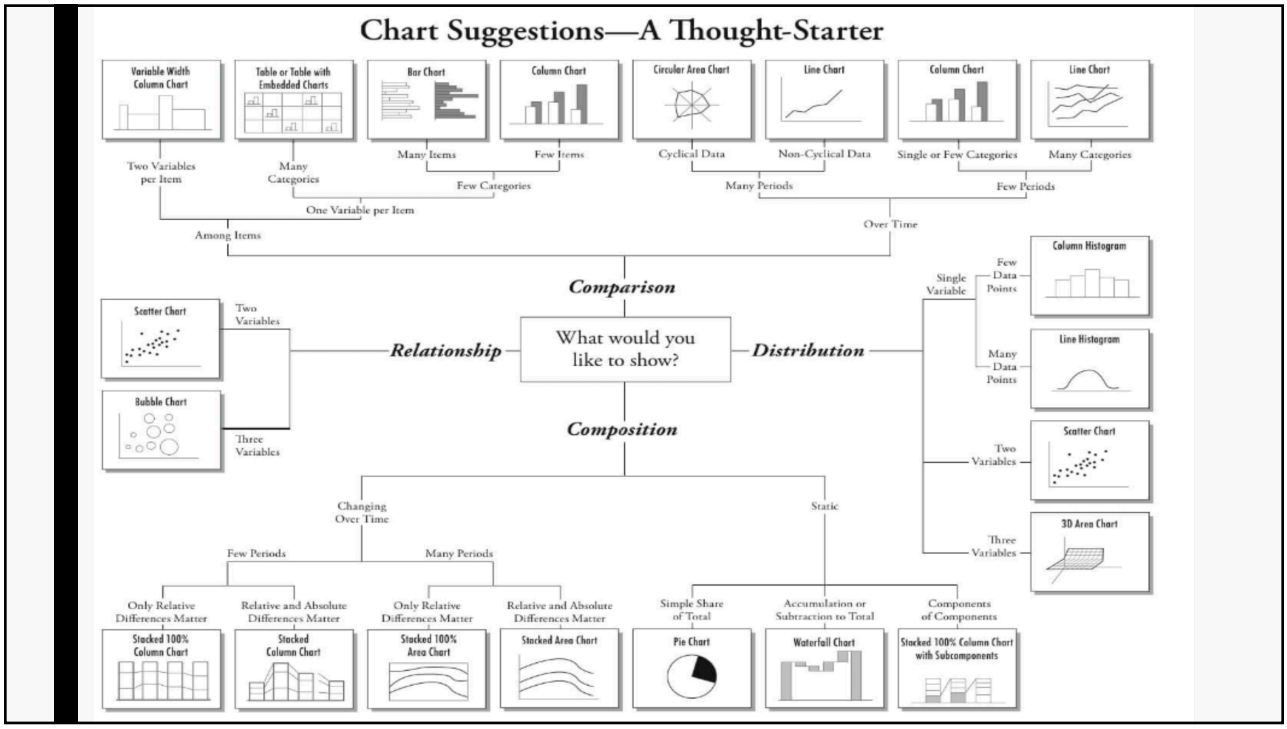
- Keep concise but give enough information

Selecting the right chart type

There are four basic chart types
– Comparison (Bar Chart..)
– Composition (Pie Chart..)
– Distribution (Bubble Chart, Heat Map..)
– Relationship (Histogram)

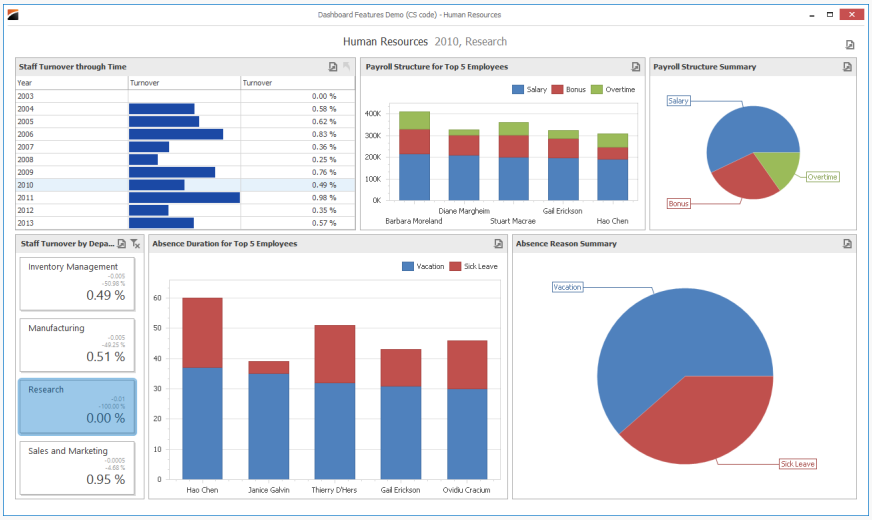
• To determine which amongst these is best suited for your data, one should answer a few questions like,

– How many variables do you want to show in a single chart?
– How many data points will you display for each variable?
– Will you display values over a period of time, or among items or groups?



Eight types of relationships between data
– Ranking comparison
– Categorical/Nominal comparison
– Time series, ordered intervals
– Proportion of the whole (contribution/composition)
– Variance/Deviation (historical or other benchmark)
– Distribution (histograms, etc.)
– Correlation (scatter plots, bubble charts, etc.)
– geoSpatial (maps with data overlays, linked to location)

Visualization Landscape



Graphics in R

In your day-to-day activities, you'll come across the below listed charts most of the time

- Line chart
- Scatter plot
- Histogram
- Bar and Stack Bar Chart
- Box Plot
- Heat Map
- Correlogram

Graphics in R

Usage of various plots

– **Line chart**: is used for continuous variable and is used to see the trend in the data set.

– **Scatter Plot**: Scatter plot is used to see the relationship between two continuous variables.

– **Histogram**: Histogram is used to plot continuous variable. It breaks the data into bins and shows frequency distribution of these bins. One can always the bin size and see the effect it has on visualization.

– **Bar and Stack Bar Chart**: Bar charts are recommended when you want to plot a categorical variable whereas stacked bar chart is an advanced version of bar chart, used for visualizing a combination of categorical variables.

Graphics in R

Usage of various plots

– **Box Plot**: Box plots are used to plot a combination of categorical and continuous variables. This plot is useful for visualizing the spread of the data and detect outliers. It shows five statistically number- the minimum, the 25th percentile (1st quartile), the median, the 75th percentile (3rd quartile) and the maximum.

– **Heat Map**: Heat map uses density of colors o display relationship between tow or more variables in a two dimensional image. It allows you to explore two dimensions as the axis and the third dimension by intensity of color.

– **Correlogram**: is used to test the level of co-relation among the variables. The cells of the matrix can be shaded or colored to show the co-relation value.

Graphics in R

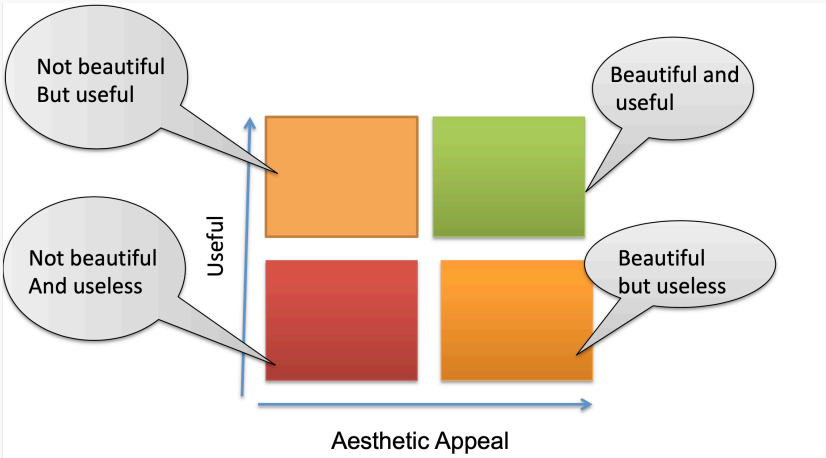
R has 3 main packages for data visualization:

- **Graphics**
  – It is part of base R and is the fundamental package for visualizing data. It has a lot of good features and we can create all the basic plots using this package.
- **ggplot2**
  – ggplot2 is a data visualization package for the statistical programming language R. Created by Hadley Wickham in 2005, ggplot2 is an implementation of Leland Wilkinson's Grammar of Graphics—a general scheme for data visualization which breaks up graphs into semantic components such as scales and layers.
- **Lattice**
  – The Lattice package is inspired by Trellis Graphics. It is a very powerful data visualization system with an emphasis on multivariable data.

Graphics in R

Function	Graph Type
plot	Scatter plots and various other like line
barplot	Bar plot (including stacked and grouped bar plots)
hist	Histograms and (relative) frequency diagrams
curve	Curves of mathematical expressions
pie	Pie charts
boxplot	Box and whisker plots

Evaluate Graphics



Idea credit: Vincent Granville

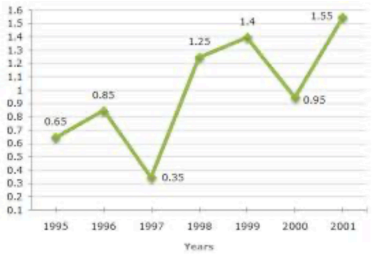
Graphics in R

We will begin exploring the plot() function. The following data will be used:

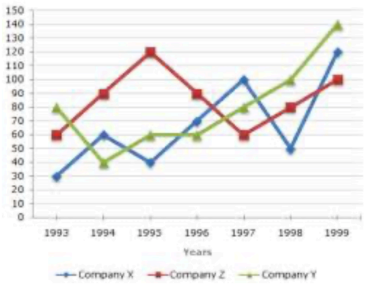
- One continuous variable
- One categorical variable
- Two continuous variables
- Two categorical variables
- One continuous and one categorical variable
- One categorical an one continuous variable

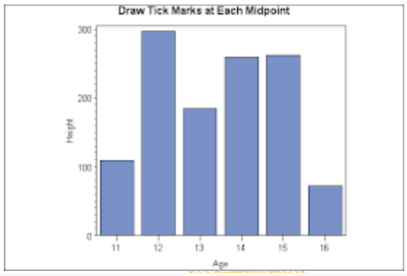
Last two cases are similar but the difference lies in the variables being assigned to X and Y axis

Graphs



Line Chart



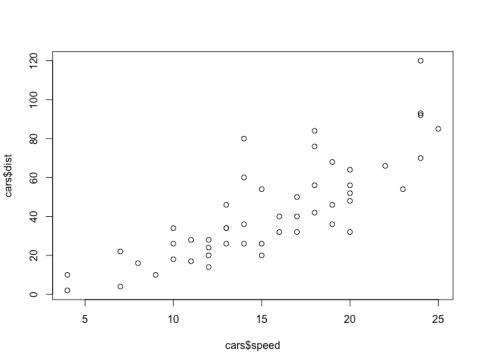


Bar Chart

Install the below visualization packages

```
install.packages("ggplot2")
install.packages(" lattice")
install.packages(" highcharter ")
install.packages(" leaflet ")
install.packages(" RColorBrewer ")
install.packages(" plotly ")
install.packages(" sunburstR ")
install.packages(" RGL ")
install.packages(" dygraphs ")
install.packages("plotrix")
install.packages("hrbrthemes")
```

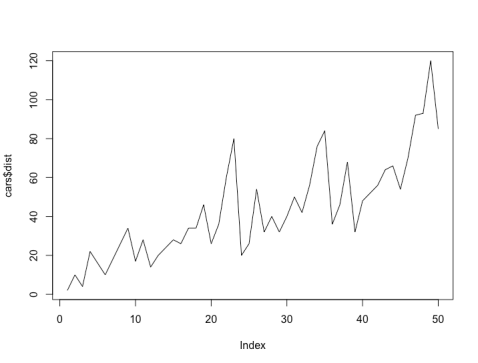
plot(cars$dist~cars$speed)



The plot() function creates a Scatter plot when a single continuous variable is used

Basic graphing functions

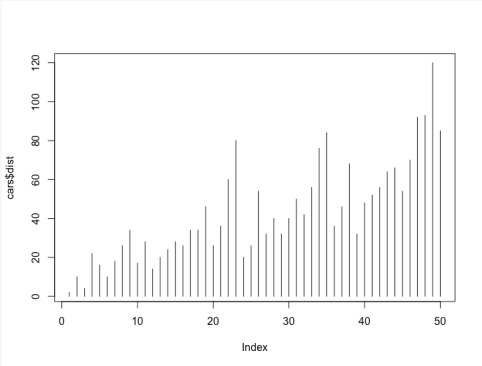
• plot(cars$dist, type = 'l')



To plot a line chart we can add type argument with 'l'

Basic graphing functions

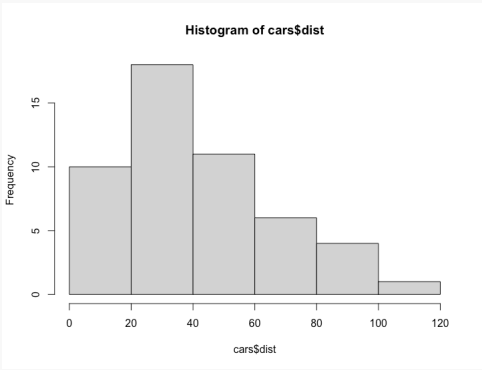
• plot(cars$dist, type = 'h')



Simple Exploratory Plots

- Usually a good idea to check your data with a quick hist()
- You can do this for any numerical column you have

hist(cars$dist)



Title and Labels

Here we will learn to enhance the plot by adding/modifying the following features

- Title
- Subtitle
- Axis Labels
- Axis Range

Title and Labels

We created plots which did not have any title or labels. One should know what the X and Y axis represent as well as the primary information being communicated by the plot. The title, axis labels and range play an important role in making the plot.

There are two ways of adding above mentioned elements to a plot

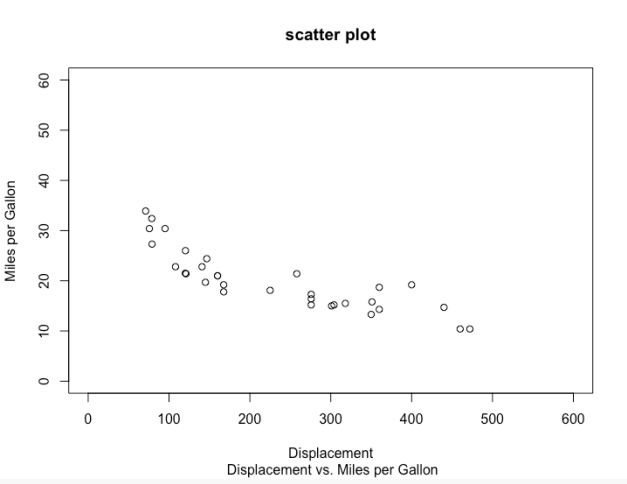
– Use the relevant arguments within the plot() function – Use the title() function

Title and Labels

Feature	Argument	Value	Example
Title	main	String	" Scattor plot"
Subtitle	sub	String	" Displacement vs. MPG"
X axis label	xlab	String	"Displacement"
Y axis Label	ylab	String	"Miles per gallon"
X axis range	xlim	Numeric Vector	c(0, 600)
Y axis range	ylim	Numeric Vector	C(0,60)

Title and Labels

plot(mtcars$disp, mtcars$mpg, main = "scatter plot",sub = " Displacement vs. Miles per Gallon", xlab = "Displacement", ylab = "Miles per Gallon", xlim = c(0,600), ylim = c(0,60))



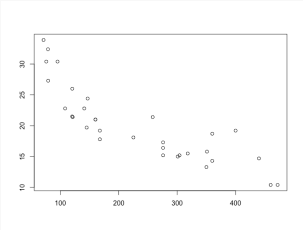
title() function

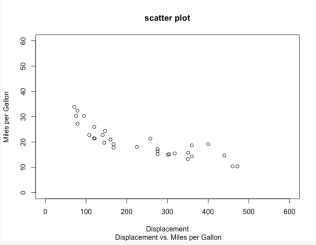
plot(mtcars$disp, mtcars$mpg, ann = FALSE)

ann = FALSE ensures that plot() function does not add the default labels.

>title(main = "scatter plot",sub = " Displacement vs. Miles per Gallon", xlab = "Displacement", ylab = "Miles per Gallon")

Note: the range of the axis can not be modified using the title() function.





color

Now we will learn to add colors to the following using the col argument:

- Plot Symbol
- Title and subtitle
- Axis
- Axis Labels
- Foreground

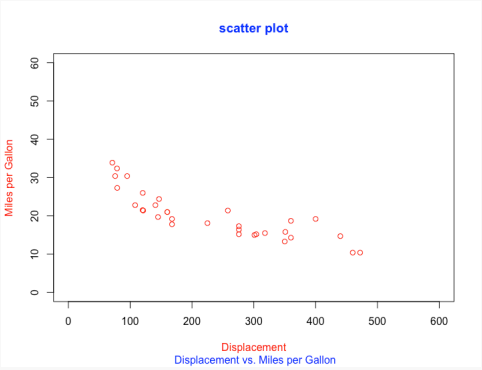
color

The col argument can be used along with other arguments to specify the the color of the title, subtitle, axes and the labels

Feature	Argument
Symbol	col
Title	col.main
Subtitle	col.sub
Axis	col.axis
Label	col.lab
Foreground	fg

color

```
plot(mtcars$disp, mtcars$mpg, col = "red", main = "scatter plot",col.main =
"blue",sub = " Displacement vs. Miles per Gallon",col.sub = "blue", xlab =
"Displacement", ylab = "Miles per Gallon", col.lab = "red",xlim = c(0,600),
ylim = c(0,60))
```



font

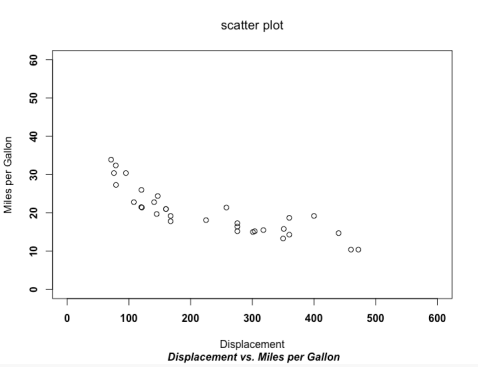
The font argument can be used to specify the font of the title, subtitle, axes and the labels.

Feature	Argument
Title	font.main
Subtitle	font.sub
Axis	font.axis
Labels	font.lab

Value	Font type
1	Plain
2	Bold
3	Italic
4	Bold Italic
5	Symbol

font

```
plot(mtcars$disp, mtcars$mpg, main = "scatter plot",font.main = 1,sub =
" Displacement vs. Miles per Gallon",font.sub = 4, xlab =
"Displacement", ylab = "Miles per Gallon", font.lab = 1,xlim = c(0,600),
ylim = c(0,60), font.axis = 2)
```



Font size

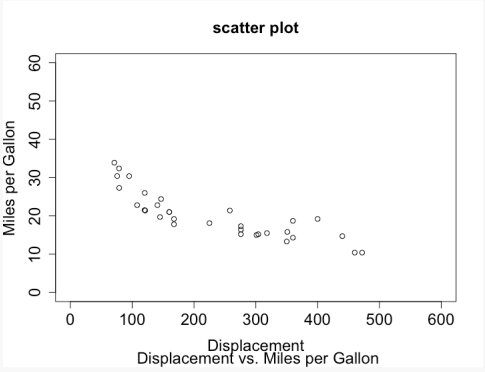
The cex argument can be used to specify the font size of the title, subtitle, axes and the labels.

The value taken by cex argument are relative to 1 i.e. the default size is represented by 1 If the value is less than 1, the size of the font will be relatively smaller, and if the value is greater than 1, the size of the font will be relatively greater, e.g. 1.5 will be 50% bigger.

Feature	Argument
Title	cex.main
Subtitle	cex.sub
Axis	cex.axis
Labels	cex.lab

Font size

```
plot(mtcars$disp, mtcars$mpg, main = "scatter plot",cex.main = 1.5,
sub = " Displacement vs. Miles per Gallon",cex.sub = 1.5,
xlab = "Displacement", ylab = "Miles per Gallon", cex.lab = 1.5,
xlim = c(0,600), ylim = c(0,60), cex.axis = 1.5)
```

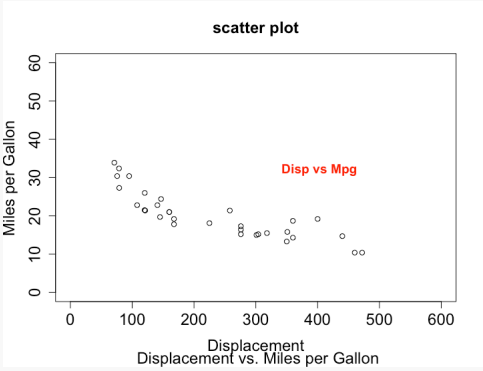


Text Annotations

The text and the mtext() function can be used to add text annotations to the plots. While the text() function places the text inside the plot, the mtext() function places the text on the margins of the plot.

Text Annotations

```
>plot(mtcars$disp, mtcars$mpg)
> # add text
>mtext( " Disp vs Mpg", col = "red", font = 2,cex = 1.2)
```

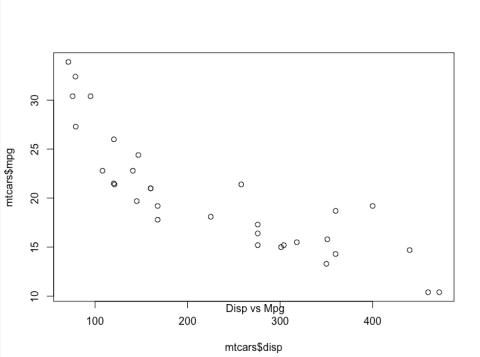


Text Annotations

The margin on which we want to place the text can be specified using the side argument. It takes 4 values from 1-4 each representing one side of the plot

>mtext(" Disp vs Mpg", side =1)

side	Margin
1	Bottom
2	Left
3	Top
4	Right



Mapping elements of a graph

- Coordinate position
- Line
- Size
- Color
- Shape
- Text
- Preferred Input data objects – Matrices and data frames – Vectors

Layout

Combine multiple graphs in a single frame using:

- par() function
- layout() function

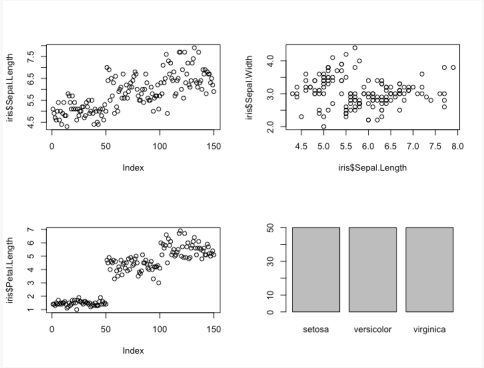
•Often it is useful to have multiple plots on the same page or frame as it allows to get a comprehensive view of a particular variable. The par() function can be used to set graphical parameters regarding plot layout using mfcol and mfrow arguments.

•The layout() function serves the same purpose but offers more flexibility by allowing us to modify the height and width of rows and columns.

Option	Description	Arguments
mfrow	Fill by rows	# of rows and columns
mfcol	Fill by columns	# of rows and columns

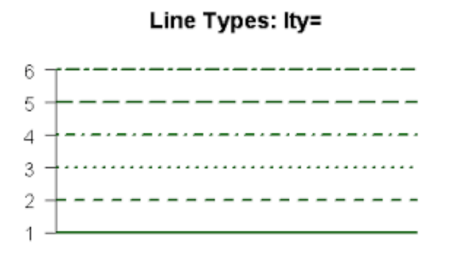
Let us combine 4 plots in 2 rows and 2 columns.

```
# store the current parameter settings in init_par
init_par<-par(no.readonly = TRUE)
# specify that 4 plots to be combined and filled by rows
par(mfrow = c(2,2))
#specify the margin
par(mar = rep(2, 4))
# specify the graphs
plot(iris$Sepal.Length)
plot(iris$Sepal.Length, iris$Sepal.Width)
plot(iris$Petal.Length)
plot(iris$Species)
# restore the original settings
par(init_par)
```



Parameters for specifying lines

pch	Symbol to use when plotting points
lty	Line type
lwd	Line width



Why ggplot2?

R Base Graphics	ggplot2
• In R, base graph start out easy • But become tedious and code looks quite complicated when more details are added	• ggplot gives very fine control • ggplot has a "grammar" – Created by Lee Wilkinson – Built layer by layer • Easier once you get started • After some practice, you can build publication level charts

Why ggplot2?

- It gives professional quality graphs
- It is pretty
- It's easy to manipulate
- It has knowledge transfers to other packages/Languages
- It has steep learning curve
- Also, it can be slow

Components of ggplot

- The data to be plotted/graphed – ggplot works only with data frames
- The geom objects to represent the data (shapes) – points, lines, text, segments, bars
- The Aesthetics are the visual properties of the geoms – x,y, line-color,fill, size, point-shape, line-type
- The mapping from Data to Aesthetics

– Statistical transfromations: Scales, size, groups

Optional: The stat tells ggplot how to transform the data geom's and stat's go together
Stat_bin, stat_smooth etc from Data to Aesthetics
Extra: Themes, legends, title, axes, facets

Lets Do a scatter plot of weight versus MPG.

```
plot(mtcars$wt, mtcars$mpg)
```

Now the qplot version.
Load library(ggplot2)

```
qplot(mtcars$wt, mtcars$mpg)
```

Alternate syntax declares the data.frame and just uses variable names from it.

```
qplot(wt, mpg, data = mtcars)
```

This plot is created in ggplot syntax with

```
ggplot(data = mtcars, aes(x = wt, y = mpg)) + geom_point()
```


Creating a line graph

```
str(pressure)
```

'data.frame': 19 obs. of 2 variables: ## $ temperature: num 0 20 40 60 80 100 120 140 160 180 ... ## $ pressure : num 0.0002 0.0012 0.006 0.03 0.09 0.27 0.75 1.85 4.2 8.8 ...

```
plot(pressure$temperature, pressure$pressure, type = "l") # Add the points
points(pressure$temperature, pressure$pressure) # Add similar plots for pressure/2 colored in red
lines(pressure$temperature, pressure$pressure/2, type = "l", col = "red")
points(pressure$temperature, pressure$pressure/2, col = "red")
```

With qplot we can plot a line fit to the points by specifing the *geom*, short for geometry of what is plotted.

```
qplot(temperature, pressure, data = pressure, geom = "line")
```

```
# Equivalently: ggplot(data = pressure, aes(x = temperature, y = pressure)) + geom_line()
```

```
# Add points
qplot(temperature, pressure, data = pressure, geom = c("line", "point"))

ggplot(data = pressure, aes(x = temperature, y = pressure)) + geom_line() + geom_point()
```

Creating a histogram
A histogram is not a direct plot of data, but a summary of values in specified bins.

```
hist(mtcars$mpg)
```

This grouped data in ranges of 5 and plotted the number of samples in that range. We can specify a range; i.e., bin width.

```
hist(mtcars$mpg, breaks = 10)
```

Here's how ggplot does this:

```
ggplot(data = mtcars, aes(x = mpg)) + geom_histogram(binwidth = 4)
```

Here, the *y* variable was calculated. In ggplot this is known as a *statistic* associated with the plot.

Boxplots
This is a plot a continuous variable versus a discrete one.

```
str(ToothGrowth)

summary(ToothGrowth$dose)

table(ToothGrowth$dose)

# Here, x is a factor, declaring it to be discrete
plot(ToothGrowth$supp, ToothGrowth$len)

# In formula syntax this is
 boxplot(len ~ supp, data = ToothGrowth)
```

We can also include the interactions of variables in the formula syntax.

```
boxplot(len ~ supp + dose, data = ToothGrowth)
```

The latter is done in ggplot with

```
ggplot(data = ToothGrowth, aes(x = interaction(supp, dose), y = len)) + geom_boxplot()
```

The structure of ggplot
ggplot was based on the Grammar of Graphics developed by Wilkinson, 2005. It puts an organized framework around the various parts of a graph. The components of this grammar that we'll use in plotting are as follows.
Mandatory features
data is simply what it says. It's the source of what you'll plot. In ggplot2 this must be a data.frame
aesthetic mapping Here, an aesthetic is a visual feature that cna be used to represent properties of the variables, like x, y, color, shape, size, linetype, pointtype, etc. Variables in the data are mapped to aesthetics for plotting purposes.
geom = geometric object. This is what you actually see, be it line, points, polygons, bars, etc.
Optional or implicit features
stat is a statistical transformation of the data producing some quantity that is then mapped to an aesthetic.
scales map values in the data space to values in the aesthetic space (color, size, shape, ...). Scales are reported on the plot in a legend or axis labels.
coord is a coordinate system. This describes how x and y are layed out on the plane. Normally, cartesian coordinates are used.
facet A faceting specification describes how to break up the data into subsets and display those in subplots.

Layers and persistance

A very handy feature of ggplot is that it returns a ggplot object that can later to reused or added to.

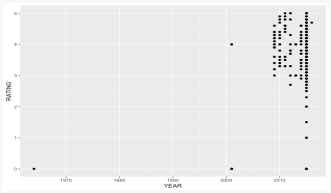
```
tooth_box_plot <- ggplot(data = ToothGrowth, aes(x = interaction(supp, dose), y = len)) + geom_boxplot()
tooth_box_plot

class(tooth_box_plot)

names(tooth_box_plot)
```

First principles of creating a plot
1.A dataframe containing what you are plotting should be specified.
2.An aesthetic must be given declaring x, y or both as some variable in the data.frame. For some plots (histograms, density, e.g.), one of these will be calculated as a stat from the data.
3.A geom must be specified, stating how the aesthetics will appear as geomtrical objects.

Building Plots Iteratively…

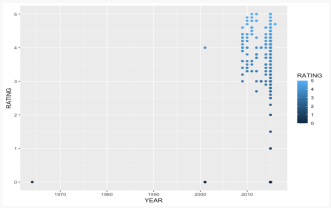
Movies data set Question: How have the ratings changed over the years?

```
library(ggplot2)
library(ggplot2movies)
p <- ggplot(movies , aes(x=year, y=rating)) +geom_point()
print(p)
```



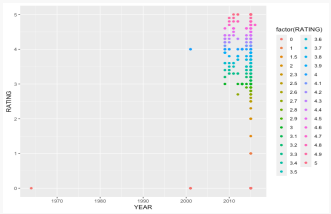
Maybe we need different colors

```
library(ggplot2)
p <- ggplot(movies , aes(x=year, y=rating, color = rating))
+geom_point()
 print(p)
```



Maybe we need different colors

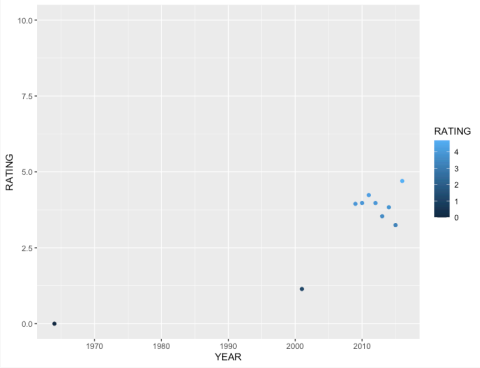
```
library(ggplot2)
p <- ggplot(movies , aes(x=year, y=rating, color = factor(rating))) +geom_point()
print(p)
```



Looks very colorful,
but this is not what we want!

Building Plots Iteratively…

- We need only one point for each year
- We need some way to summarize the ratings into one value
- Not clear what new column we need to add in the "movies" dataset
- Let's create a new data frame for our needs. It will have only two columns! Years and Rating




```
aggregated_ratings_df <- aggregate(RATING ~ YEAR,data=movies,FUN=mean)
str(aggregated_ratings_df)
p2 <- ggplot(aggregated_ratings_df,aes(x=YEAR, y=RATING, color =RATING)) +geom_point()
p3 <-p2+scale_y_continuous(limits = c(0,10))
plot(p3)
```

Geom's control the plot types

- The geom layer defines the overall look of the plot. geom tell us the chart type

geom_point()	geom_line()
geom_bar()	geom_histogram()
geom_segment()	geom_text()
geom_polygon()	geom_map()

Read more at http://docs.ggplot2.org/current/
Next we use **aesthetics** to control the shape, size, color and position of the **geoms**

Understanding ggplot aesthetics

- aes(x=df_variable_x, y = df_variable_y)

When mapping you can take any df.column and 'map' it to an aesthetic

aes(fill = df_variable_name)
aes(color = df_variable_name)
aes(shape = df_variable_name)

VERY IMPORTANT: variables go inside the aes() where fixed aesthetics go outside

```
ggplot(movies, aes(x=year, y=rating, color = mpaa)) + geom_point()
# instance of " mapping" (inside aes)

ggplot(movies, aes(x=year, y=rating), color = "blue") +geom_point()
 # an instance of setting color outside aes()
```

Improving the Look and Feel
Titles, Legends, colors

Plot and Legend

```
p <- ggplot(data=PlantGrowth, aes(x=group, y=weight, fill=group)) +
geom_boxplot()
p

# Remove legend for a particular aesthetic (fill)
p + guides(fill=FALSE)

# It can also be done when specifying the scale
p + scale_fill_discrete(guide=FALSE)

# This removes all legends
p + theme(legend.position="none")

#This changes the order of items to trt1, ctrl, trt2:
p + scale_fill_discrete(breaks=c("trt1","ctrl","trt2"))
```

Plot and Legend

Reversing the order of items in the legend

```
# These two methods are equivalent:
p + guides(fill = guide_legend(reverse=TRUE))

p + scale_fill_discrete(guide = guide_legend(reverse=TRUE))

# You can also modify the scale directly:
p + scale_fill_discrete(breaks = rev(levels(PlantGrowth$group)))

# Remove title for fill legend
p + guides(fill=guide_legend(title=NULL))
# Remove title for all legends
p + theme(legend.title=element_blank())
```


Using scales

The legend can be a guide for fill, colour, linetype, shape, or other aesthetics.

With fill and color
Because group, the variable in the legend, is mapped to the color fill, it is necessary to use scale_fill_xxx, where xxx is a method of mapping each factor level of group to different colors.
The default is to use a different hue on the color wheel for each factor level,
but it is also possible to manually specify the colors for each level.

```
p + scale_fill_discrete(name="Experimental\nCondition")

p + scale_fill_discrete(name="Experimental\nCondition",
      breaks=c("ctrl", "trt1", "trt2"),
      labels=c("Control", "Treatment 1", "Treatment 2"))

# Using a manual scale instead of hue
 p + scale_fill_manual(values=c("#999999", "#E69F00", "#56B4E9"),
      name="Experimental\nCondition",
      breaks=c("ctrl", "trt1", "trt2"),
      labels=c("Control", "Treatment 1", "Treatment 2"))
```

Note that this didn't change the x axis labels.

Controlling Colors in ggplot

- color = “red”
- aes(color = df.var)
- By default, ggplot chooses its own color set
- ggplot chooses colors for “ categorical variables”
- ggplot decides a color scale for continuous variables
- However, everything can be changed
- We do that through the scale_fill_xxx or the scales_color_xxx functions

library(ggplot2movies)

- p<-ggplot(movies, aes(x=year, y = rating, color = rating)) + geom_point()
- p <-p+scale_color_continuous(low = “red”, high = “green”)

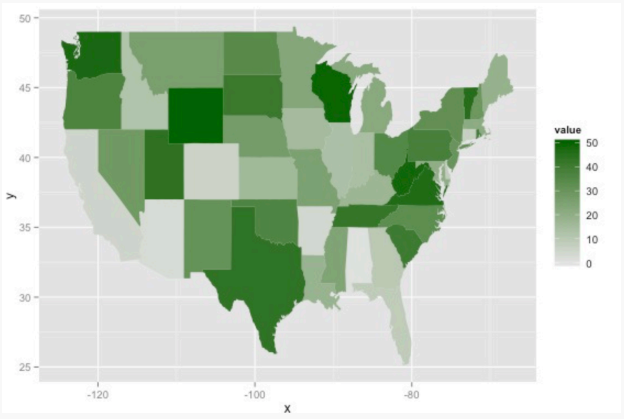
Geo-spatial plotting

Plotting on Maps (package ggmap)

- Get a base map ready
- Get the data frame ready
- Add data to Map

Choropleth plotting

• A thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map, such as population density or per capita income



Choropleth plotting

1. First, create the maps data frame!

```
states_map <- map_data("state") # Create a data frame of map data!

#2. Create a data frame with the values for color intensity!
region <- tolower(state.name)
 # state.name is a built-in dataset!
count.vector <- 1:50
# just a value!
df<- data.frame(region, count.vector)
 #creating a 50x2 data frame here!
names(df) <- c("state", "value")

#Plot the choropleth!

mp <- ggplot(df, aes(map_id=state)) + geom_map(aes(fill=value), map=states_map)
mp <- mp + expand_limits(x = states_map$long, y = states_map$lat)
mp <- mp + scale_fill_gradient(low='grey90', high='darkgreen', limits=c(0,50))
mp
```

Saving Graphs

Once created the graph the graphs can be saved in different formats. You must use the dev.off() command to tell R that you are finished plotting, otherwise your graph will not show up.

Type	Function
pdf	pdf("fn.pdf")
png	png("fn.png")
jpeg	jpeg("fn.jpeg")
bmp	bmp("fn.bmp")
postscript	postscript("fn.ps")

Pie Charts

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)

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.

A very simple pie-chart is created using just the input vector and labels. The below script will create and save the pie chart in the current R working directory.

```
# Create data for the graph.

x <- c(39, 3, 8, 4.3)
labels <- c(”California", ”Nevada", ”Washington", ”Oregon")
# Give the chart file a name.
png(file = ”Statepop.png")
# Plot the chart.
pie(x,labels)
# Save the file.
dev.off()
```


Pie Chart Title and Colors

We can expand the features of the chart by adding more parameters to the function. We will use parameter **main** to add a title to the chart and another parameter is **col** which will make use of rainbow color pallet while drawing the chart. The length of the pallet should be same as the number of values we have for the chart. Hence we use length(x).

```
# Create data for the graph.

x <- c(39, 3, 8, 4.3)
labels <- c(”California", ”Nevada", ”Washington", ”Oregon")
# Give the chart file a name.
png(file = ”State_title_colors.png")
# Plot the chart.
# Plot the chart with title and rainbow color pallet.
pie(x, labels, main = ”State pie chart", col = rainbow(length(x)))
# 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(39, 3, 8, 4.3)
labels <- c("California", "Nevada", "Washington", "Oregon")

piepercent<- round(100*x/sum(x), 1)

# Give the chart file a name.

png(file = "State_percentage_legends.jpg")

# Plot the chart.

pie(x, labels = piepercent, main = "City pie chart",col = rainbow(length(x)))
legend("topright", c(”California", ”Nevada", ”Washington", ”Oregon") , cex = 0.8, fill = rainbow(length(x)))
# Save the file.
dev.off()
```


3D Pie Chart
A pie chart with 3 dimensions can be drawn using additional packages. The package **plotrix** has a function called **pie3D()** that is used for this.

```
# Get the library.
library(plotrix)
# Create data for the graph.

x <- c(39, 3, 8, 4.3)
labels <- c("California", "Nevada", "Washington", "Oregon")

# 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 States ")
# Save the file. dev.off()
```

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.

Following is the description of the parameters :
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.

Example
A simple bar chart is created using just the input vector and the name of each bar.

```
# Create the data for the chart
H <- c(8,13,30,5,43)
# Give the chart file a name
png(file = "barchart.png")
# Plot the bar chart
barplot(H)
# Save the file
dev.off()
```

Bar Chart Labels, Title and Colors

The features of the bar chart can be expanded by adding more parameters. The **main** parameter is used to add **title**. The **col** parameter is used to add colors to the bars.

The **args.name** is a vector having same number of values as the input vector to describe the meaning of each bar.

```
# Create the data for the chart
H <- c(8,13,30,5,43)
M <- c("Apr","May","Jun","Jul", "Aug")
# Give the chart file a name
png(file = "barchart_months_revenue.png")
# Plot the bar chart
barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="blue",main="Revenue chart",border="red")
# Save the file
dev.off()
```


Group Bar Chart and Stacked Bar Chart

We can create bar chart with groups of bars and stacks in each bar by using a matrix as input values.

More than two variables are represented as a matrix which is used to create the 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()
```

Boxplots are a measure of how well distributed is the data in a data set. It divides the data set into three quartiles. This graph represents the minimum, maximum, median, first quartile and third quartile in the data set. It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them.
Boxplots are created in R by using the **boxplot()** function.
Syntax
The basic syntax to create a boxplot in R is −
boxplot(x, data, notch, varwidth, names, main)

Following is the description of the parameters :−
x is a vector or a formula.
data is the data frame.
notch is a logical value. Set as TRUE to draw a notch.
varwidth is a logical value. Set as true to draw width of the box proportionate to the sample size.
names are the group labels which will be printed under each boxplot.
main is used to give a title to the graph.
Example
We use the data set "mtcars" available in the R environment to create a basic boxplot. Let's look at the columns "mpg" and "cyl" in mtcars.


```
input <- mtcars[,c('mpg','cyl')]
print(head(input))

# Give the chart file a name.
png(file = "boxplot.png")
# Plot the chart.
boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders", ylab = "Miles Per Gallon", main = "Mileage Data")
# Save the file.
 dev.off()
```

Boxplot with Notch
We can draw boxplot with notch to find out how the medians of different data groups match with each other.
The below script will create a boxplot graph with notch for each of the data group.

```
# Give the chart file a name.
png(file = "boxplot_with_notch.png")
# Plot the chart.
 boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders", ylab = "Miles Per Gallon", main = "Mileage Data", notch = TRUE, varwidth = TRUE, col = c("green","yellow","purple"), names = c("High","Medium","Low") )
# Save the file.
dev.off()
```


A histogram represents the frequencies of values of a variable bucketed into ranges. Histogram is similar to bar chat but the difference is it groups the values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range.
R creates histogram using **hist()** function. This function takes a vector as an input and uses some more parameters to plot histograms.
Syntax
The basic syntax for creating a histogram using R is −
hist(v,main,xlab,xlim,ylim,breaks,col,border)

Following is the description of the parameters:
v is a vector containing numeric values used in histogram.
main indicates title of the chart.
col is used to set color of the bars.
border is used to set border color of each bar.
xlab is used to give description of x-axis.
xlim is used to specify the range of values on the x-axis.
ylim is used to specify the range of values on the y-axis.
breaks is used to mention the width of each bar.

```
# Create data for the graph.
v <- c(9,13,21,8,36,22,12,41,31,33,19)
# Give the chart file a name.
png(file = "histogram.png")
 # Create the histogram.
hist(v,xlab = "Weight",col = "yellow",border = "blue")
# Save the file.
dev.off()
```


Range of X and Y values

To specify the range of values allowed in X axis and Y axis, we can use the xlim and ylim parameters.

The width of each of the bar can be decided by using breaks.

```
# Create data for the graph.
v <- c(9,13,21,8,36,22,12,41,31,33,19)
# Give the chart file a name.
png(file = "histogram_lim_breaks.png")
# Create the histogram.
hist(v,xlab = "Weight",col = "green",border = "red", xlim = c(0,40), ylim = c(0,5), breaks = 5)
# 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 :
•**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".

```
# Create the data for the chart.
v <- c(7,12,28,3,41)
# Give the chart file a name.
png(file = "line_chart.jpg")
# Plot the bar chart.
plot(v,type = "o")
 # Save the file.
dev.off()
```

Line Chart Title, Color and Labels
The features of the line chart can be expanded by using additional parameters. We add color to the points and lines, give a title to the chart and add labels to the axes.

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


Multiple Lines in a Line Chart
More than one line can be drawn on the same chart by using the **lines()**function.
After the first line is plotted, the lines() function can use an additional vector as input to draw the second line in the chart

```
# 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()
```

Scatterplots show many points plotted in the Cartesian plane. Each point represents the values of two variables. One variable is chosen in the horizontal axis and another in the vertical axis.
The simple scatterplot is created using the **plot()** function.
Syntax
The basic syntax for creating scatterplot in R is −
plot(x, y, main, xlab, ylab, xlim, ylim, axes)
Following is the description of the parameters :−
•**x** is the data set whose values are the horizontal coordinates.
•**y** is the data set whose values are the vertical coordinates.
•**main** is the tile of the graph.
•**xlab** is the label in the horizontal axis.
•**ylab** is the label in the vertical axis.
•**xlim** is the limits of the values of x used for plotting.
•**ylim** is the limits of the values of y used for plotting.
•**axes** indicates whether both axes should be drawn on the plot.

Example
We use the data set **"mtcars"** available in the R environment to create a basic scatterplot. Let's use the columns "wt" and "mpg" in mtcars.

```
input <- mtcars[,c('wt','mpg')]
 print(head(input))
```

Live Demo

```
# Get the input values.
input <- mtcars[,c('wt','mpg')]
 # Give the chart file a name.
png(file = "scatterplot.png")
# Plot the chart for cars with weight between 2.5 to 5 and mileage between 15 and 30.
plot(x = input$wt,y = input$mpg, xlab = "Weight", ylab = "Milage", xlim = c(2.5,5), ylim = c(15,30),
main = "Weight vs Milage" )
# Save the file.
dev.off()
```

Scatterplot Matrices

When we have more than two variables and we want to find the correlation between one variable versus the remaining ones we use scatterplot matrix. We use **pairs()** function to create matrices of scatterplots.

Syntax

The basic syntax for creating scatterplot matrices in R is −

pairs(formula, data)

Following is the description of the parameters :

•**formula** represents the series of variables used in pairs.

•**data** represents the data set from which the variables will be taken.

Example

Each variable is paired up with each of the remaining variable. A scatterplot is plotted for each pair.


```
# Give the chart file a name.
png(file = "scatterplot_matrices.png")
# Plot the matrices between 4 variables giving 12 plots.
# One variable with 3 others and total 4 variables.
pairs(~wt+mpg+disp+cyl,data = mtcars, main = "Scatterplot Matrix")
# Save the file.
dev.off()
```

Time Series Charts using ggplot

```
library(ggplot2)
library(dplyr)

# Dummy data
data <- data.frame(
  day = as.Date("2017-06-14") - 0:364,
  value = runif(365) + seq(-140, 224)^2 / 10000
)
#The runif() function generates random deviates of the uniform distribution
# Most basic bubble plot
p <- ggplot(data, aes(x=day, y=value)) +
  geom_line() +
  xlab("")
p

p+scale_x_date(date_labels = "%b")
p+scale_x_date(date_labels = "%Y %b %d")
p+scale_x_date(date_labels = "%W")
p+scale_x_date(date_labels = "%m-%Y")
```

```
seq(...)
```

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Press F1 for additional help

Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a %A	abbreviated weekday unabbreviated weekday	Mon Monday
%m	month (00-12)	00-12
%b %B	abbreviated month unabbreviated month	Jan January
%y %Y	2-digit year 4-digit year	07 2007

It also possible to control the amount of break and minor breaks to display with date_breaks and date_minor_breaks.

```
p + scale_x_date(date_breaks = "1 week", date_labels = "%W")
p + scale_x_date(date_minor_breaks = "2 day")
```

```
library(ggplot2)
library(dplyr)
library(hrbrthemes)

# Dummy data
data <- data.frame(
  day = as.Date("2017-06-14") - 0:364,
  value = runif(365) - seq(-140, 224)^2 / 10000
)

p <- ggplot(data, aes(x=day, y=value)) +
  geom_line( color="#69b3a2") +
  xlab("") +
  theme_ipsum() +
  theme(axis.text.x=element_text(angle=60, hjust=1))

p
```


Select Time Frame

Use the limit option of the scale_x_date() function to select a time frame in the data:

```
p <- ggplot(data, aes(x=day, y=value)) +
 geom_line( color="steelblue") +
geom_point() +
 xlab("") +
 theme_ipsum() +
 theme(axis.text.x=element_text(angle=60, hjust=1)) +
scale_x_date(limit=c(as.Date("2017-01-01"),as.Date("2017-02-11"))) +
ylim(0,1.5)

p
```

Visualization – Additional Resources

Cookbook for R >> Graphs (website)
http://www.cookbook-r.com/Graphs/
Very good resource and well explained

Intro to R Graphics with ggplot2 – Ista Zahn SlideShare of a good set of slides.
Includes an example to recreate ‘ The Economist” style graphs

Conditional Plots
http://answers.oreilly.com/topic/2428-when-and- how-t-use-conditional-plots-co-plots-for-data- analysis