Data Visualization with R

Base Graphics

Aravind Hebbali

2026-09-26

Contents

Data Visualization with R

Base Graphics

Preface

This book requires R 4.1 or later.

Software information

The R session information when compiling this book is shown below:

```
sessionInfo()
```

```
R version 4.5.2 (2025-10-31 ucrt)
Platform: x86_64-w64-mingw32/x64
Running under: Windows 10 x64 (build 19045)

Matrix products: default
  LAPACK version 3.12.1

locale:
[1] LC_COLLATE=English_India.utf8  LC_CTYPE=English_India.utf8
[3] LC_MONETARY=English_India.utf8 LC_NUMERIC=C
[5] LC_TIME=English_India.utf8

time zone: Asia/Calcutta
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.5.2  fastmap_1.2.0   cli_3.6.5       tools_4.5.2
 [5] htmltools_0.5.9 yaml_2.3.12     rmarkdown_2.30  knitr_1.50
 [9] jsonlite_2.0.0  xfun_0.61       digest_0.6.39   rlang_1.2.0
[13] evaluate_1.0.5
```

We do not add prompts (> and +) to R source code in this book, and we comment out the text output with two hashes ## by default, as you can see from the R session information above. This is for your convenience when you want to copy and run the code (the text output will be ignored since it is commented out). Package names are in bold text (e.g., **rmarkdown**), and function names are followed by parentheses (e.g., `bookdown::render_book()`). The double-colon operator `::` means accessing an object from a package.

Introduction

Introduction

This is the first chapter of **Data Visualization With R**. The objective of the book is to provide a gentle introduction to working with base graphics in R.

- what is data visualization
- why visualize data

- understand R graphics system
  - graphics
  - ggplot2
  - lattice
- build some simple plots

What is data visualization?

In simple words, data visualization is the representation of data in graphical format.

Why visualize data?

- Explore: Visualization helps in exploring and explaining patterns and trends
- Detect: Patterns or anomalies in data can be detected by looking at graphs
- Make sense: Possible to make sense of large amount of data efficiently and in time
- Communicate: Easy to communicate and share the insights from data

R Graphics System

- graphics
- ggplot2
- lattice

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 it.

ggplot2

ggplot2, created by Hadley Wickham, is based on the Grammar of Graphics written by Leland Wilkinson. It has a structured approach to data visualization and builds upon the features available in the Graphics and Lattice packages.

Lattice

The lattice package is inspired by Trellis Graphics and created by Deepayan Sarkar. It is a very powerful data visualization system with an emphasis on multivariate data.

Getting Help

Use the `help()` to learn more about `plot()` function and `mtcars` data set.

```
help(plot)
help(mtcars)
```

mtcars

```
{webr-r}
head(mtcars)
```

Variable Info

```
{webr-r}
str(mtcars)
```

plot()

Now that we have some idea about the data set, let us explore the `plot()` function. We will use the following different data inputs and observe the kind of plots that are generated:

- Case 1: 1 continuous variable
- Case 2: 1 categorical variable
- Case 3: 2 continuous variables
- Case 4: 2 categorical variables
- Case 5: 1 continuous and 1 categorical variable
- Case 6: 1 categorical and 1 continuous variable

One continuous variable

```
{webr-r}
plot(mtcars$mpg)
```

One categorical variable

```
{webr-r}
plot(as.factor(mtcars$cyl))
```

Two continuous variables

```
{webr-r}
plot(mtcars$disp, mtcars$mpg)
```

Two categorical variables

```
{webr-r}
plot(as.factor(mtcars$am), as.factor(mtcars$cyl))
```

Continuous/Categorical variable

```
{webr-r}
plot(mtcars$mpg, as.factor(mtcars$cyl))
```

Categorical/Continuous variable

```
{webr-r}
plot(as.factor(mtcars$cyl), mtcars$mpg)
```

Title & Axis Labels

Introduction

In this chapter, we will learn how to add:

- Title
- Subtitle
- Axis Labels

to a plot and to modify:

- the range of an axis

In the previous chapter, we created plots which did not have any title or labels. Such plots are of no use to any one as they do not indicate what the X and Y axis represent or the primary information being communicated by the plot. The `title` and `labels` play an important part in making the plot holistic. There are two ways to add them to a plot:

- use the relevant arguments within the `plot()` function
- use the `title()` function

We will explore both the above methods one by one, and you can choose the method most convenient to you. Let us begin with the `plot()` function:

Syntax

Feature	Argument	Value	Example
Title	main	string	"Scatter Plot"
Subtitle	sub	string	"Displacement vs Miles Per Gallon"
X Axis Label	xlab	string	"Displacement"
Y Axis Label	ylab	string	"Miles Per Gallon"

Title

You can add a title to the plot using the `main` argument in the `plot()` function. Ensure that the title is enclosed in single/double quotes as it is a string. Let us create a scatter plot of `disp` and `mpg` from `mtcars` data set, and add a title to it.

```
{webr-r}
plot(mtcars$disp, mtcars$mpg,
     main = 'Displacement vs Miles Per Gallon')
```

Subtitle

You can add a subtitle to the plot using the `sub` argument in the `plot()` function. The subtitle will appear below the X axis label. Ensure that the subtitle is enclosed in single/double quotes as it is a string. Let us add a subtitle to the plot we created in the previous example:

```
{webr-r}
plot(mtcars$disp, mtcars$mpg,
     main = 'Displacement vs Miles Per Gallon',
     sub = 'Mileage is inversely related to Displacement')
```

Axis Labels

In the plots created in the previous examples, the axis labels appear as `mtcars$mpg` and `mtcars$disp`. It is not the best way to name the axis and it will make more sense to use names that describe the data. Let us modify the axis labels using the `xlab` and `ylab` arguments in the `plot()` function:

```
{webr-r}
plot(mtcars$disp, mtcars$mpg,
```

```
     main = 'Displacement vs Miles Per Gallon',
     sub = 'Mileage is inversely related to Displacement',
     xlab = 'Displacement', ylab = 'Miles Per Gallon')
```

title()

We can add title, subtitle and axis labels using the `title()` function as well. Let us recreate the plots from the previous examples but this time we will use the `title()` instead of the `plot()` function. We will continue to use the `plot()` function to create the plot.

```
{webr-r}
# create scatter plot
plot(mtcars$disp, mtcars$mpg)

# add title, subtitle and axis labels
title(main = 'Displacement vs Miles Per Gallon',
      sub = 'Mileage is inversely related to Displacement',
      xlab = 'Displacement', ylab = 'Miles Per Gallon')
```

Do you notice that the axis labels are overwritten? This happens because the `plot()` function adds the default labels and we add a new set of labels without modifying the existing ones. The solution is to instruct the `plot()` function not to add any labels to the X and Y axis. This can be achieved using the `ann` (annotate) argument in the `plot()` function and set it to `FALSE`. Let us try it:

```
{webr-r}
# create scatter plot
plot(mtcars$disp, mtcars$mpg, ann = FALSE)

# add title, subtitle and axis labels
title(main = 'Displacement vs Miles Per Gallon',
      sub = 'Mileage is inversely related to Displacement',
      xlab = 'Displacement', ylab = 'Miles Per Gallon')
```

The axis labels are legible and not overwritten. You can use either the `plot()` function or the `title()` function to add title, subtitle and axis labels but ensure that in case you use the `title()` function, set `ann` argument to `FALSE` in the `plot()` function.

Axis Range

In certain cases, you would want to modify the range of the axis of the plots. By default, the `plot()` function will take into account the `min` and `max` values of the variable(s) and set the range for the axis. We can modify the range by using the `xlim` and `ylim` arguments in the `plot()` function. Both the `xlim` and `ylim` arguments take 2 values as inputes. The first value is the minimum value for the axis and the second value is the maximum value for the axis. The `plot()` function will return an error if we do not specify two values for both `xlim` and `ylim` arguments. Let us recreate the plot from the previous examples but change the range of both the X and Y axis:

```
{webr-r}
plot(mtcars$disp, mtcars$mpg,
     xlim = c(0, 600), ylim = c(0, 50))
```

Keep in mind that the axis ranges cannot be modified using the `title()` function.

Scatter Plots

Introduction

In this chapter, we will learn how to create scatter plots.

- adding color to the points
- modify shape of the points
- modify size of the points

Basic Plot

Let us recreate the plot that we had created in Chapter 1 by using the `mtcars` data set. We will use the `disp` (displacement) and `mpg` (miles per gallon) variables. `disp` will be on the X axis and `mpg` will be on the Y axis.

```
{webr-r}
plot(mtcars$disp, mtcars$mpg)
```

We have created a very basic plot and any one looking at it for the first time will get confused with the axis labels `mtcars$disp` and `mtcars$mpg`. Let us put into practice what we learnt in Chapter 2, and add a title to the plot, and make the axis labels more meaningful.

```
{webr-r}
plot(mtcars$disp, mtcars$mpg,
     main = 'Displacement vs Miles Per Gallon',
     xlab = 'Displacement', ylab = 'Miles Per Gallon')
```

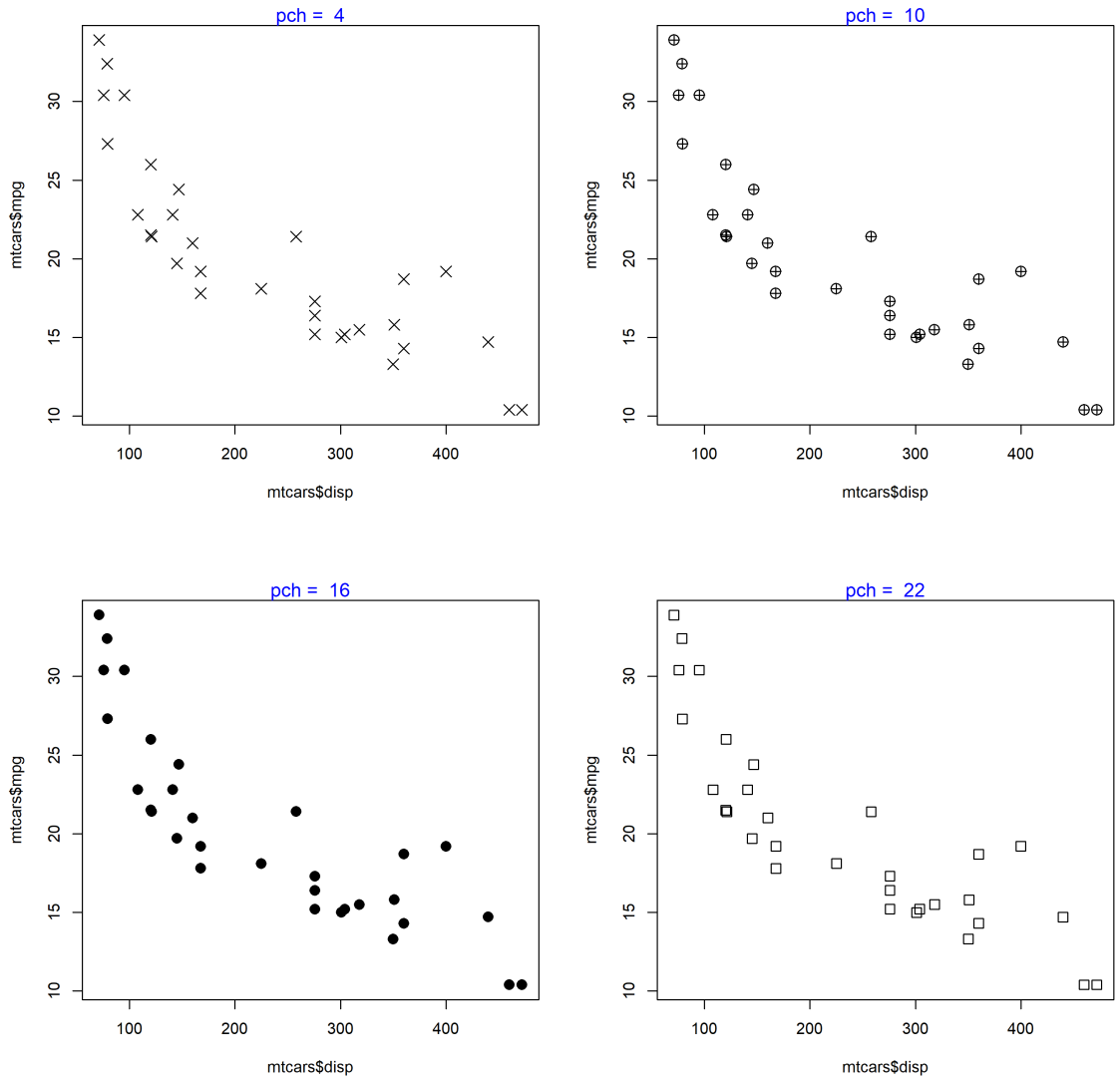
Now the plot clearly communicates that it represents the relationship between the displacement and mileage of cars. Now the color of the points in the plot is black by default. Some of us may agree that black is beautiful but not all of us will like it. As a first step in enhancing the way our plot looks, let us change the shape of the points.

Shape

The shape of the point can be specified using the `pch` argument. It will take values between **0** and **25**. Below is an example:

```
{webr-r}
# point shape
plot(mtcars$disp, mtcars$mpg, pch = 6)
```

Let us check out a few of the other shapes:



We can specify the shape based on a third (categorical variable as well). In the below plot, the shape is based on the levels of the categorical variable `cyl` (number of cylinders) from the `mtcars` data set. Note that `nlevels(factor(mtcars$cyl))` returns a single value (3), so it would give every point the same shape — instead, map each observation to its own level with `as.numeric(factor(...))`:

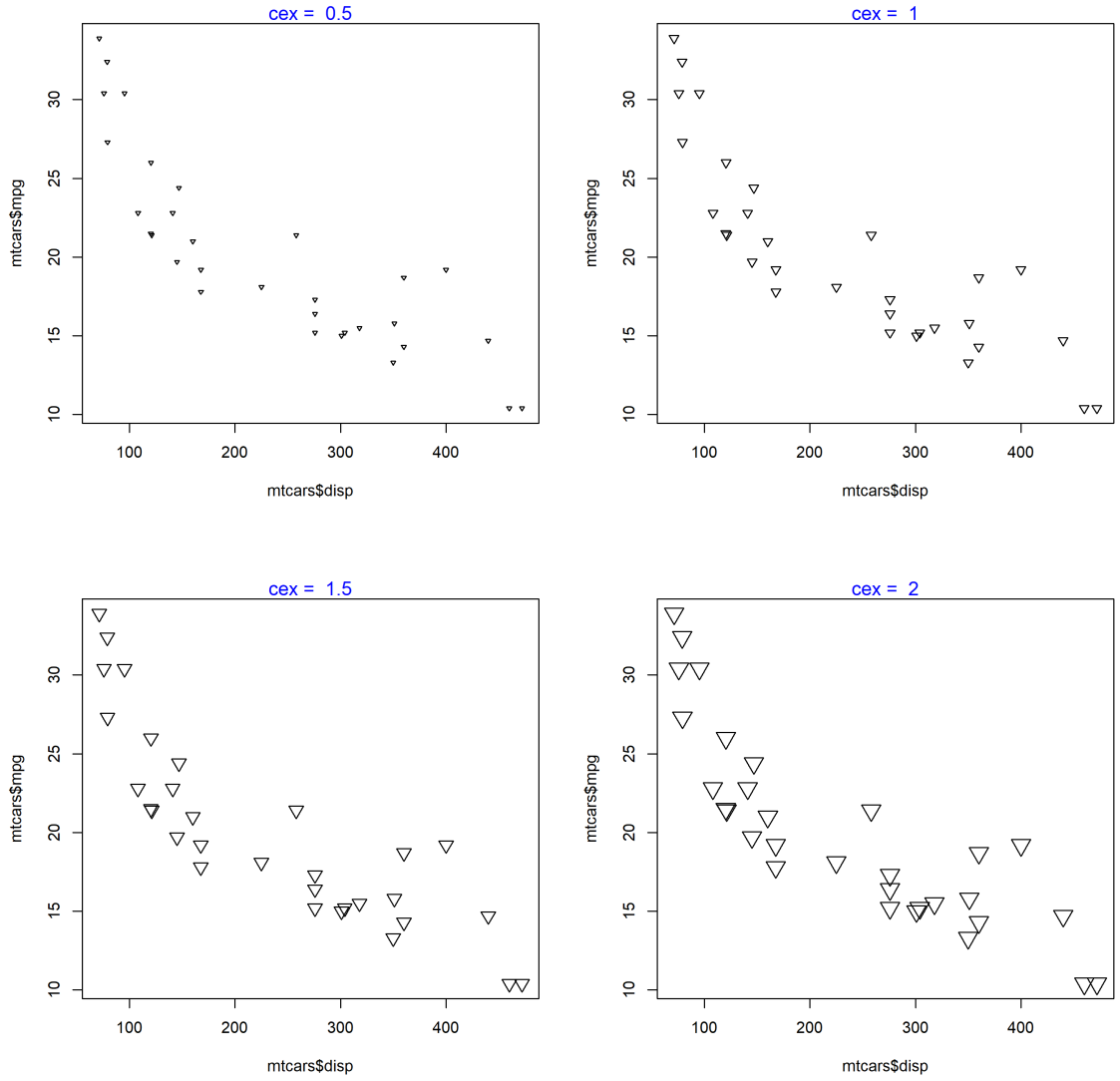
```
{webr-r}
# shape based on a third categorical variable
plot(mtcars$disp, mtcars$mpg, pch = as.numeric(factor(mtcars$cyl)))
legend('topright', legend = c('4 cyl', '6 cyl', '8 cyl'), pch = 1:3)
```

Size

The size of the points in the scatter plot can be specified using the `cex` argument in the `plot()` function. The default value for `cex` is
1.

```
{webr-r}
# point size
plot(mtcars$disp, mtcars$mpg, cex = 1.5)
```

The below plots show the size of the points for values relative to 1.



Color

We can specify a border color for the points using the `col` argument and a background color using the `bg` argument. The background color can be specified only for points whose `pch` argument takes values between **21** and **25**. Let us look at some examples to understand this distinction between border and background color.

```
{webr-r}
# shape between 0 and 21
plot(mtcars$disp, mtcars$mpg, pch = 5, col = 'blue', bg = 'red')
```

You can observe that although we have specified a background color using the `bg` argument, we do not see the red background color as the value specified for the `pch` (shape) argument is not between 21 and 25. In the next example, we will use a value between 21 and 25 so that the `pch` argument is effective.

```
{webr-r}
# shape between 22 and 25
plot(mtcars$disp, mtcars$mpg, pch = 24, col = 'red', bg = 'blue')
```

The color of the points can be specified using (levels) of a categorical variable as well. In the next example, we will use the `cyl` variable to specify the color of the points.

```
{webr-r}
# color based on a third variable
plot(mtcars$disp, mtcars$mpg, pch = 5, col = factor(mtcars$cyl))
```

Since `cyl` is a categorical variable with 3 levels, we can see that the points now have 3 different colors. The above method is useful when you want to segregate the points in a scatter plot based on a third variable.

Line Graphs

Introduction

In this chapter, we will build line graphs. To be more specific we will learn to

- create line plots
- add color to lines
- modify line type/style
- modify line width
- add points to the lines
- modify axis range
- add additional lines to the plot

Basic Plot

To build a line graph, we will learn a new argument in the `plot()` function called `type`. It allows us to specify the symbol that must be used to represent the data. Let us begin by building a simple line graph. We will use the `AirPassengers` data set in this chapter. Before we begin to build the plot, let us take a quick look at the data in order to understand what we are plotting.

```
head(AirPassengers)
```

```
     Jan Feb Mar Apr May Jun
1949 112 118 132 129 121 135
```

In order to build a line plot, we will set the `type` argument in the `plot()` function to **l** (line). There are other values which `type` takes but we will explore them later.

```
data <- head(AirPassengers)
plot(data, type = 'l')
```

If you do not like plain lines, you can represent the data using lines interspersed with points by setting the `type` argument to **b** (both lines and points).

```
plot(data, type = 'b')
```

Another option is to have the points and lines overplotted. It can be achieved by setting the `type` argument to **o** (overplotted).

```
plot(data, type = 'o')
```

You can also create lines without points but with breaks instead by setting the `type` argument to **c**.

```
plot(data, type = 'c')
```

Color

So now we know how to build a simple line graph. Let us now make this plot more elegant by modifying its appearance. Let us begin by adding some color to the line using the `col` argument in the `plot()` function.

```
plot(data, type = 'l', col = 'blue')
```

If you have points along with the line, they will have the same color as well.

```
plot(data, type = 'b', col = 'blue')
```

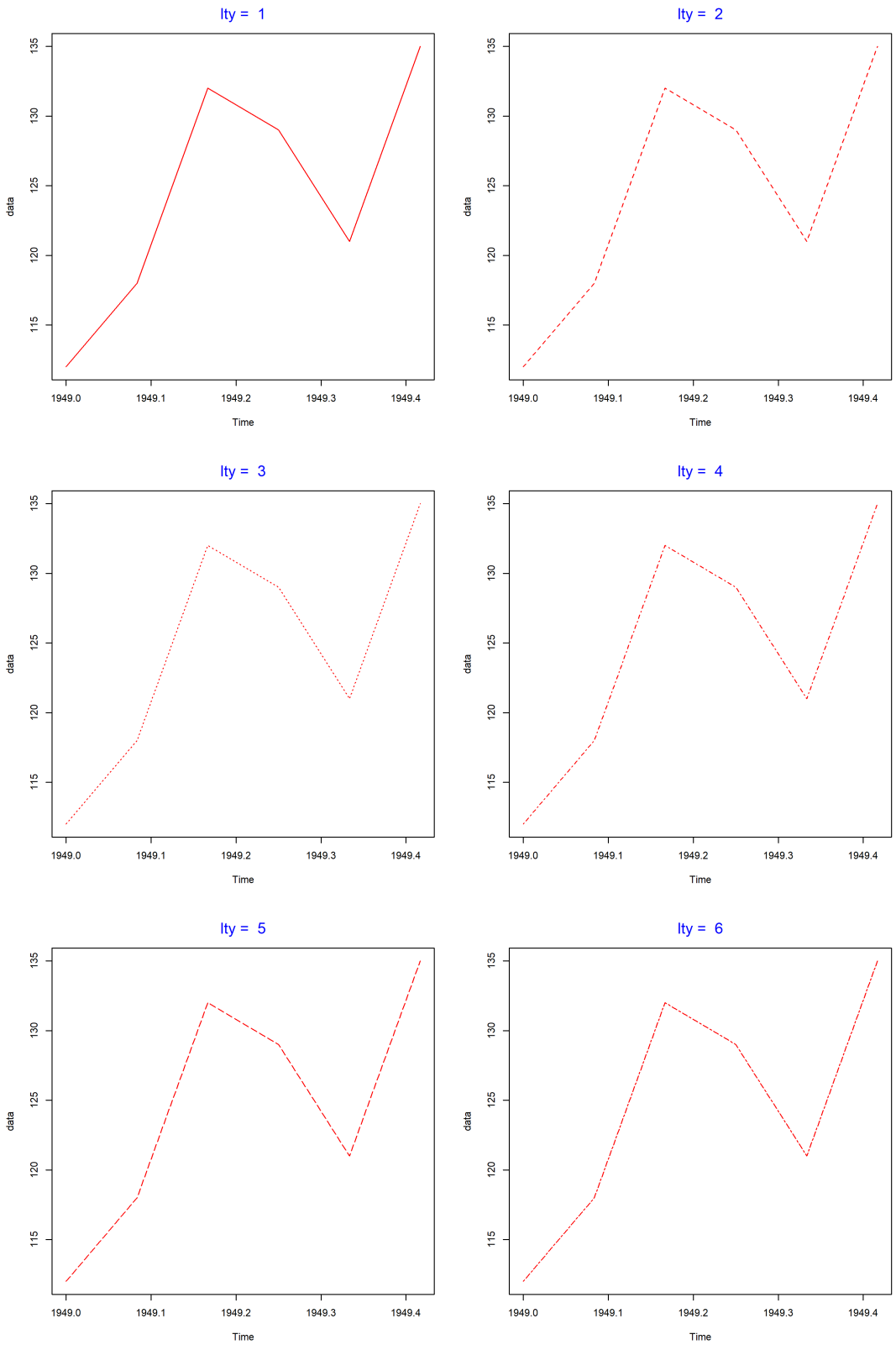
Line Type

The line type can be modified using the `lty` argument. It takes values from 1 to 6 and the default value is 1. Below is an example:

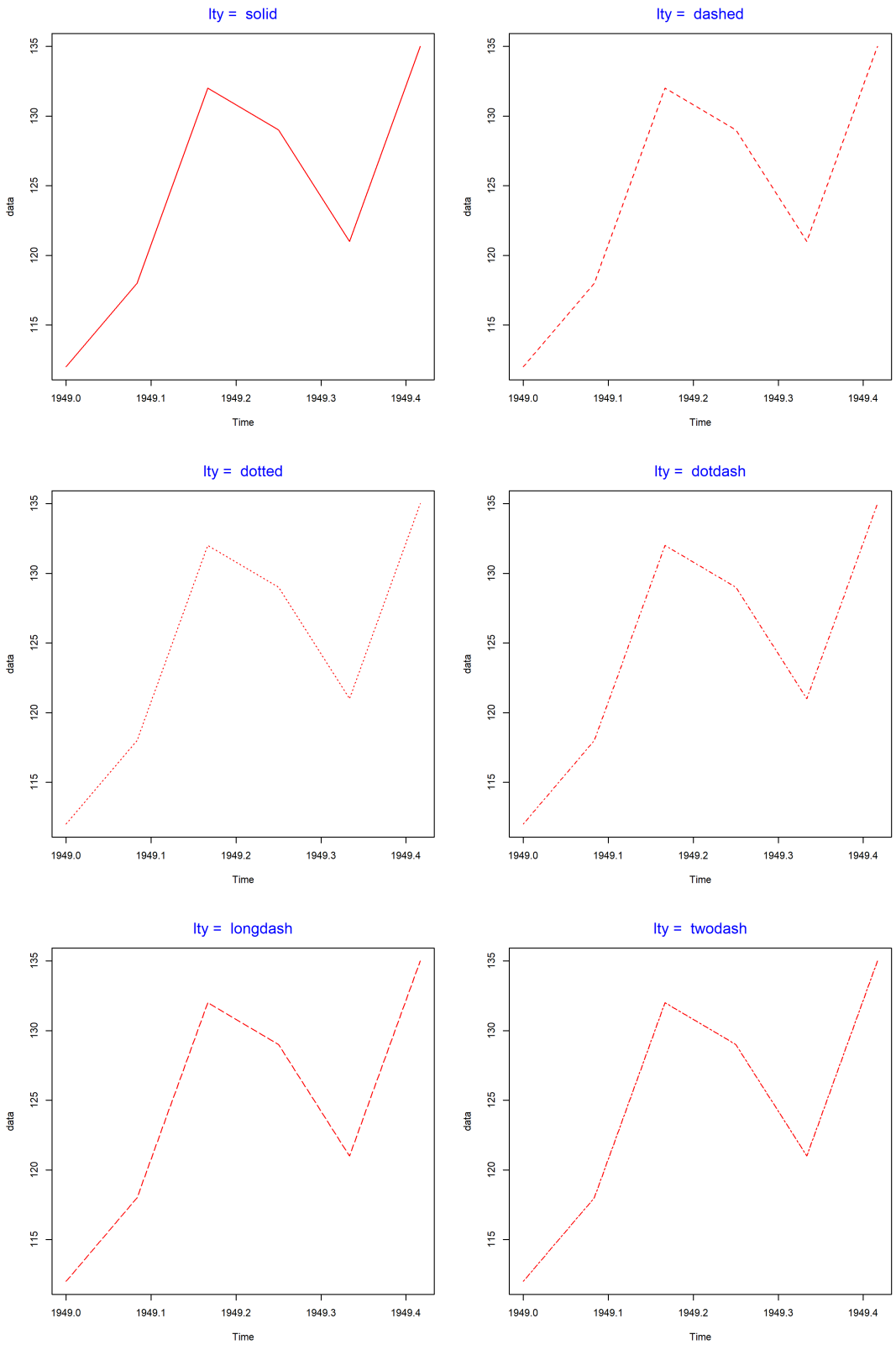
- 1:solid
- 2:dashed
- 3:dotted
- 4:dotdash
- 5:longdash
- 6:twodash

```
plot(data, type = 'l', lty = 3)
```

Let us look at all the line types in the below example:


lty = 1
lty = 2
lty = 3
lty = 4
lty = 5
lty = 6
data
Time
115
120
125
130
135
1949.0
1949.1
1949.2
1949.3
1949.4

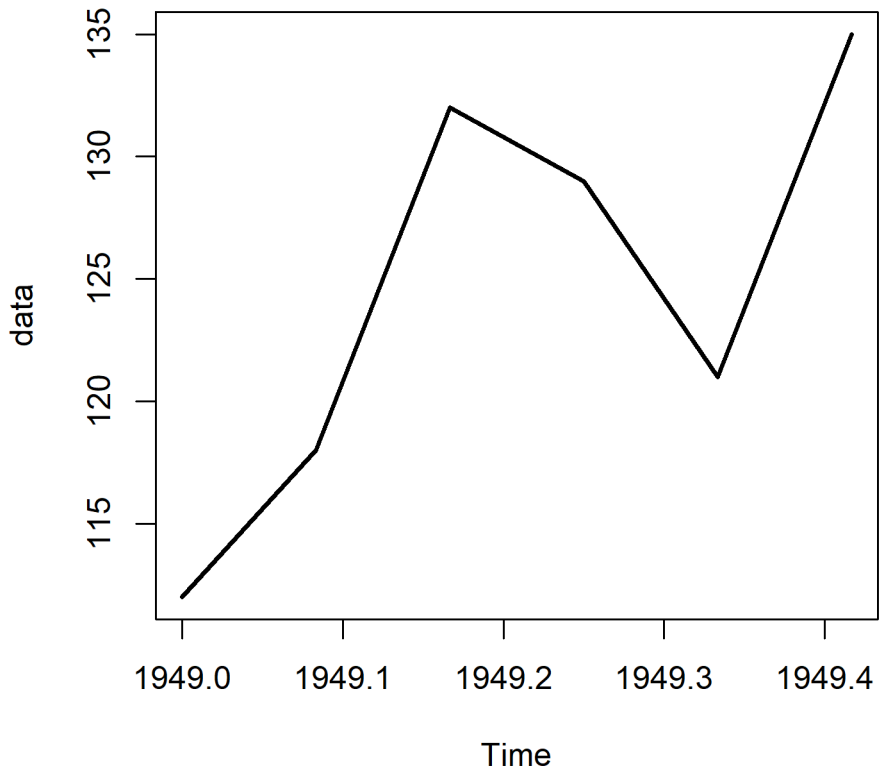
Instead of specifying the numbers 1:6, you can use their description as well.


lty = solid
lty = dashed
lty = dotted
lty = dotdash
lty = longdash
lty = twodash
data
Time
115
120
125
130
135
1949.0
1949.1
1949.2
1949.3
1949.4

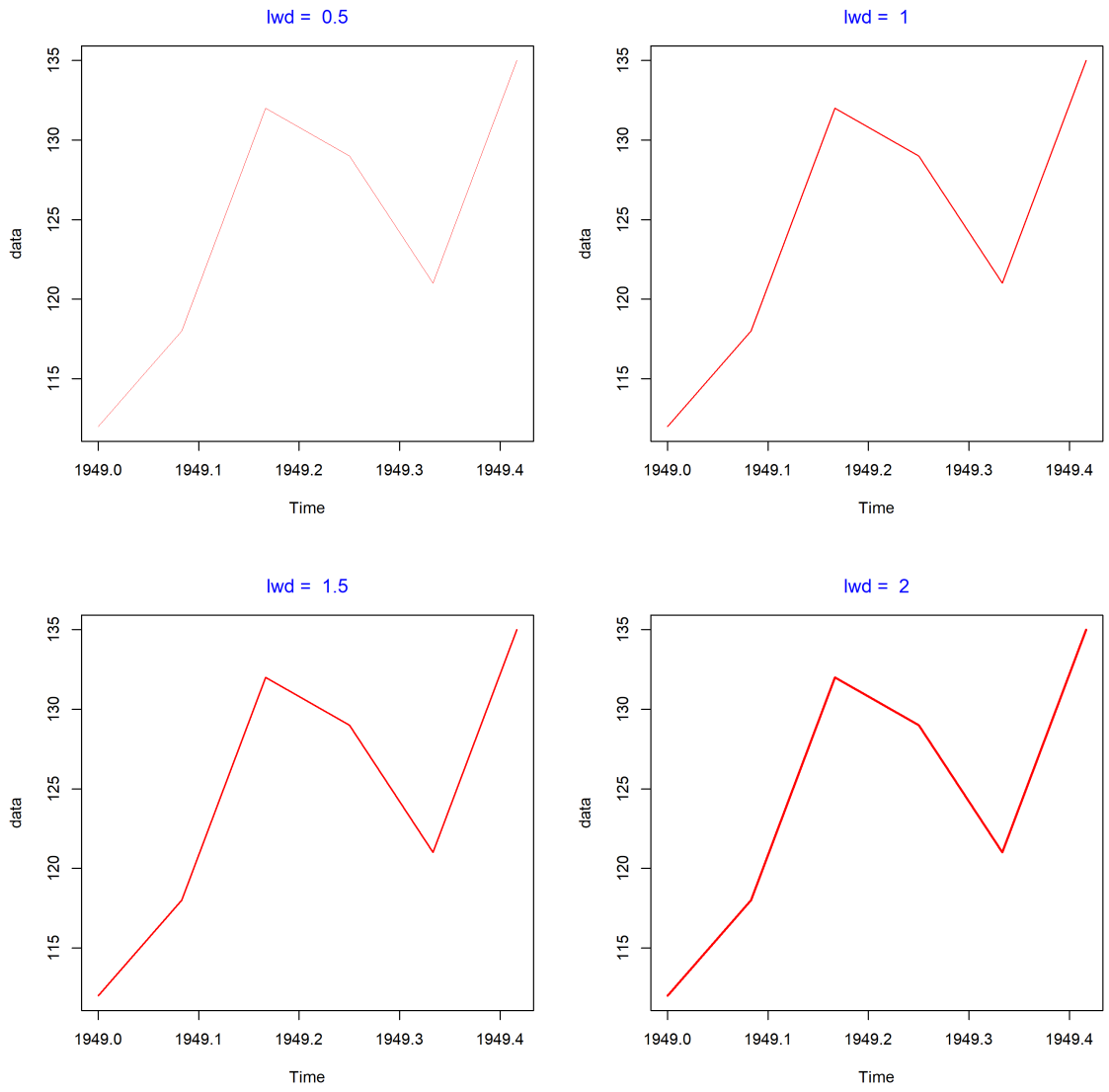
Line Width

The width of the lines can be modified using the `lwd` argument in the `plot()` function. The default value for width is 1.

```
plot(data, type = 'l', lwd = 2.5)
```



In the below example, we look at the width of the lines relative to the default value.



Enhance Points

We can enhance the points in the line plot in the same way as we enhanced the points in the scatter plot in this previous chapter. Let us look at an example:

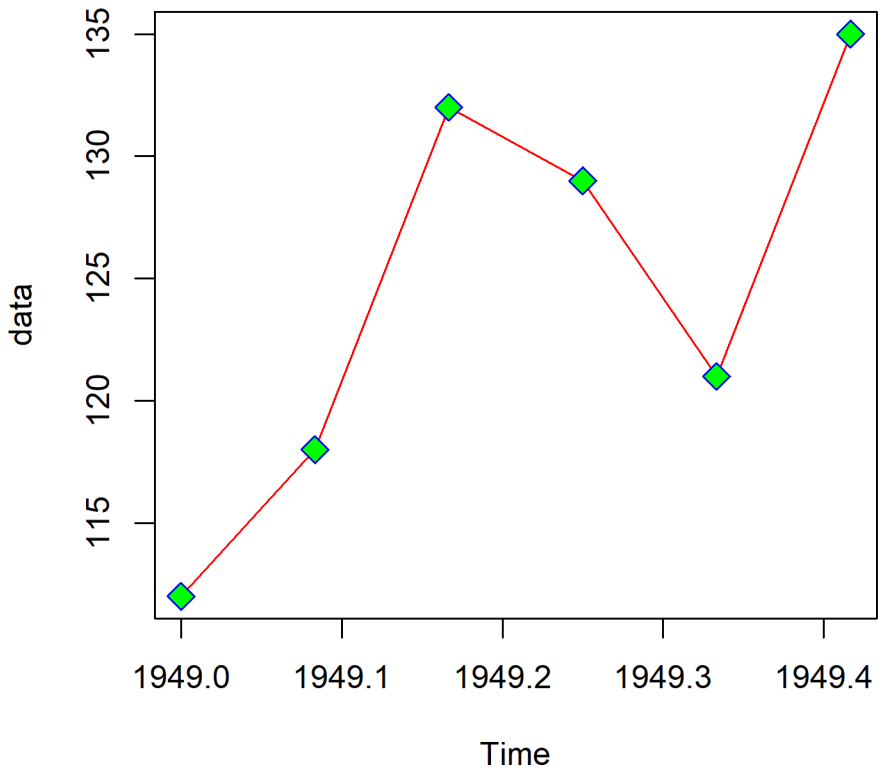
```
plot(data, type = 'b', pch = 23, col = 'red', cex = 1.5)
```

We have used the `pch`, `col` and `cex` arguments to modify the shape, color and size of the points. One drawback of the above method is that the color of the line and the points will be the same. What if we want them to have different colors? The solution is as follows:

- build the line graph using the `plot()` function
- add the points to the above plot using the `points()` function

In the next example, let us first build the line plot, add points using the `points()` function and then specify separate colors to the line and the points.

```
plot(data, type = 'l', col = 'red')
points(data, pch = 23, col = 'blue', bg = 'green', cex = 1.5)
```



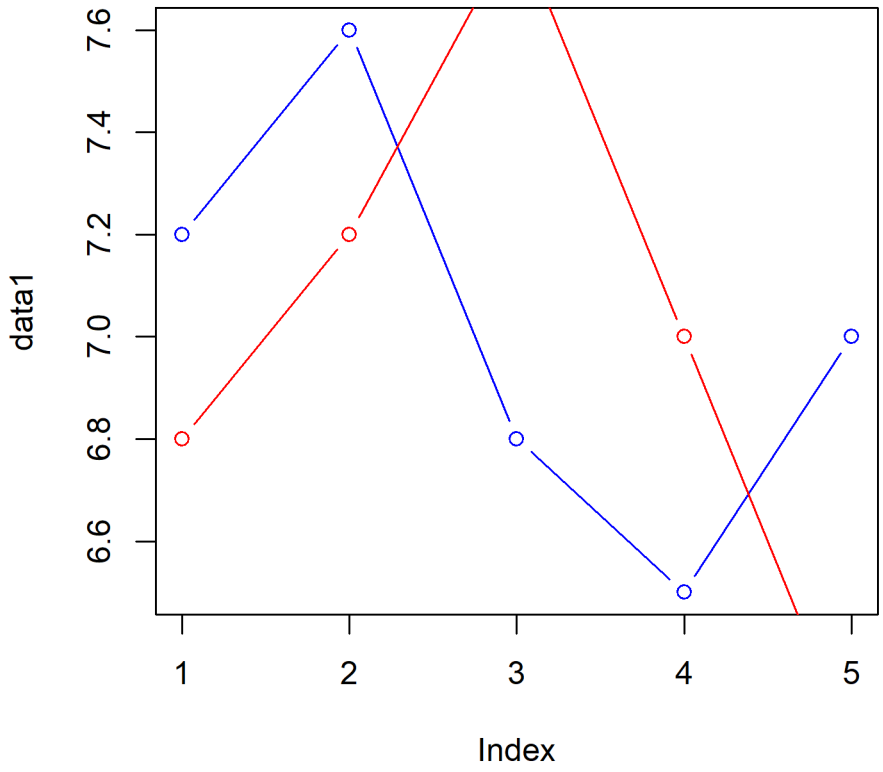
Additional Lines

If you want to compare variables, you would want to add additional lines to the line graph. In R, this can be achieved using the `lines()` function. First we create the line plot using the base variable and then we can add as many lines as we want using the `lines()` function.

Before you add additional lines, it is important to ensure that the range of both the axis are modified to accommodate the data of the additional lines. If we do not modify the axis range, some of the lines will be outside the plot.

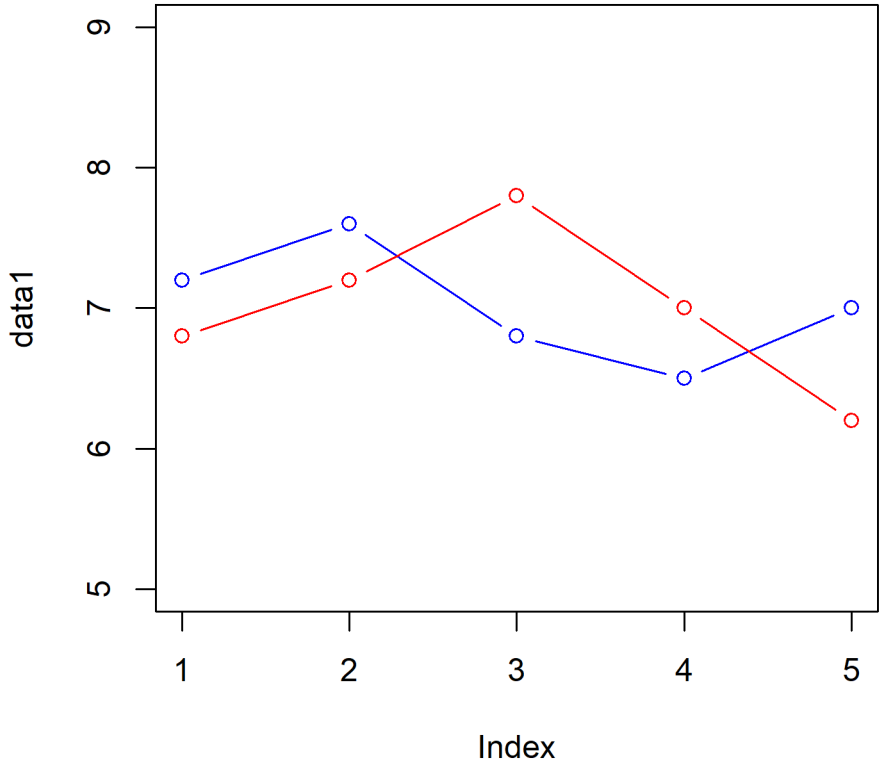
Let us now create a line plot and add an additional line using the `lines()` function.We will use some dummy data for this example:

```
data1 <- c(7.2, 7.6, 6.8, 6.5, 7)
data2 <- c(6.8, 7.2, 7.8, 7, 6.2)
plot(data1, type = "b", col = "blue")
lines(data2, type = "b", col = "red")
```



As you can see the second line is outside the plot. Let us recreate the plot but this time we will modify the range of the axis to accommodate the second line (data2).

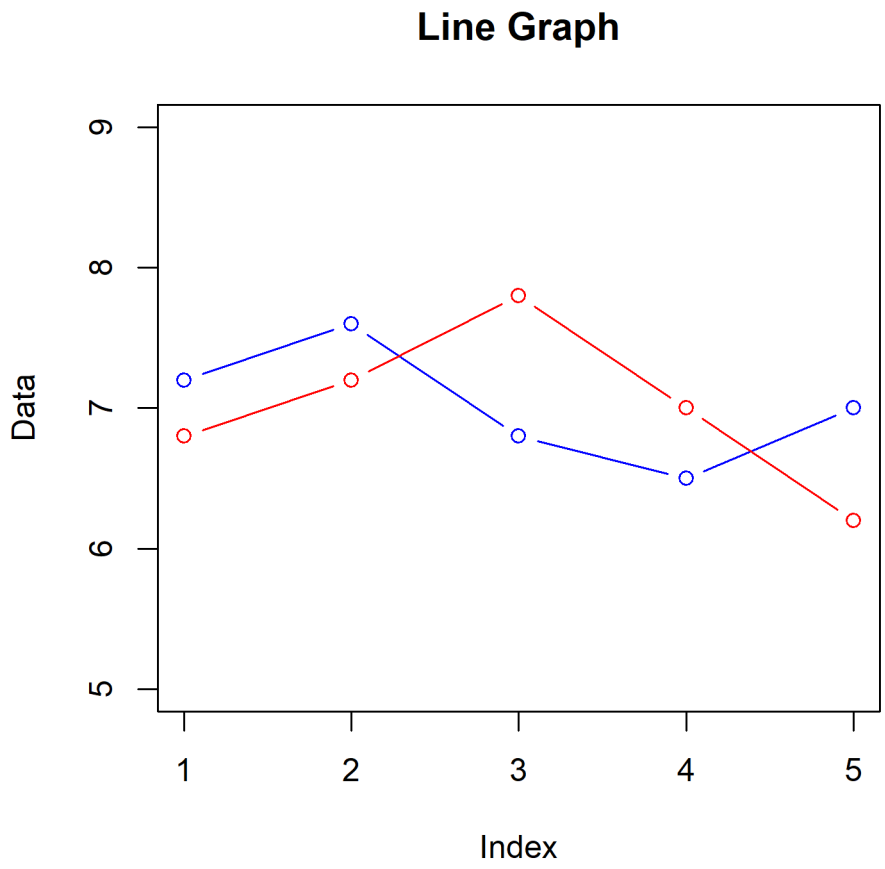
```
plot(data1, type = "b", col = "blue", ylim = c(5, 9))
lines(data2, type = "b", col = "red")
```



Putting it all together...

Finally let us enhance the plot by adding a title and modifying the axis labels.

```
plot(data1, type = "b", col = "blue",
     ylim = c(5, 9), main = 'Line Graph',
     xlab = 'Index', ylab = 'Data')
lines(data2, type = "b", col = "red")
```



Bar Plots

Introduction

In this chapter, we will visualize categorical data using univariate and bivariate bar plots. More specifically, we will learn to:

- create
  - simple bar plot
  - stacked bar plot
  - grouped bar plot
- modify bar
  - direction
  - color
  - line color
  - width
  - labels
- modify axis range

- remove axes from the plot
- specify the line type of the X axes
- offset the Y axes
- modify legend

Bar Plot

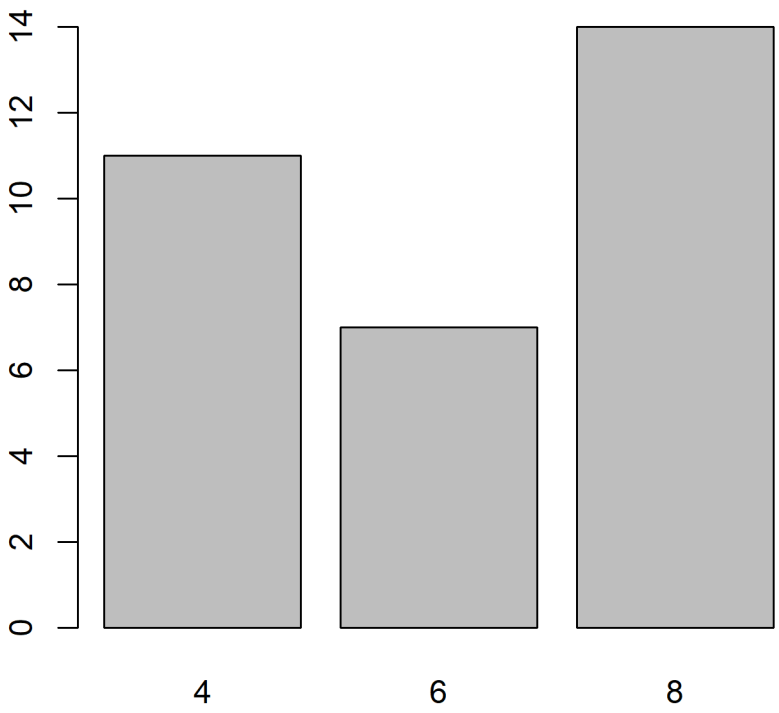
A bar plot represents data in rectangular bars. The length of the bars are proportional to the values they represent. Bar plots can be either horizontal or vertical. The X axis of the plot represents the levels or the categories and the Y axis represents the frequency/count of the variable.

A univariate bar plot represents a single categorical variable. The categories in the variable are represented on the X axis and their frequencies on the Y axis. In the below example, the `cyl` variable from the `mtcars` data set is visualized using a bar plot. The categories or levels are `4`, `6` and `8` which represent the number of cylinders in the automobile and are represented on the X axis. The frequency for each type of cylinder is represented by the Y axis.

In R, bar plots can be created using either the `plot()` or `barplot()` function. The input to both the functions are different. In case of the `plot()` function, we can specify the variable but it must be converted to a factor variable. In case of the `barplot()` function, the input must be the count or frequency of the variable. The `table()` function can be used to generate the counts/frequency for a variable. Let us use both the functions to create the bar plot:

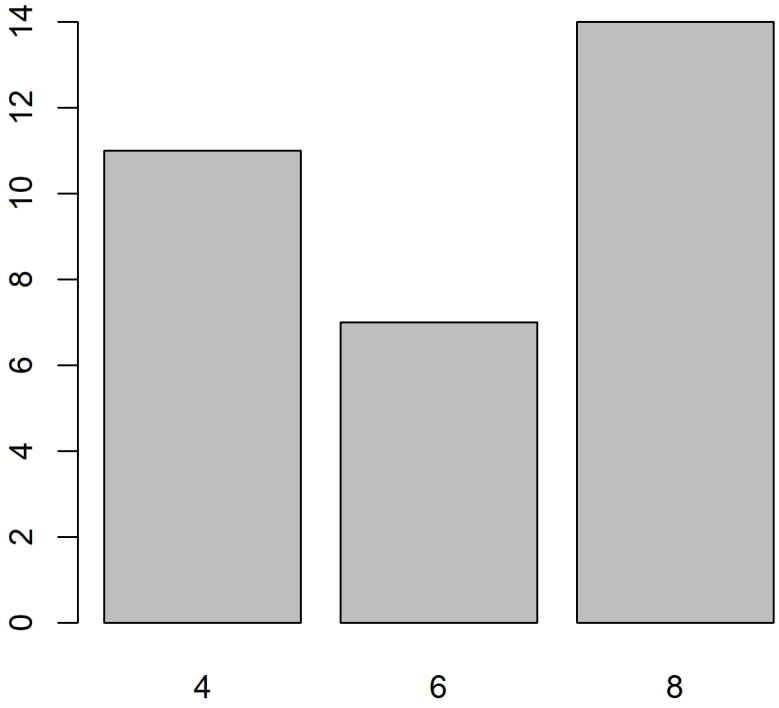
Using plot function

```
plot(as.factor(mtcars$cyl))
```



Using barplot function

```
barplot(table(mtcars$cyl))
```



If you observe carefully, the same plot is generated by both the functions. Before we explore the bar plots further, let us store the data in a new variable instead of using the `table()` function in every example:

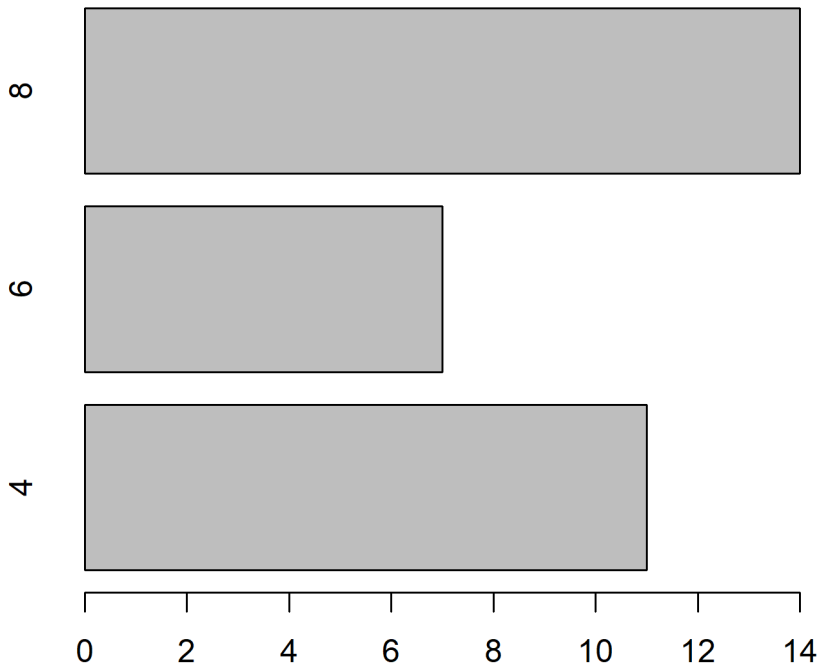
```
cyl_freq <- table(mtcars$cyl)
cyl_freq
```

```
 4  6  8
11  7 14
```

Horizontal or Vertical

Bar plots can be horizontal or veritcal (which is the default). Use the `horiz` argument in the `barplot()` function to build a horizontal bar plot. As you can see, the axis have been flipped. The Y axis represents the categories and the X axis represents their counts/frequencies.

```
barplot(cyl_freq, horiz = TRUE)
```

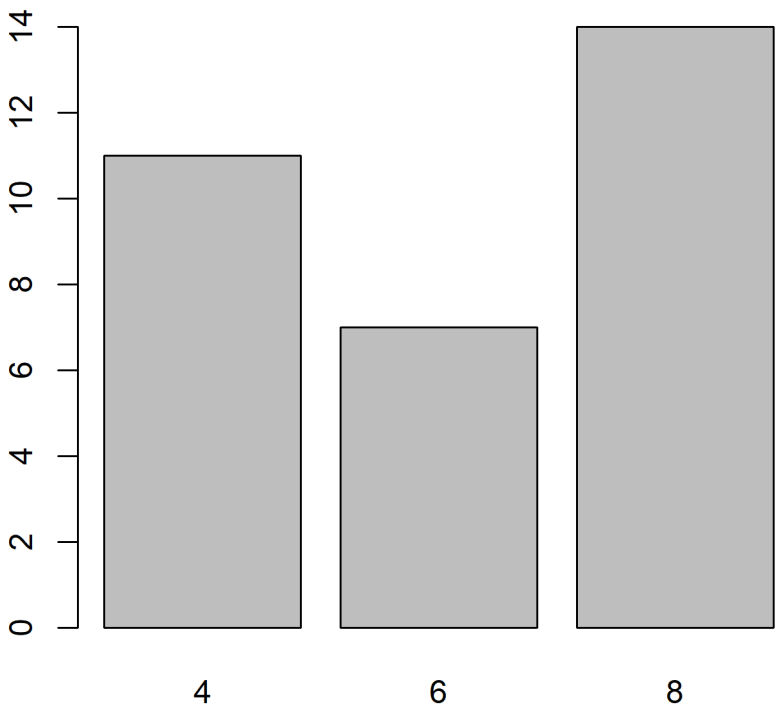


Bar Width

In the bar plot, the width of the bars and the space between them are same. A specific category of the variable can be highlighted by increasing/decreasing the width of the bar representing it. In our example, we will increase the width of the bar that represents automobiles with 8 cylinders. The `width` argument is used to specify the width of the bars. The width must be specified for all the bars in the plot. It must be a vector the length of which must be equal to the number of categories of the variable.

Equal Width

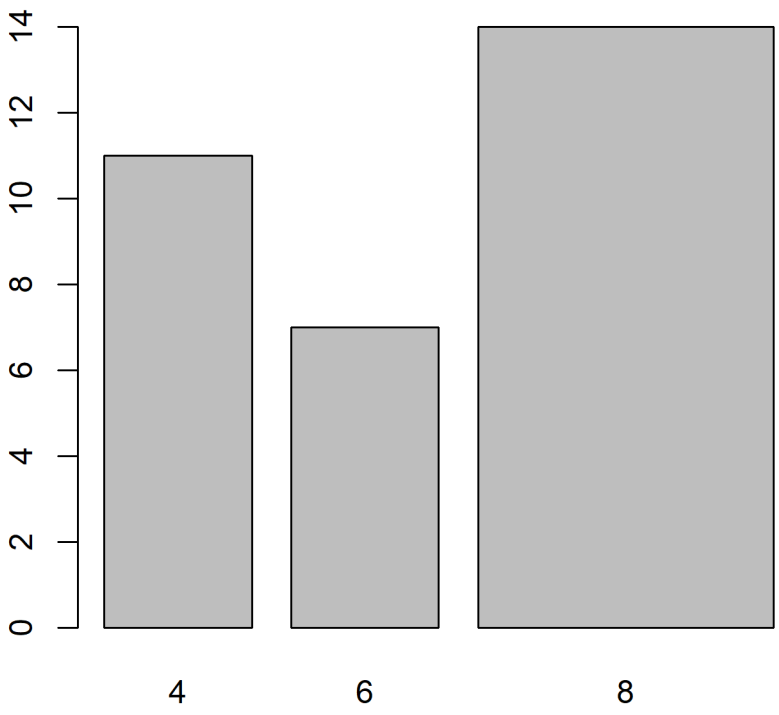
```
barplot(cyl_freq, width = 2)
```



Different Widths

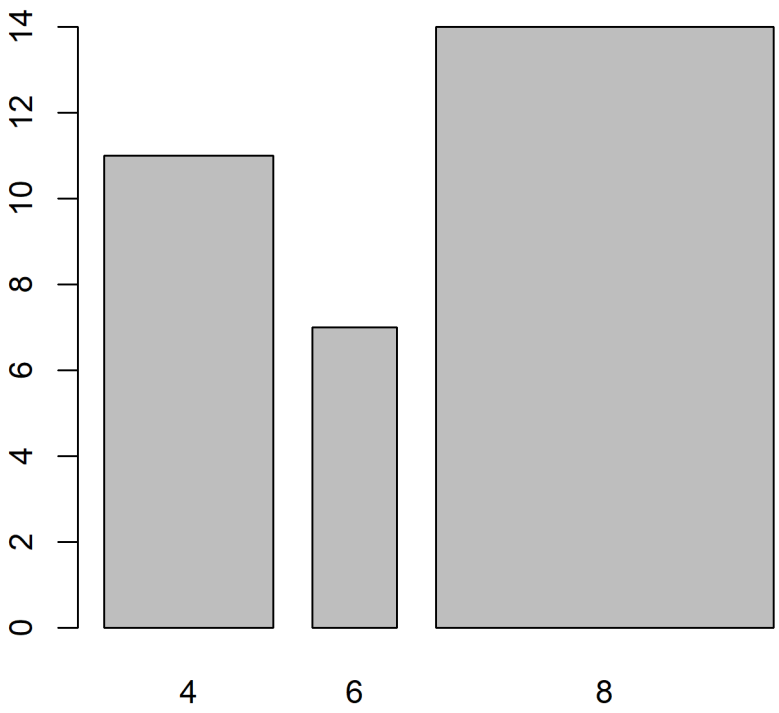
In the below example, the width of the third bar is twice the width of the other two bars

```
barplot(cyl_freq, width = c(1, 1, 2))
```



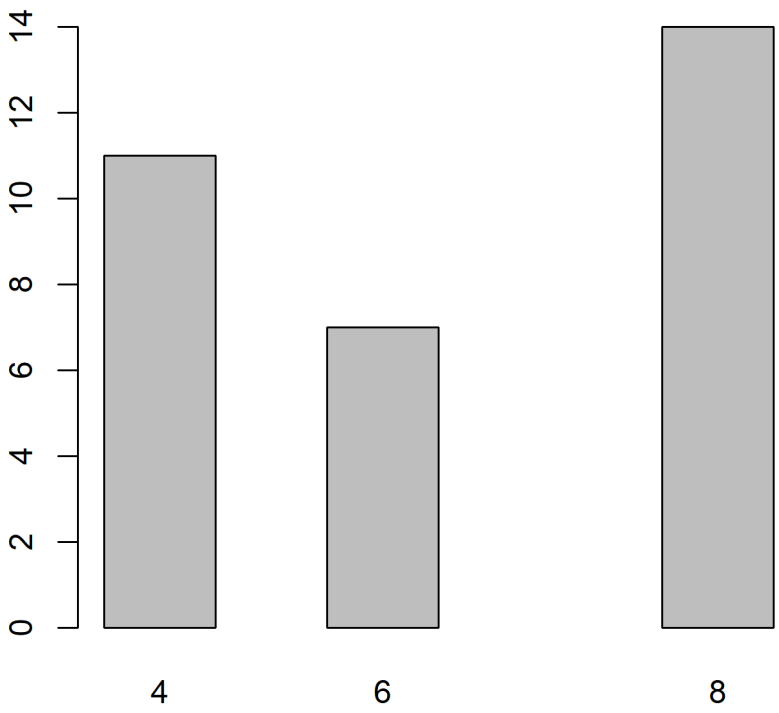
In the below example, the width of the second bar is half the width of the other first bar and the third bar is twice the width of the first bar.

```
barplot(cyl_freq, width = c(1, 0.5, 2))
```



The space between the bars can be specified in a similar manner but using the `space` argument in the `barplot()` function: In the below example, the space between the third bar and the second bar is twice the space between first and second bar.

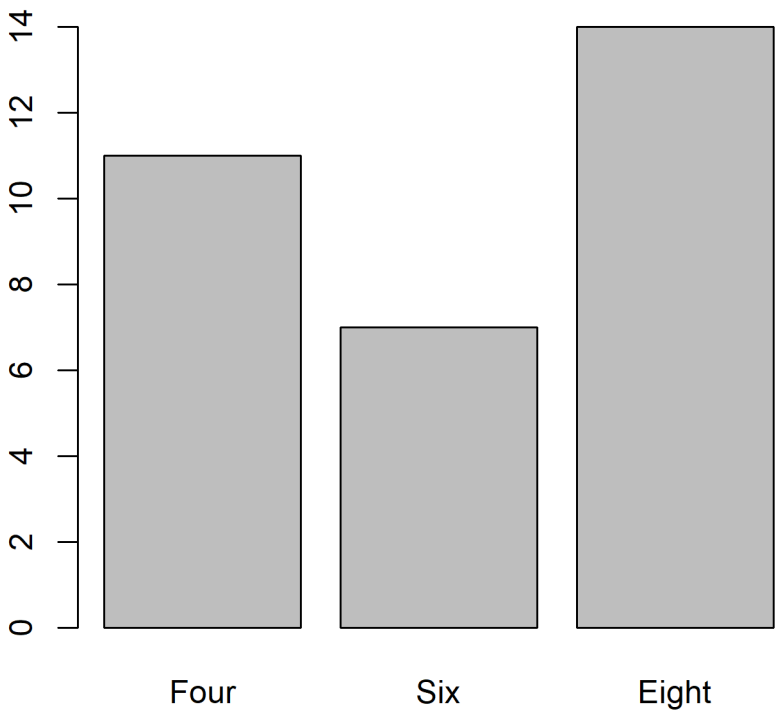
```
barplot(cyl_freq, space = c(1, 1, 2))
```



Labels

It is important to add appropriate labels to the bars in order to communicate properly. In our example, the bars represent automobiles with different number of cylinders. The labels likewise indicate the number of cylinders represented by the bars. In order to demonstrate how to add labels, we will change the labels from numbers to their corresponding words. The `names.arg` argument is used to add labels to the bars in a plot. Below is our example:

```
barplot(cyl_freq, names.arg = c('Four', 'Six', 'Eight'))
```



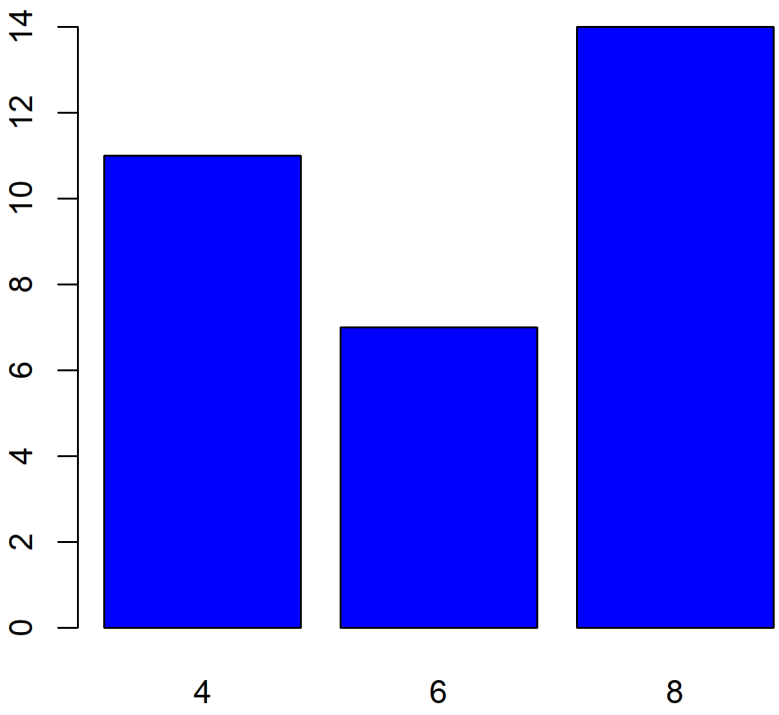
It is important to specify labels for all the bars in the plot else R will return an error.

Color

Let us add some color to the plots. In a bar plot, we can specify different colors for the bars and their borders. Use the `col` argument to add color to the bars.

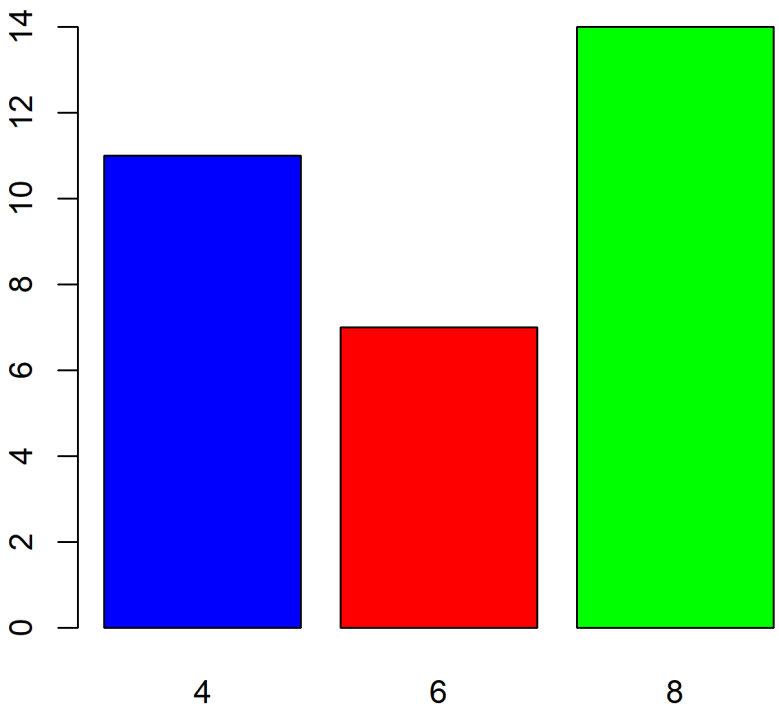
Same color for all bars

```
barplot(cyl_freq, col = 'blue')
```



Differnt color for the bars

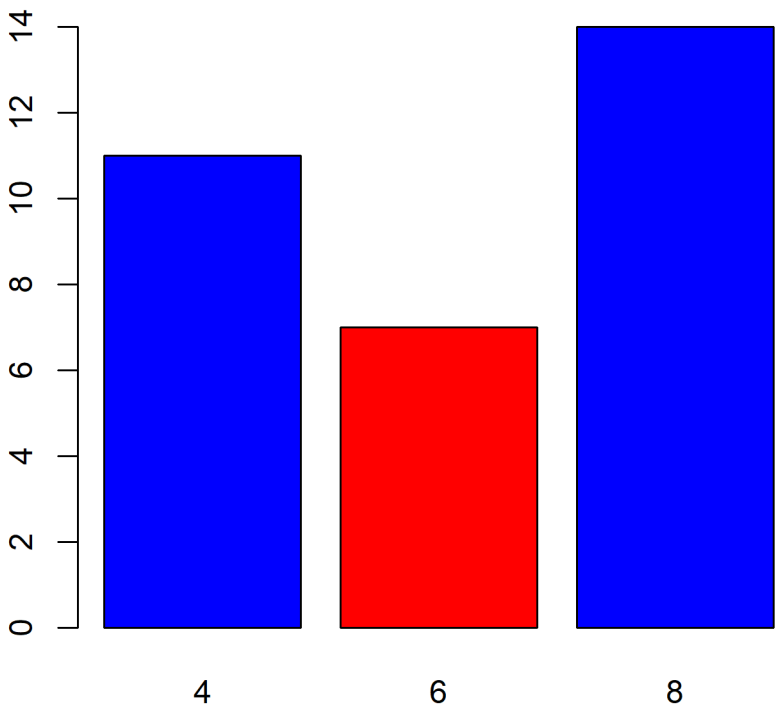
```
barplot(cyl_freq, col = c('blue', 'red', 'green'))
```



What happens if we do not specify color for all the bars? The colors you specify are recycled.

Recycling colors

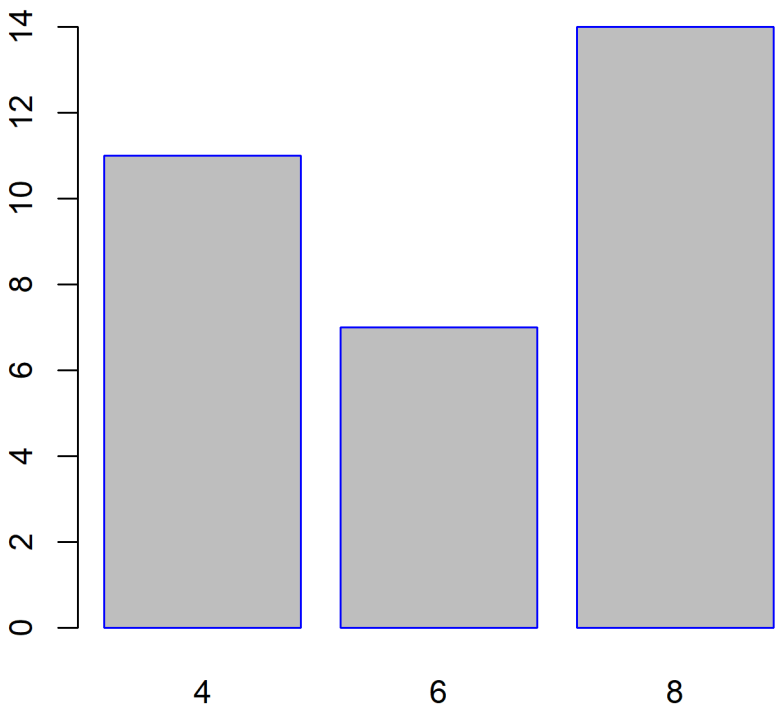
```
barplot(cyl_freq, col = c('blue', 'red'))
```



The `border` argument specifies the color of the border of the bars. The rules that apply to `col` argument apply here also. Below are the examples:

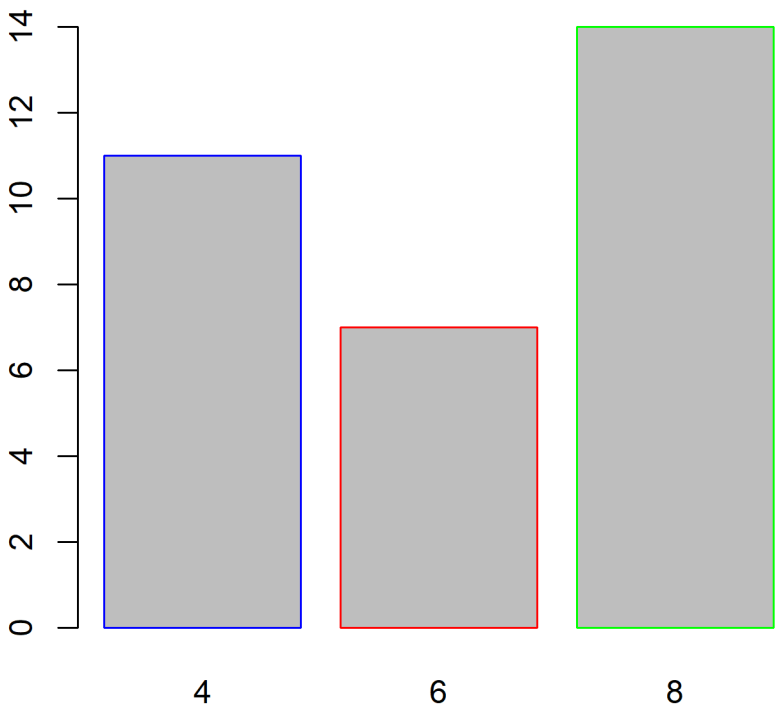
Same color for all bars

```
barplot(cyl_freq, border = 'blue')
```



Differnt color for the bars

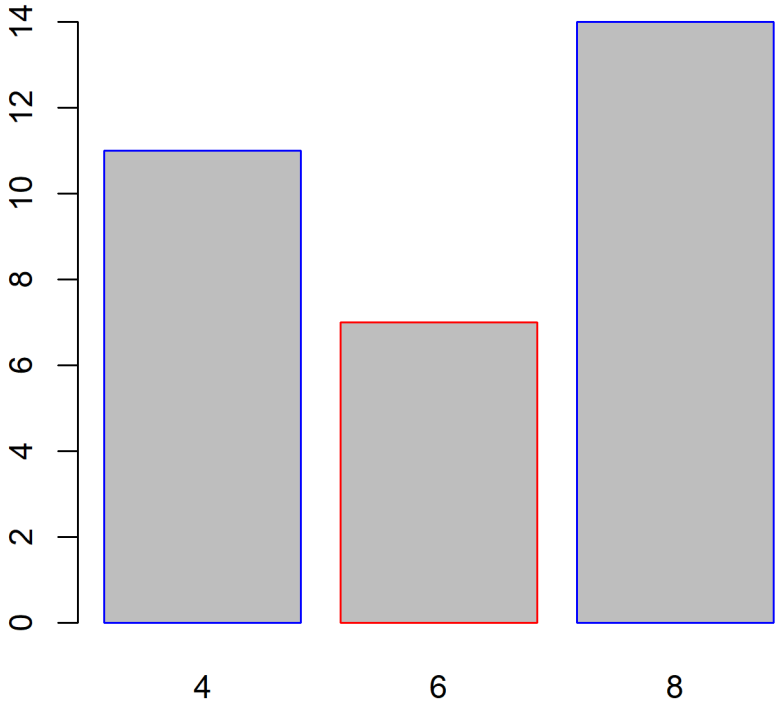
```
barplot(cyl_freq, border = c('blue', 'red', 'green'))
```



What happens if we do not specify color for all the bars? The colors you specify are recycled.

Recycling colors

```
barplot(cyl_freq, border = c('blue', 'red'))
```



Axes

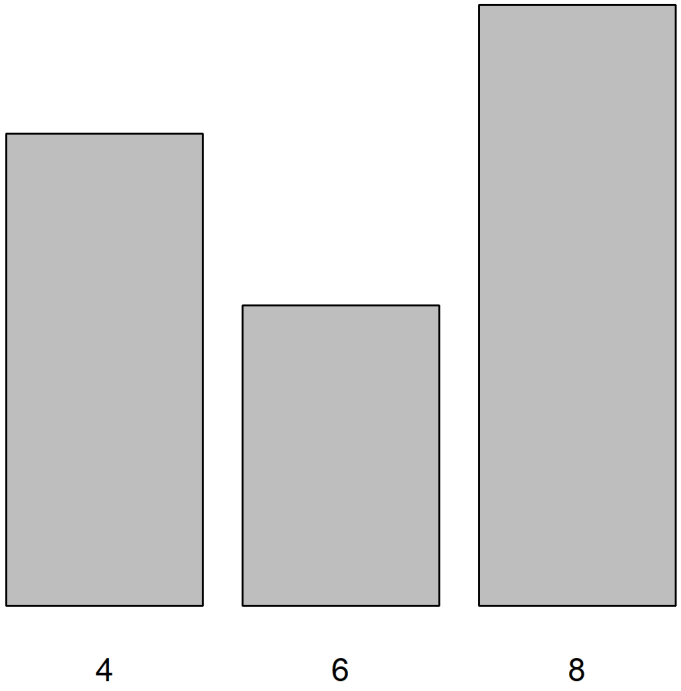
In this section, we will learn to

- remove axes from the plot
- specify the line type of the X axes
- offset the Y axes

Remove axes

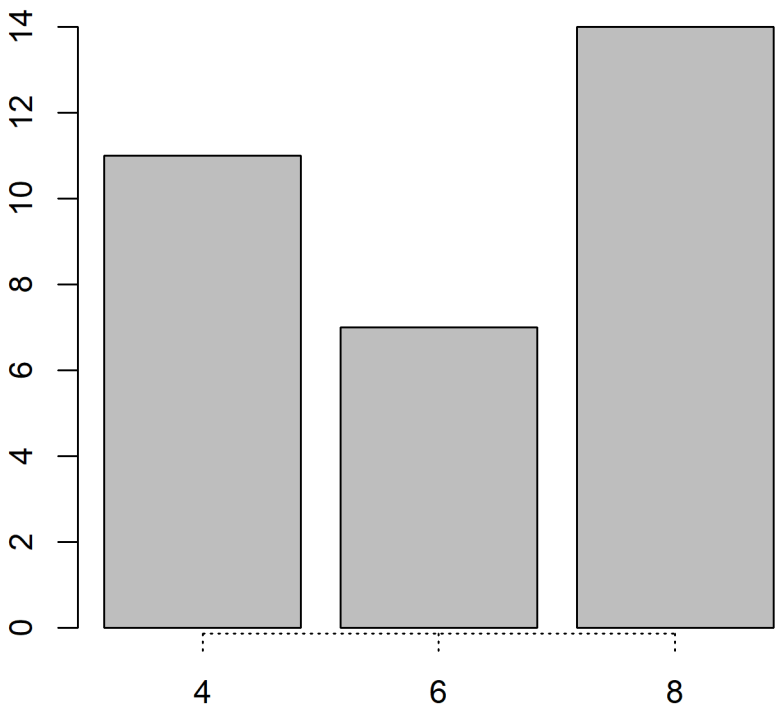
The `axes` argument can be used to retain/remove the axes from the plot. It takes logical values as input and the default is `TRUE`. Set it to `FALSE` to remove the axes from the plot:

```
barplot(cyl_freq, axes = FALSE)
```



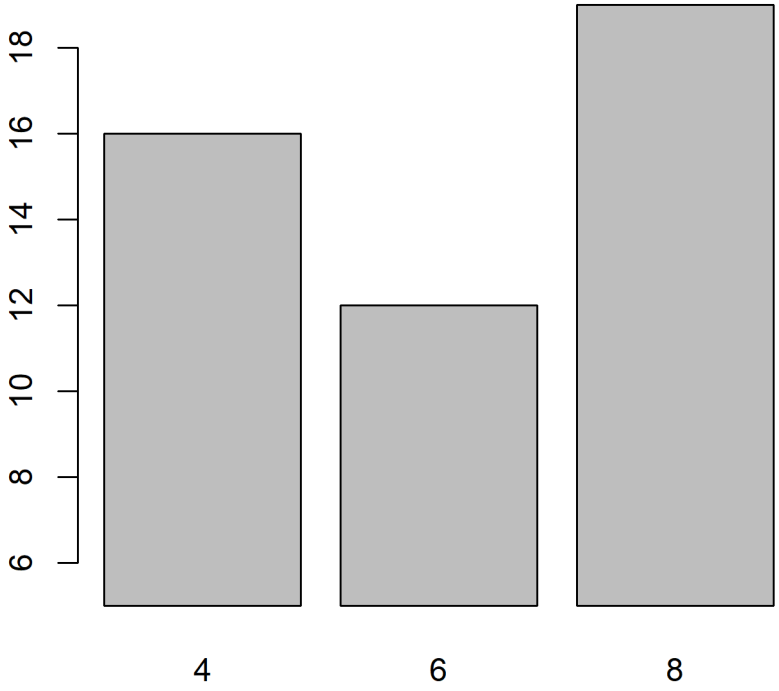
If we decide to retain the axes, the line type of the X axes can be specified using the `axis.lty` argument. It does not modify the line type of the Y axes and it will not work if the `axes` argument is set to `FALSE`.

```
barplot(cyl_freq, axis.lty = 3)
```



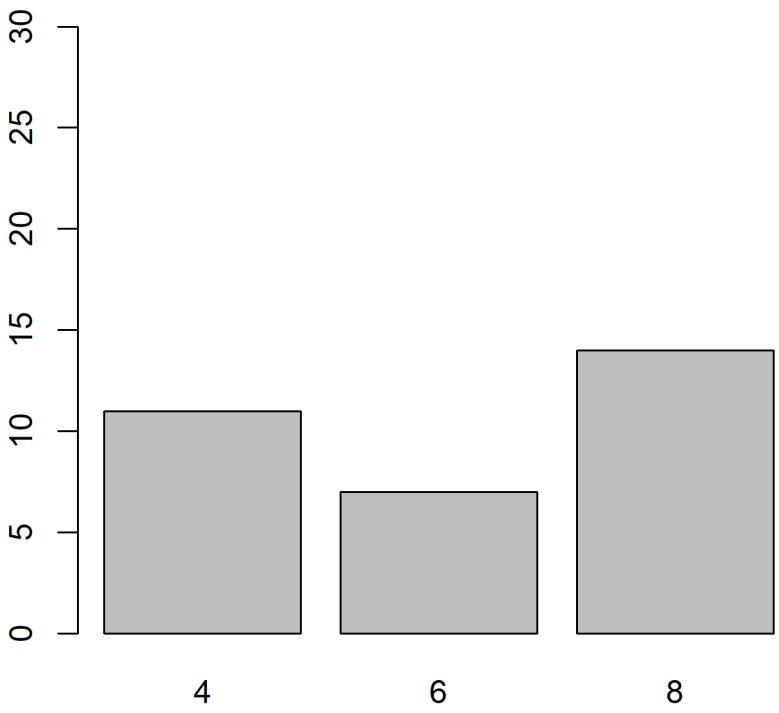
Though we cannot modify the line type of the Y axes, we can offset it using the `offset` argument. In the below example, we will offset the Y axes and you can observe that the minimum value of the Y axes is now 5 instead of 0.

```
barplot(cyl_freq, offset = 5)
```



You can similarly modify the range of the Y axes using the `ylim` argument. Although in case of bar plots, modifying the range of the plot may not be very useful.

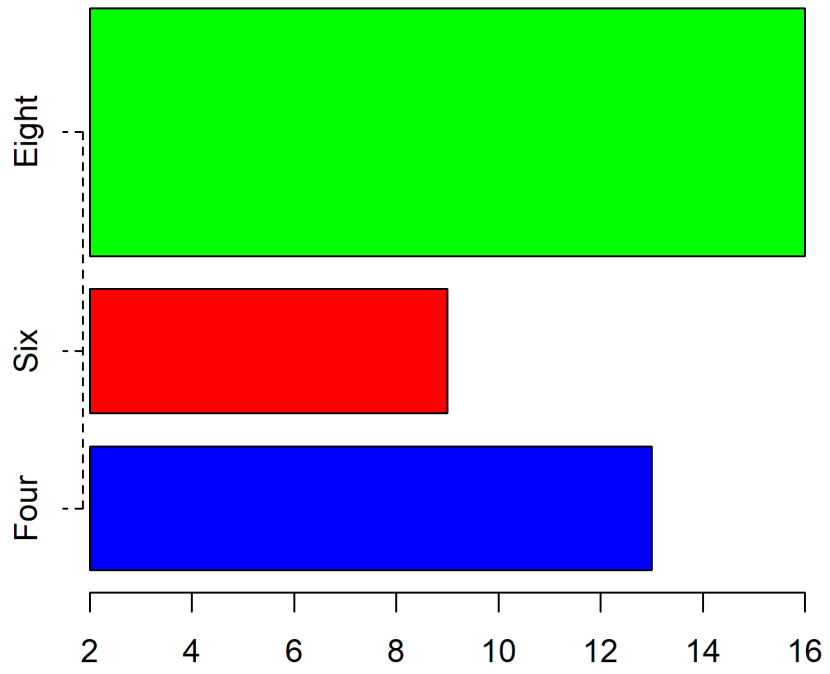
```
barplot(cyl_freq, ylim = c(0, 30))
```



Putting it all together...

Let us quickly revise what we have learnt so far and build a bar plot for visualizing the `cyl` variable in the `mtcars` data set:

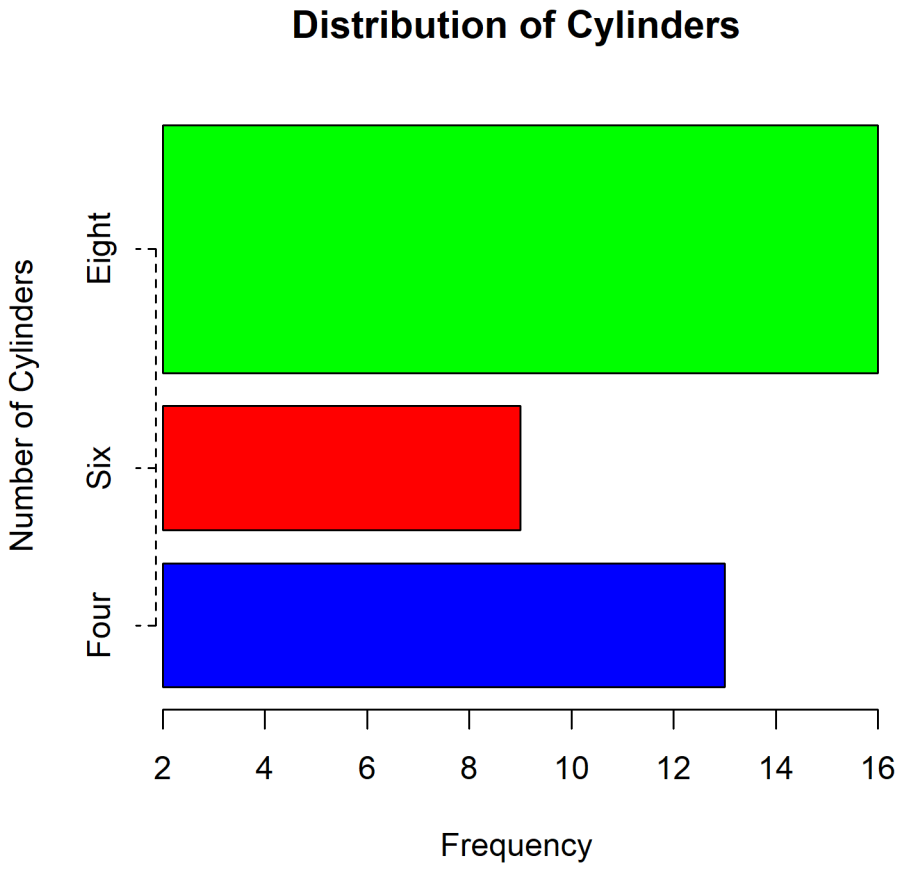
```
barplot(cyl_freq, col = c('blue', 'red', 'green'),
        horiz = TRUE, width = c(1, 1, 2),
        names.arg = c('Four', 'Six', 'Eight'),
        axis.lty = 2, offset = 2)
```



Title & Axis Labels

Well the plot looks good but for someone who does not know the underlying data, it will diffficult to understand what is being communicated. Let us add a title and labels for the axes.

```
barplot(cyl_freq, col = c('blue', 'red', 'green'),
        horiz = TRUE, width = c(1, 1, 2),
        names.arg = c('Four', 'Six', 'Eight'),
        axis.lty = 2, offset = 2)
title(main = 'Distribution of Cylinders',
      xlab = 'Frequency', ylab = 'Number of Cylinders')
```



Bivariate Bar Plots

A bivariate bar plot represents the cross table or two way table of categorical variables. They are of two types:

- Stacked Bar Plots
- Grouped Bar Plots

Before we look at bivariate bar plots, let us create a two way table of `cyl` (number of cylinders) and `gear` (number of gears) using the `table()` function:

```
table(mtcars$gear)
```

```
 3  4  5
15 12  5
```

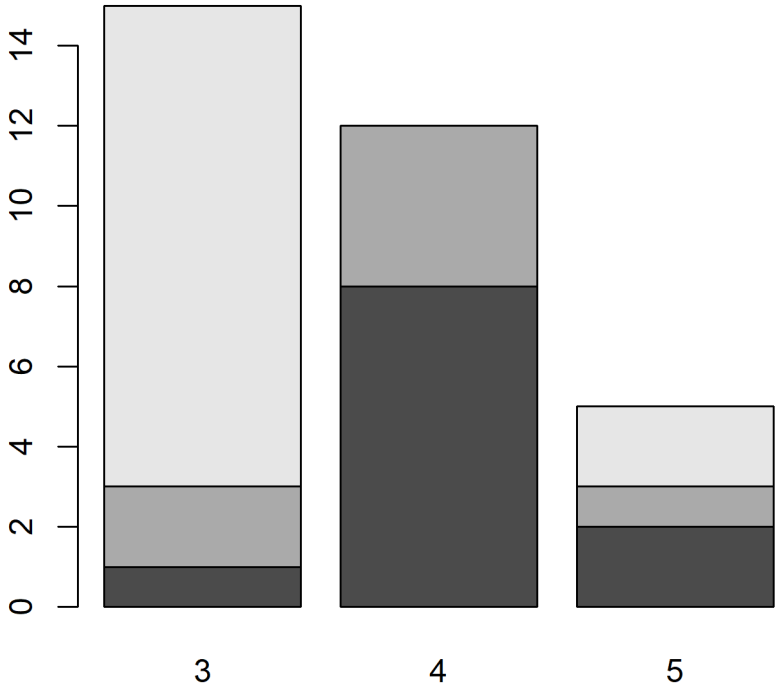
```
cyl_gear <- table(mtcars$cyl, mtcars$gear)
cyl_gear
```

```
   3  4  5
4  1  8  2
6  2  4  1
8 12  0  2
```

The number of gears is represented by the columns in the table and the numbe rof cylinders is represented by the rows.

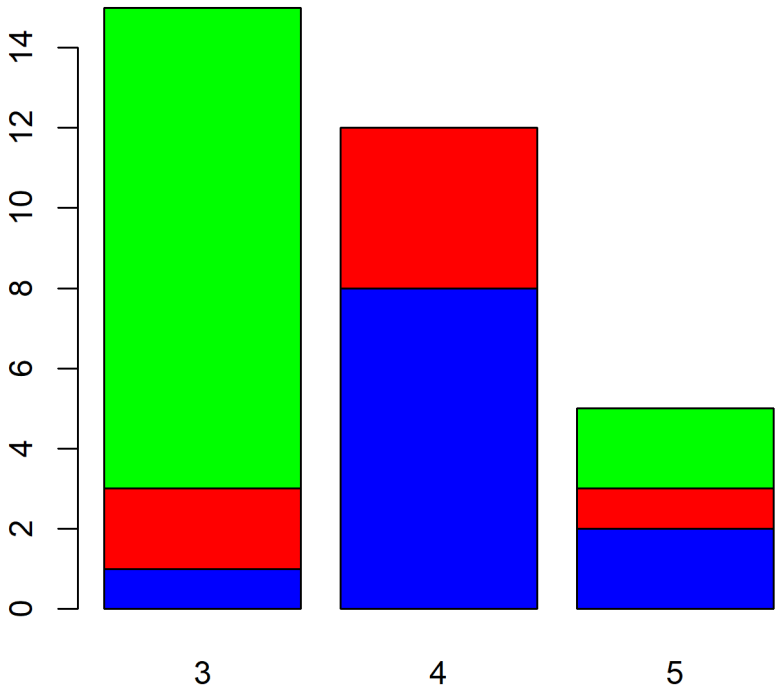
Stacked Bar Plot

```
barplot(cyl_gear)
```



The bars in the plot represent the distribution of `cyl` for each level of category of the `gear` variable. The first bar represents the distribution of cylinders for automobiles with 3 gears. From the two way table we saw earlier, the columns are the bars. The rows are represented by different sections of the bar. Let us add some colors to the plot as the default colors of the plot are not very intuitive. It will also allow us to clearly examine the distribution of `cyl` for the different levels of `gear`.

```
barplot(cyl_gear, col = c('blue', 'red', 'green'))
```

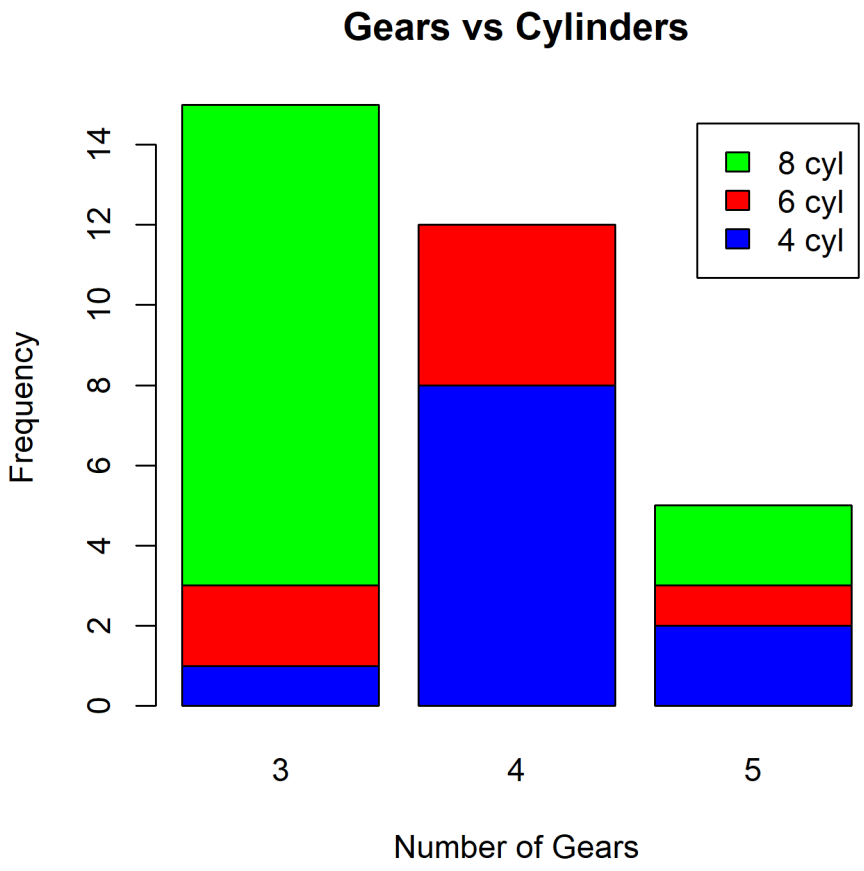


If you carefully observe the table and the plot:

- the blue sections of the bars represent the number of automobiles with 4 cylinders (in each gear group)
- the red sections represent the number of automobiles with 6 cylinders (in each gear group)
- the green sections represent the number of automobiles with 8 cylinders (in each gear group)

We need to convey the above information in some way and will do that using the `legend.text` argument. It takes logical values as inputs and the default values is `FALSE`. It adds a legend to the plot when it is set to `TRUE`. In the next example, we add a legend as well as other relevant information such as title and axis labels.

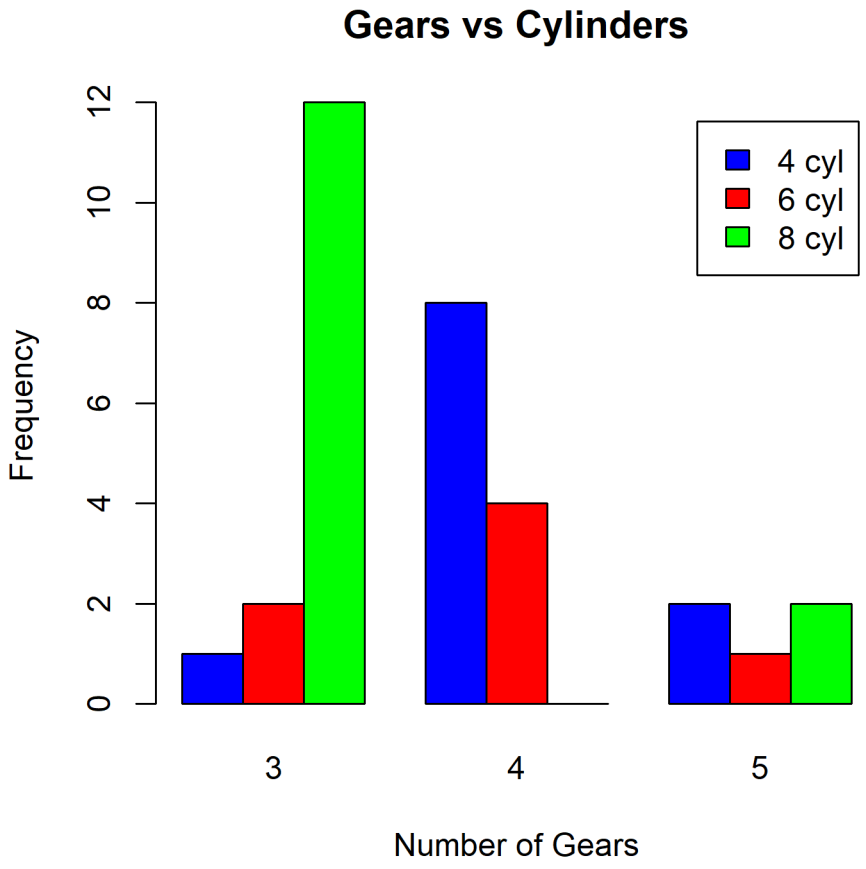
```
barplot(cyl_gear, col = c('blue', 'red', 'green'),
        main = 'Gears vs Cylinders', legend.text = c('4 cyl', '6 cyl', '8 cyl'),
        xlab = 'Number of Gears', ylab = 'Frequency')
```



Grouped Bar Plot

A grouped bar plot represents the same data as the stacked bar plot but instead of being stacked, the bars are now grouped and placed besides each other.

```
barplot(cyl_gear, col = c('blue', 'red', 'green'),
        beside = TRUE, legend.text = c('4 cyl', '6 cyl', '8 cyl'),
        main = 'Gears vs Cylinders',
        xlab = 'Number of Gears', ylab = 'Frequency')
```



The `beside` argument in `barplot()` function is set to `TRUE` to build grouped bar plots. It takes logical values as inputs and the default values is FALSE. As you can observe from the plot, the bars are placed besides each other instead of being stacked.

Box Plots

Introduction

In this chapter, we will learn to

- create univariate/multivariate box plots
- interpret box plots
- create horizontal box plots
- detect outliers
- modify box color
- use formula to compare distributions of different variables
- use notches to compare medians

Box Plot

The box plot is a standardized way of displaying the distribution of data based on the five number summary: minimum, first quartile, median, third quartile, and maximum. Box plots are useful for detecting outliers and for comparing distributions. It shows the shape, central tendancy and variability of the data.

Structure

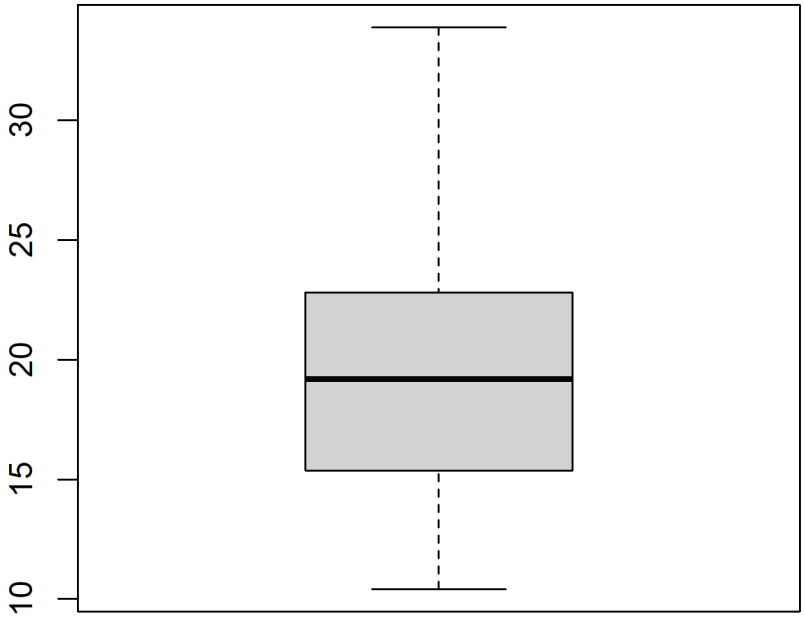
A boxplot splits the data set into quartiles. The body of the boxplot consists of a "box" (hence, the name), which goes from the first quartile (Q1) to the third quartile (Q3). Within the box, a vertical line is drawn at the Q2, the median of the data set. Two horizontal lines, called whiskers, extend from the front and back of the box. The front whisker goes from Q1 to the smallest non-outlier in the data set, and the back whisker goes from Q3 to the largest non-outlier. If the data set includes one or more outliers, they are plotted separately as points on the chart.

Univariate Box Plot

Basic Plot

Let us begin by creating a basic box plot. We will use the `boxplot()` function and specify the data.

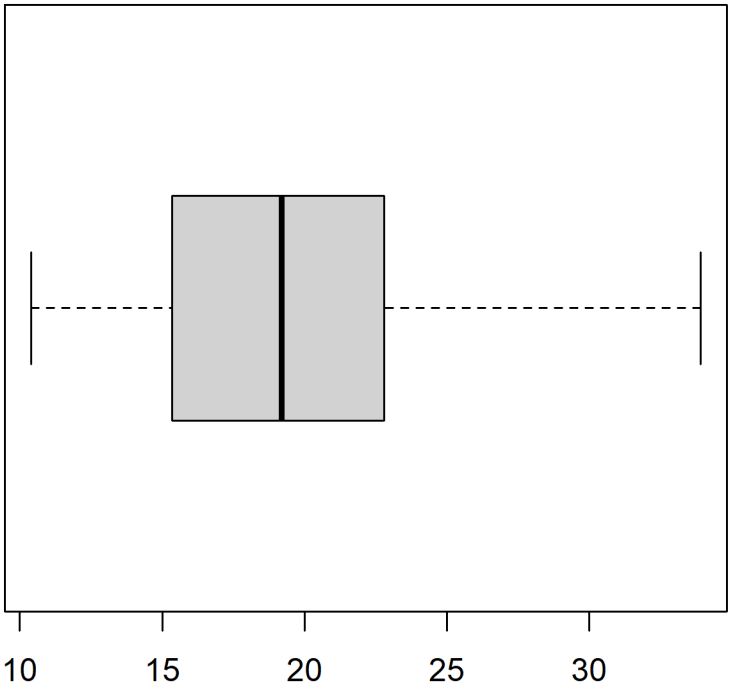
```
boxplot(mtcars$mpg)
```



Horizontal Box Plot

Use the `horizontal` argument in the `boxplot()` function to create a horizontal box plot.

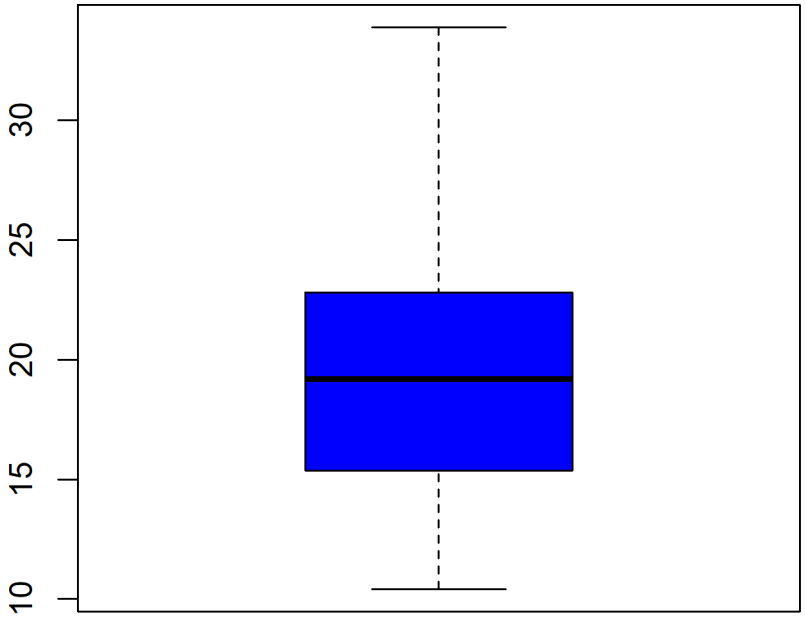
```
boxplot(mtcars$mpg, horizontal = TRUE)
```



Color

Let us add some color to the boxplot. Use the `col` argument to specify a color for the plot.

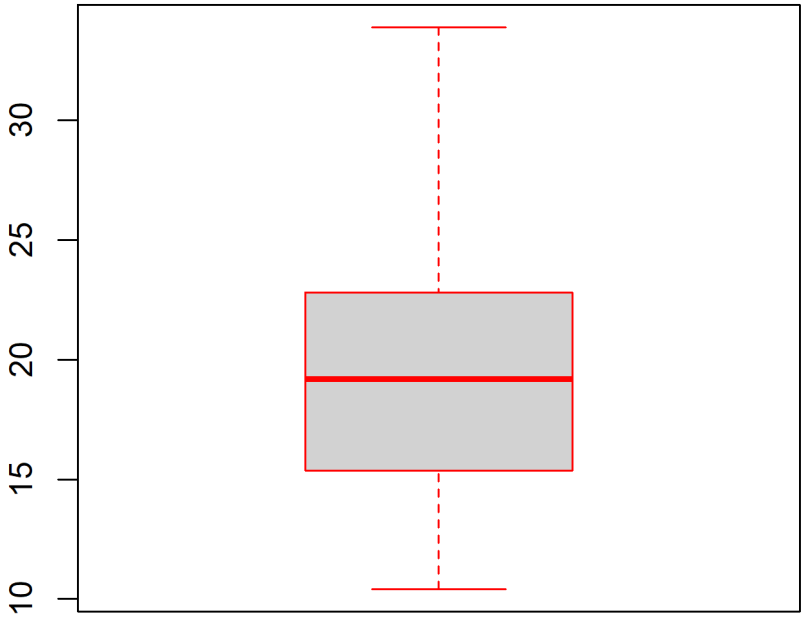
```
boxplot(mtcars$mpg, col = 'blue')
```



Border Color

We can specify a separate color for the border of the box in the boxplot. To modify the border color, use the `border` argument.

```
boxplot(mtcars$mpg, border = 'red')
```

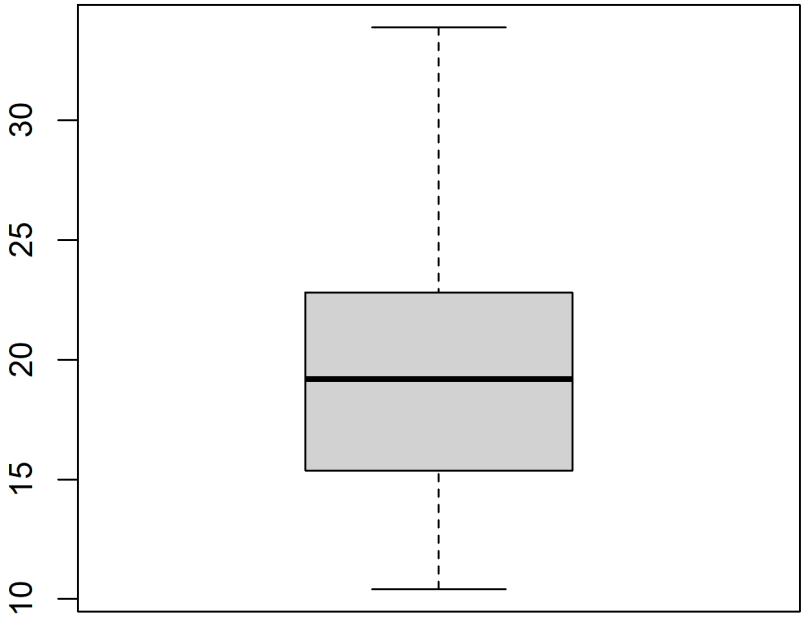


Range

The `range` argument determines how far the plot whiskers extend out from the box. If range is positive, the whiskers extend to the most extreme data point which is no more than range times the interquartile range from the box. A value of zero causes the whiskers to extend to the data extremes.

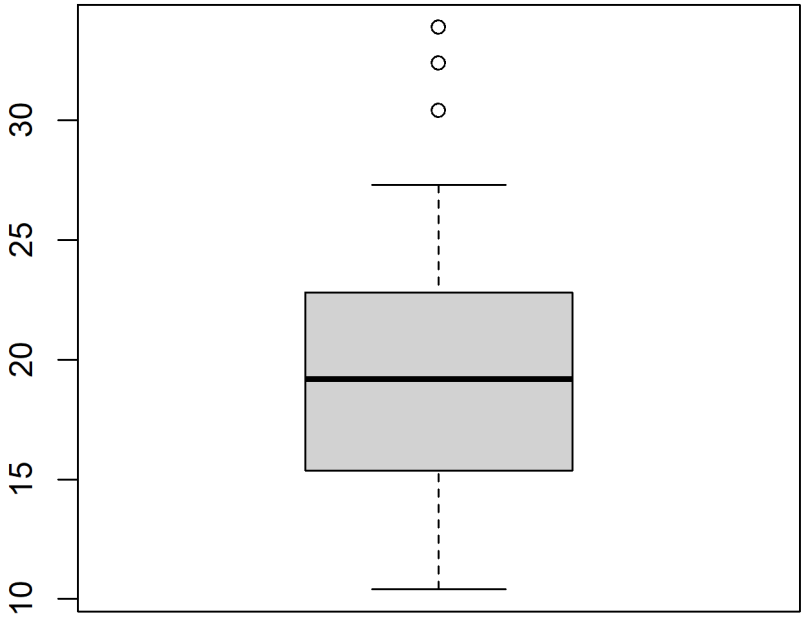
Let us set the value of `range` to 0 and observe the plot.

```
boxplot(mtcars$mpg, range = 0)
```

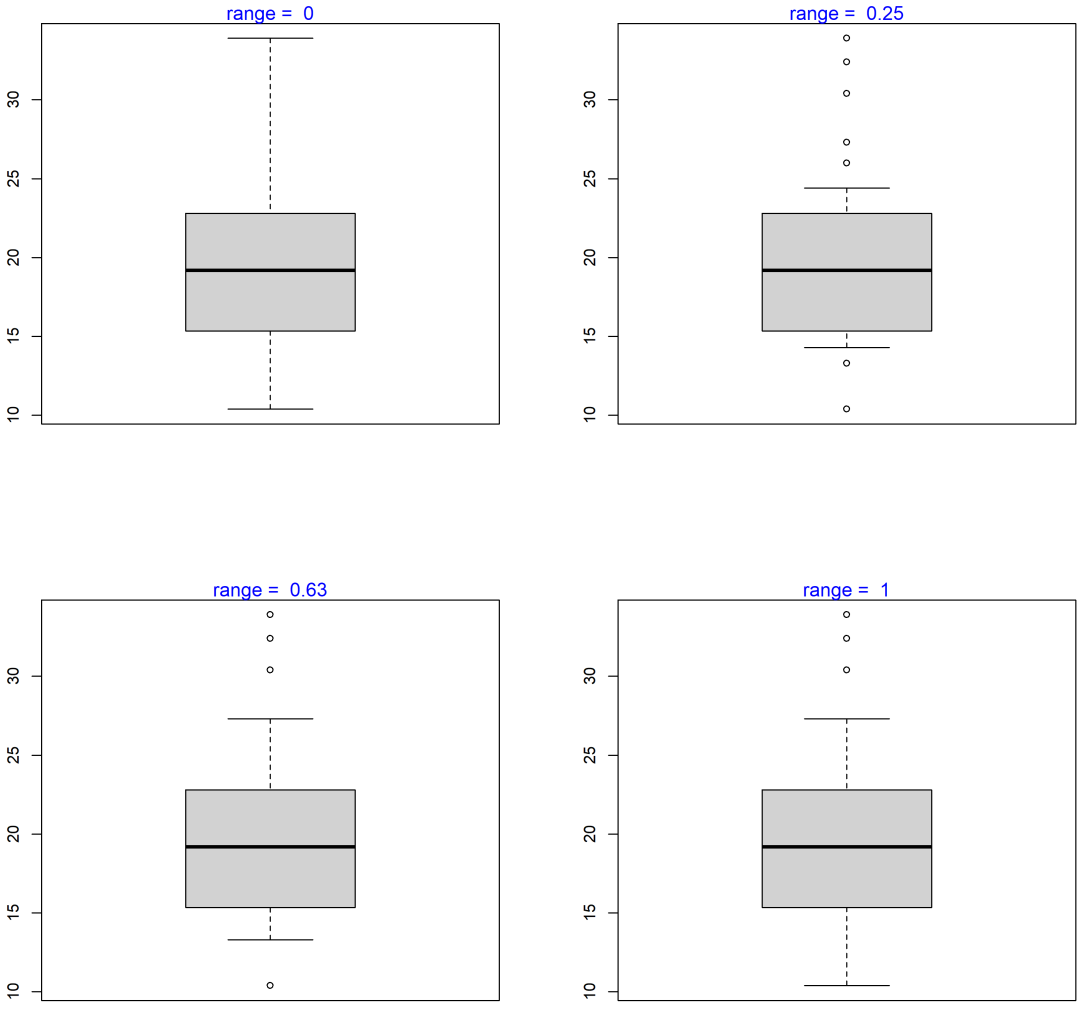


In the below plot, we set the value of `range` to 1.

```
boxplot(mtcars$mpg, range = 1)
```



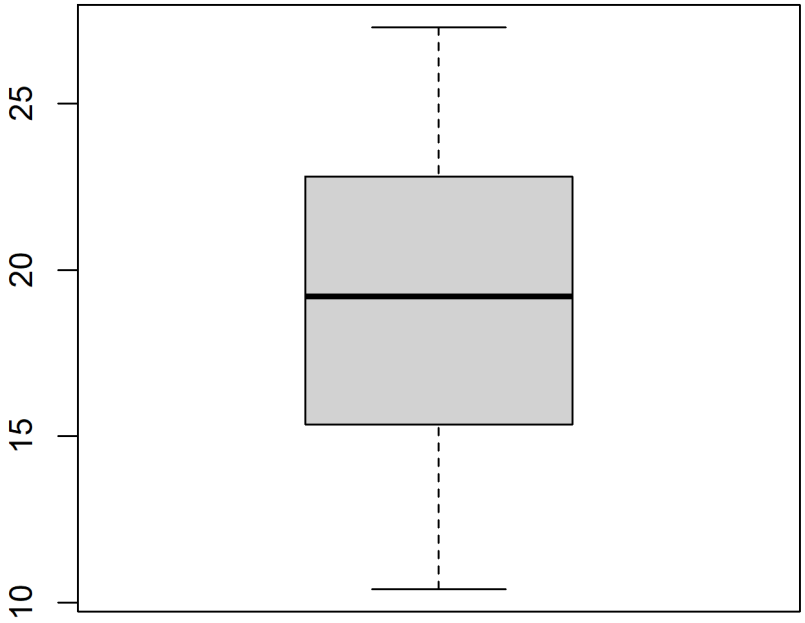
Let us observe how the plot appears as we change the value of `range` from 0 to 1.



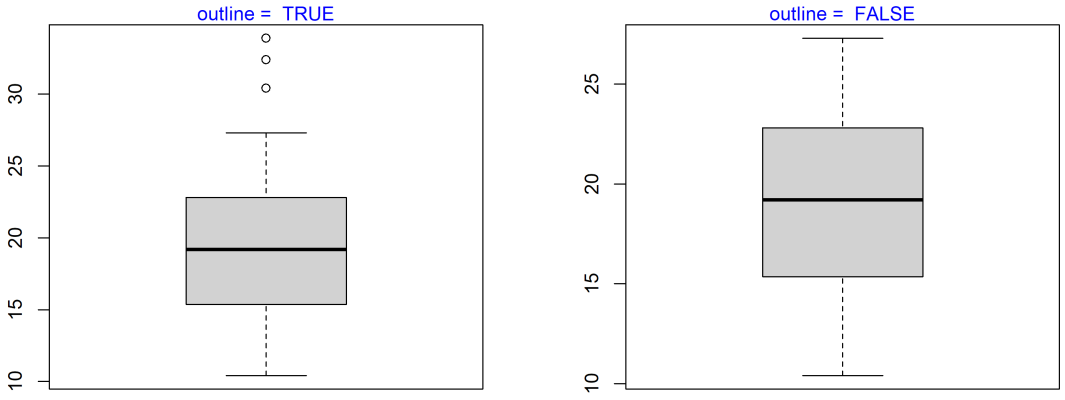
Outline

The outliers in the plot are not drawn if the `outline` argument is set to `FALSE`. The default value is `TRUE`.

```
boxplot(mtcars$mpg, range = 1, outline = FALSE)
```

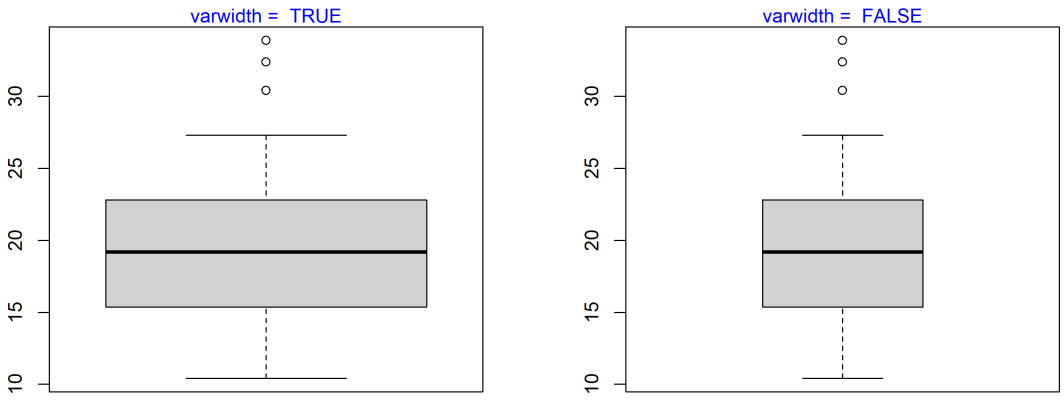


The below plot displays how the plot changes with the values set for `outline`:



Varwidth

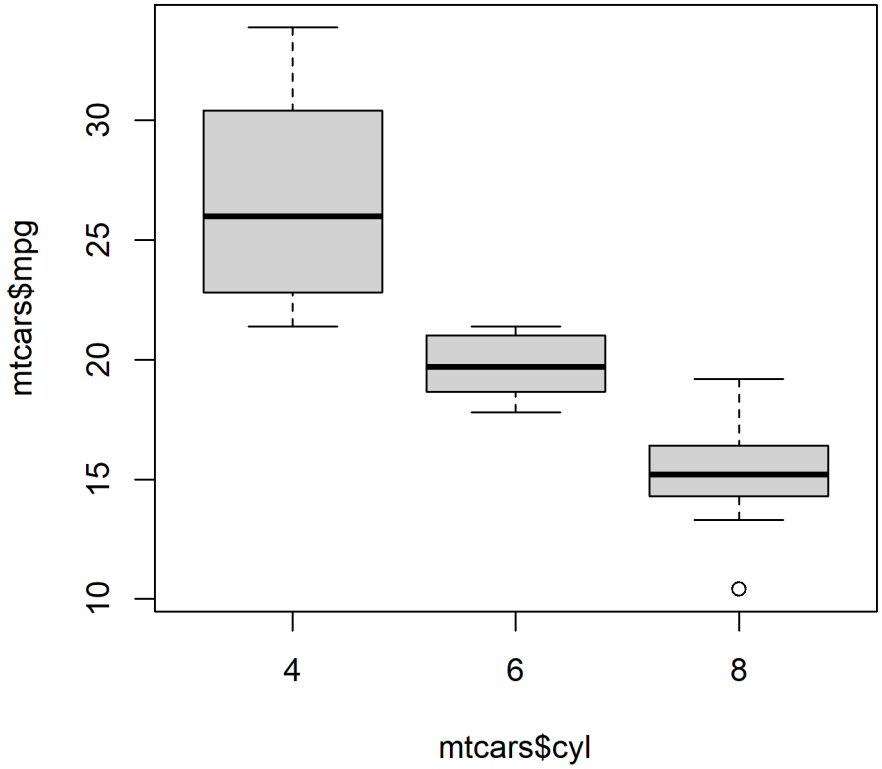
If the `varwidth` argument is set to `TRUE`, the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.



Bivariate/Multivariate Box Plot

As we said in the introduction, box plots can be used to compare distributions of several variables. Let us use the `mtcars` data set and compare the distribution of Miles Per Gallon (`mpg`) for automobiles with different number of cylinders (`cyl`). We will do this by specifying a formula as shown in the below example.

```
boxplot(mtcars$mpg ~ mtcars$cyl)
```

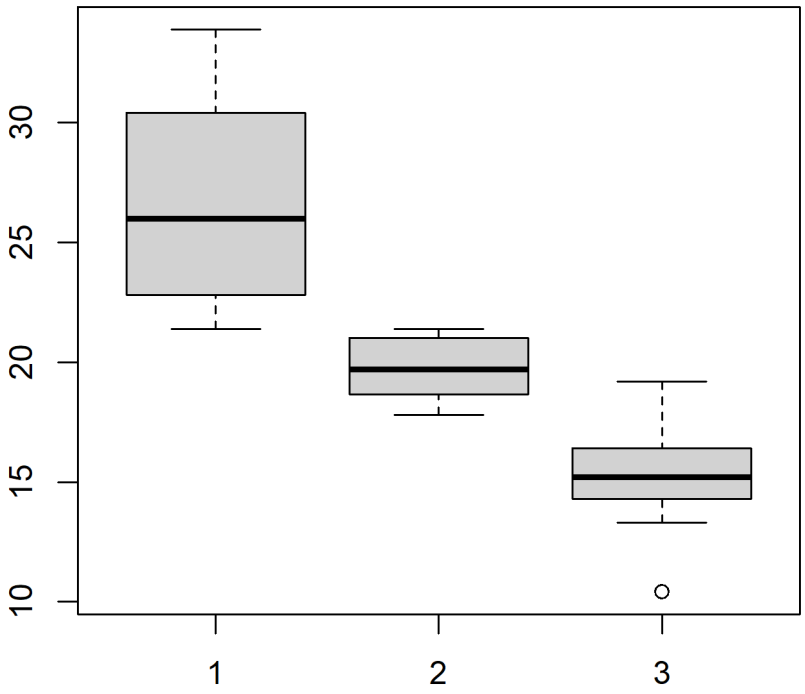


We use the formula when we are comparing the distribution of a continuous variable across different levels of a categorical variable. If we want to compare the distributions without using a categorical variable, we need to specify the variable separately in the `boxplot()` function. Below is an illustration of this method. We will split the `mpg` data using the `split()` function and plot them separately. The `split()` function splits a continuous variable based on the levels of a categorical variable.

```
mpg_split <- split(mtcars$mpg, mtcars$cyl)
```

```
mpg_split
## $`4`
##  [1] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26.0 30.4 21.4
##
## $`6`
## [1] 21.0 21.0 21.4 18.1 19.2 17.8 19.7
##
## $`8`
##  [1] 18.7 14.3 16.4 17.3 15.2 10.4 10.4 14.7 15.5 15.2 13.3 19.2 15.8 15.0
```

```r
mpg_4 <- mpg_split$`4`
mpg_6 <- mpg_split$`6`
mpg_8 <- mpg_split$`8`

boxplot(mpg_4, mpg_6, mpg_8)
```



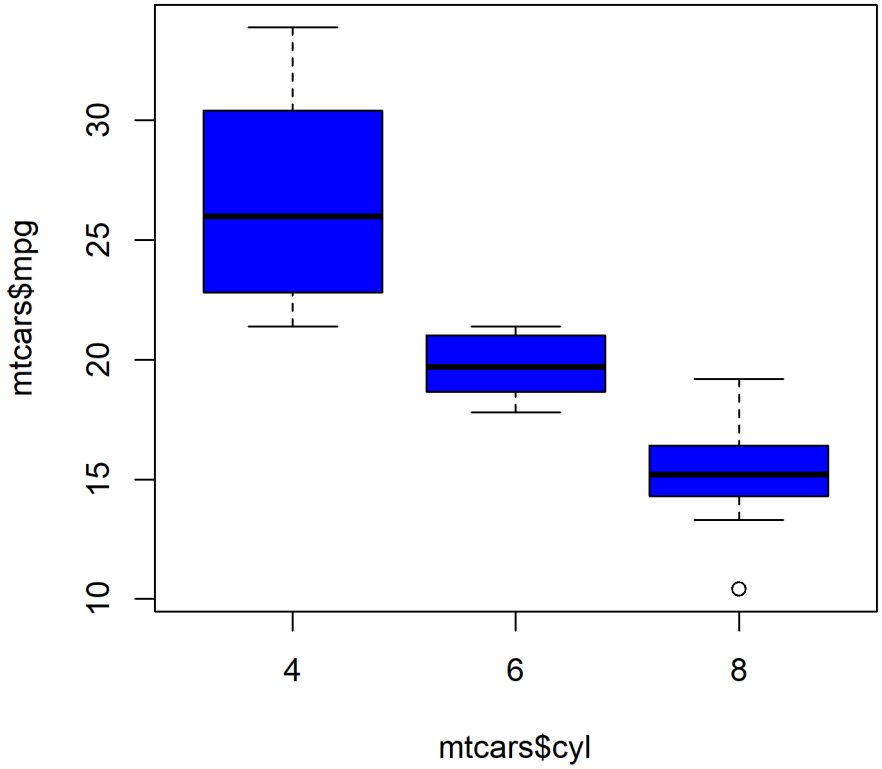
The same plot can be created in two ways. If you are comparing the distribution of a continuous variable for the different levels of a categorical variable, use the formula. If you are comparing distribution of independent variables, specify all the variablels in the `boxplot()` function.

Color

Let us add some color to the plot. We can specify as many colors as the boxes or we can specify the same color for all of them. Below are two examples where we specify the same color in the first one and different colors in the second one.

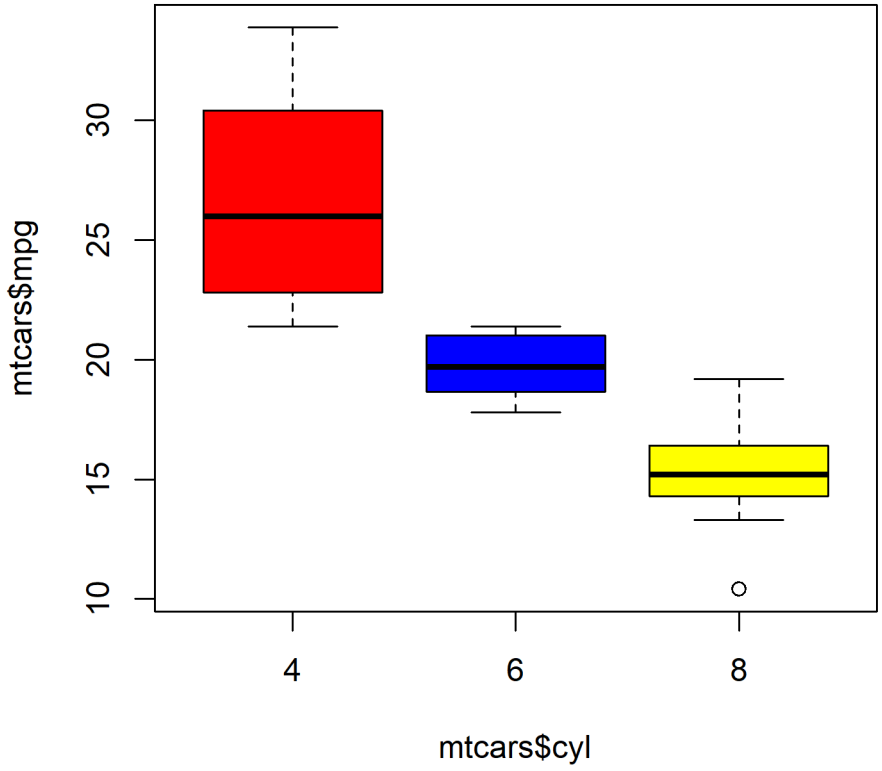
Single Color

```r
boxplot(mtcars$mpg ~ mtcars$cyl, col = 'blue')
```



Different Colors

```
boxplot(mtcars$mpg ~ mtcars$cyl,
        col = c('red', 'blue', 'yellow'))
```

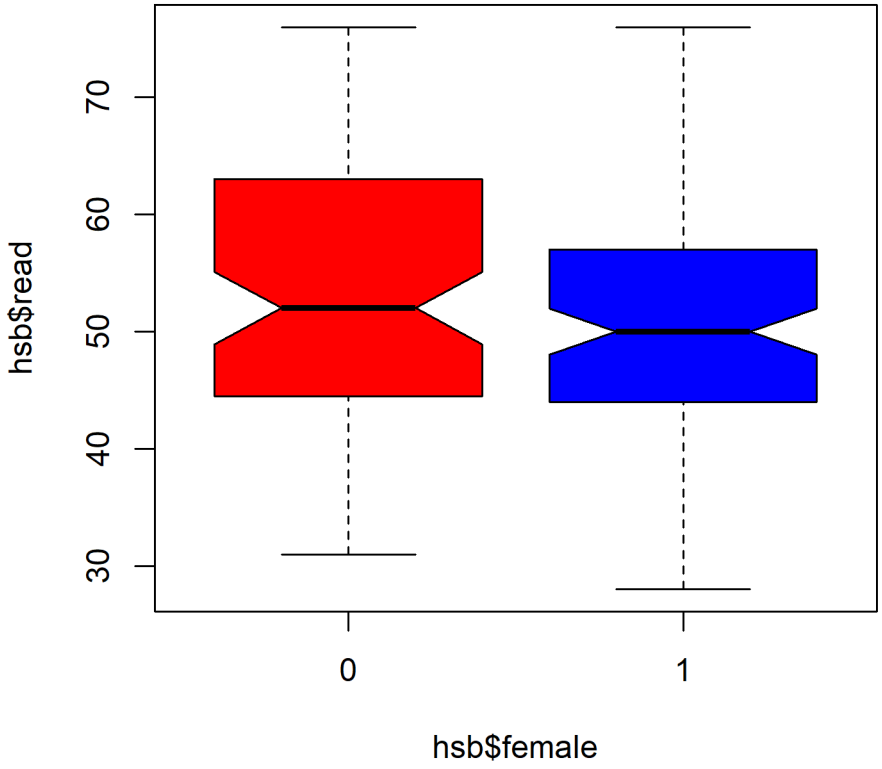


Compare Medians

If we want to test whether the medians of the different groups differ, we can use the `notch` argument and set it to `TRUE`. A notch is drawn in each side of the boxes and if the notches of the plots do not overlap, it is strong evidence that the medians differ.

We will use a different data set for this example. We use the `hsb2` data (High School and Beyond survey, 200 observations; vendored as `data/hsb2.csv`, originally from the UCLA Institute for Digital Research and Education) and compare the distribution of reading score (`read`) for males and females (`female`).

```
hsb <- read.csv('data/hsb2.csv')
boxplot(hsb$read ~ hsb$female, notch = TRUE,
        col = c('red', 'blue'))
```

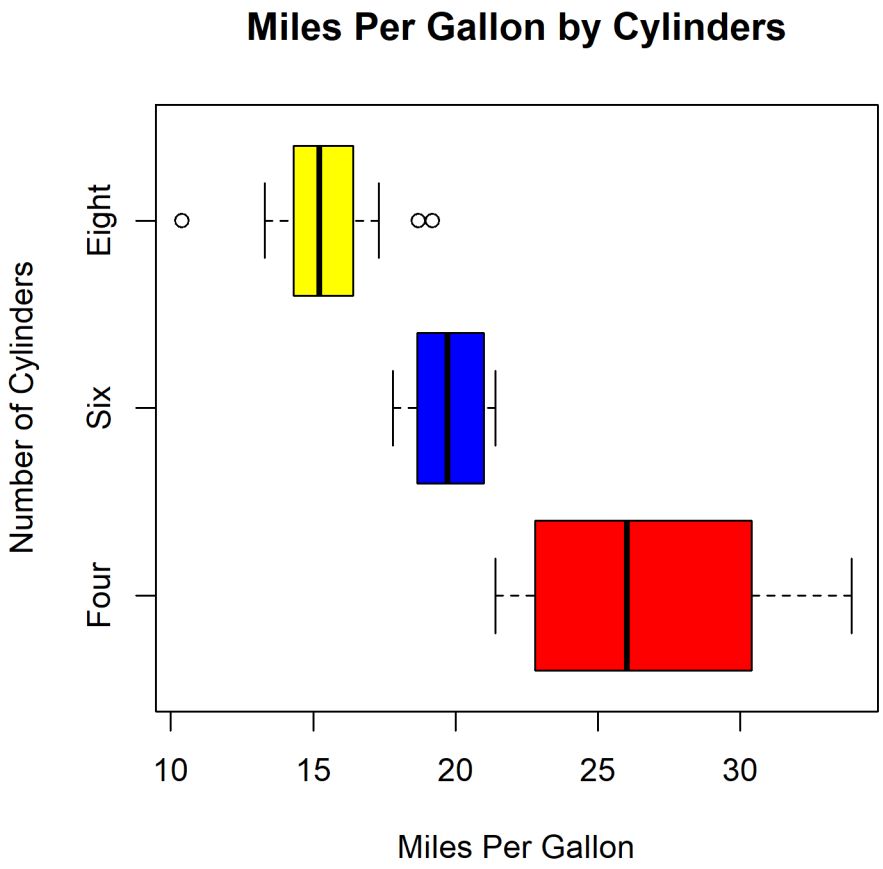


Since the notches overlap, there is no strong evidence that the medians differ.

Putting it all together

Let us conclude by adding a title and axis labels to the box plot.

```
boxplot(mtcars$mpg ~ mtcars$cyl, range = 1, outline = TRUE,
        horizontal = TRUE, col = c('red', 'blue', 'yellow'),
        main = 'Miles Per Gallon by Cylinders',
        ylab = 'Number of Cylinders', xlab = 'Miles Per Gallon',
        names = c('Four', 'Six', 'Eight'))
```



Histograms

Introduction

In this chapter, we will learn to:

- create a bare bones histogram
- specify the number of bins/intervals
- represent frequency density on the Y axis
- add colors to the bars and the border
- add labels to the bars

A histogram is a plot that can be used to examine the shape and spread of continuous data. It looks very similar to a bar graph and can be used to detect outliers and skewness in data. The histogram graphically shows the following:

- center (location) of the data
- spread (dispersion) of the data
- skewness

- outliers
- presence of multiple modes

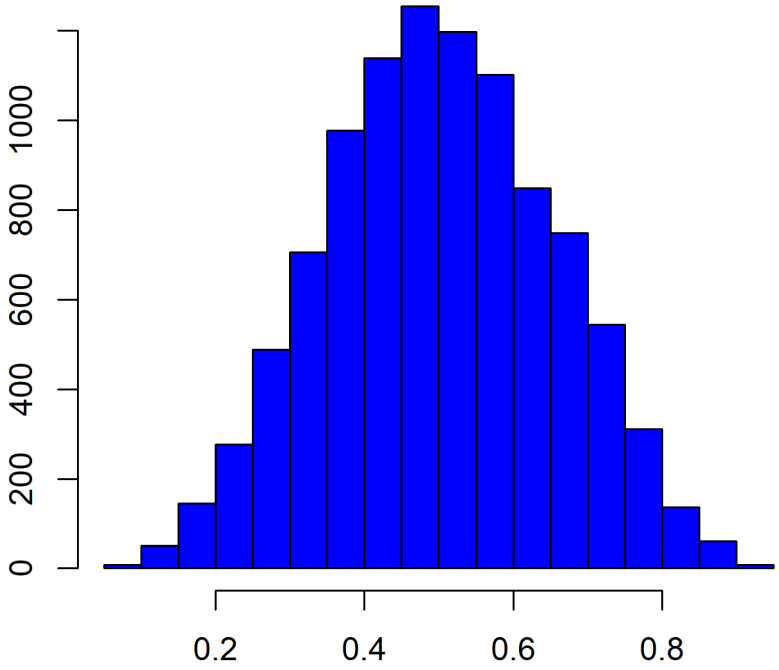
To construct a histogram, the data is split into intervals called bins. The intervals may or may not be equal sized. For each bin, the number of data points that fall into it are counted (frequency). The Y axis of the histogram represents the frequency and the X axis represents the variable.

Distributions

Before we learn how to create histograms, let us see how normal and skewed distributions look when represented by a histogram.

Normal Distribution

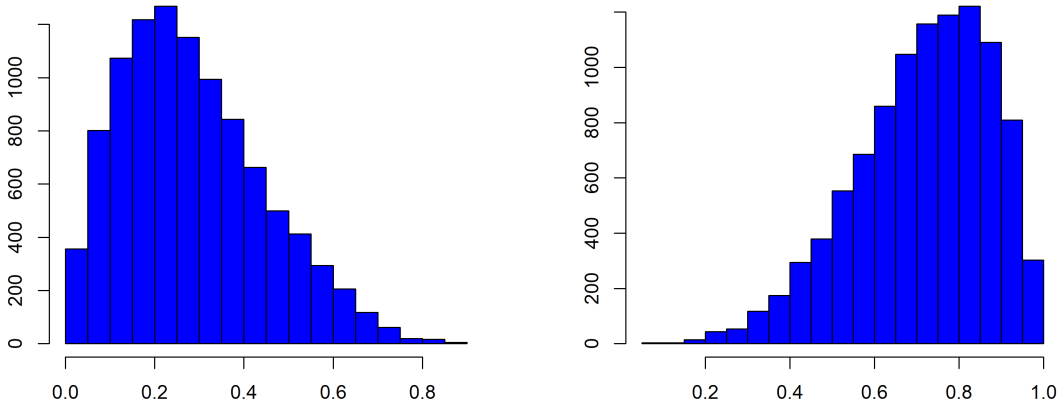
```
set.seed(1)
hist(rbeta(10000, 5, 5), ann = FALSE, col = 'blue')
```



Skewed Distributions

```
set.seed(1)
init <- par(no.readonly = TRUE)
par(mfrow = c(1, 2))
```

```r
hist(rbeta(10000, 2, 5), ann = FALSE, col = 'blue')
hist(rbeta(10000, 5, 2), ann = FALSE, col = 'blue')
```

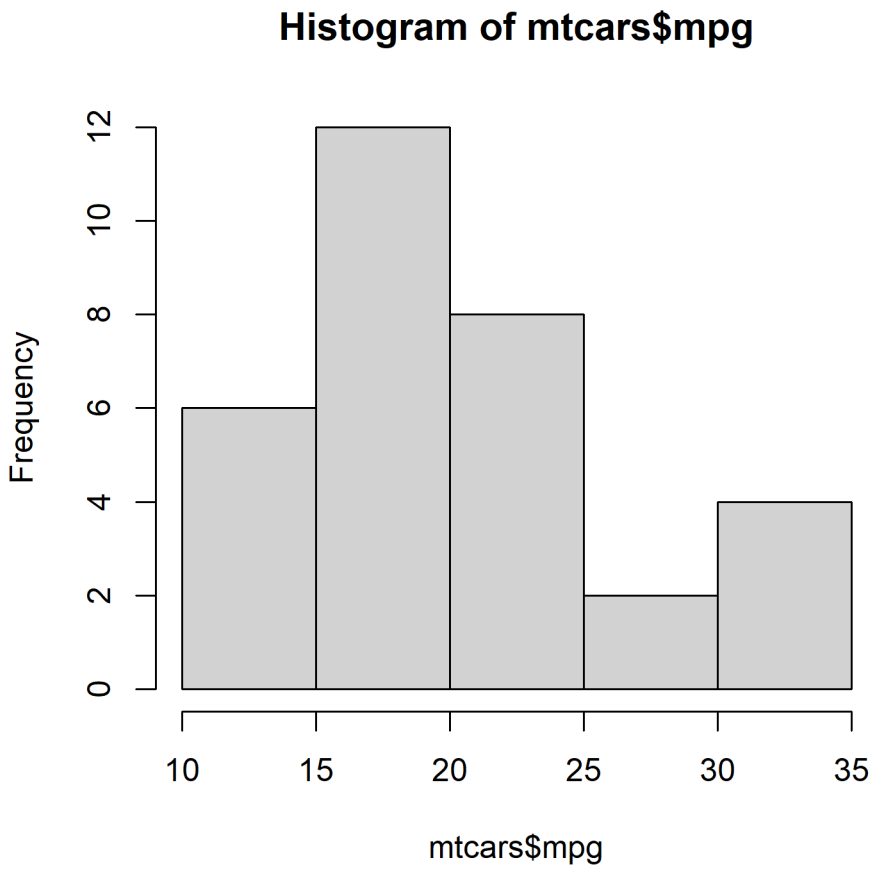



```r
par(init)
```

Basics

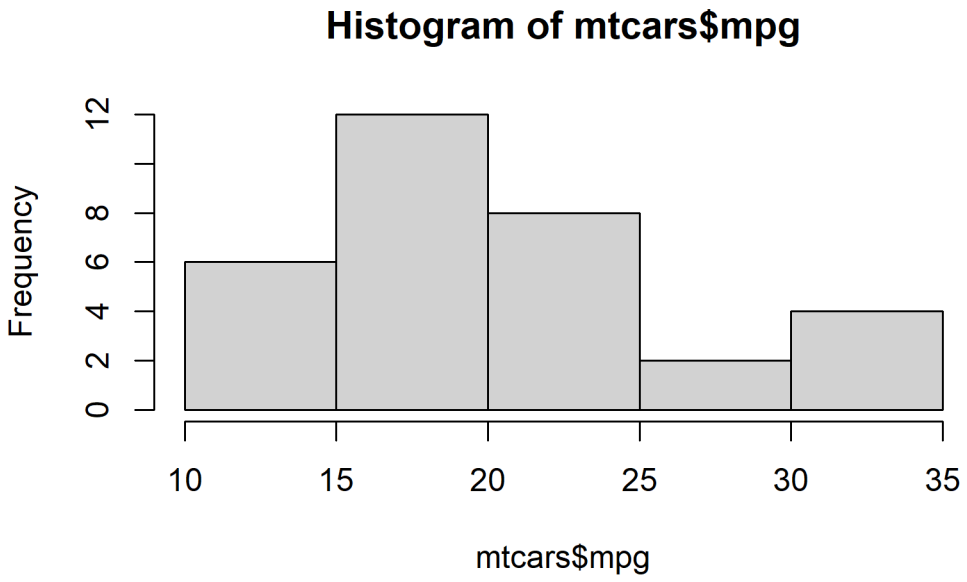
Histograms are created using the `hist()` function in R. The minimum input required to create a bare bones histogram is a continuous variable. Below is an example:

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



The `hist()` functions returns details of the histogram which can be accessed by assigning the histogram to a variable. Let us assign the above histogram to a variable `h` and use the `$` symbol to access the details stored in the variable.

```
# store the results of hist function
h <- hist(mtcars$mpg)
```

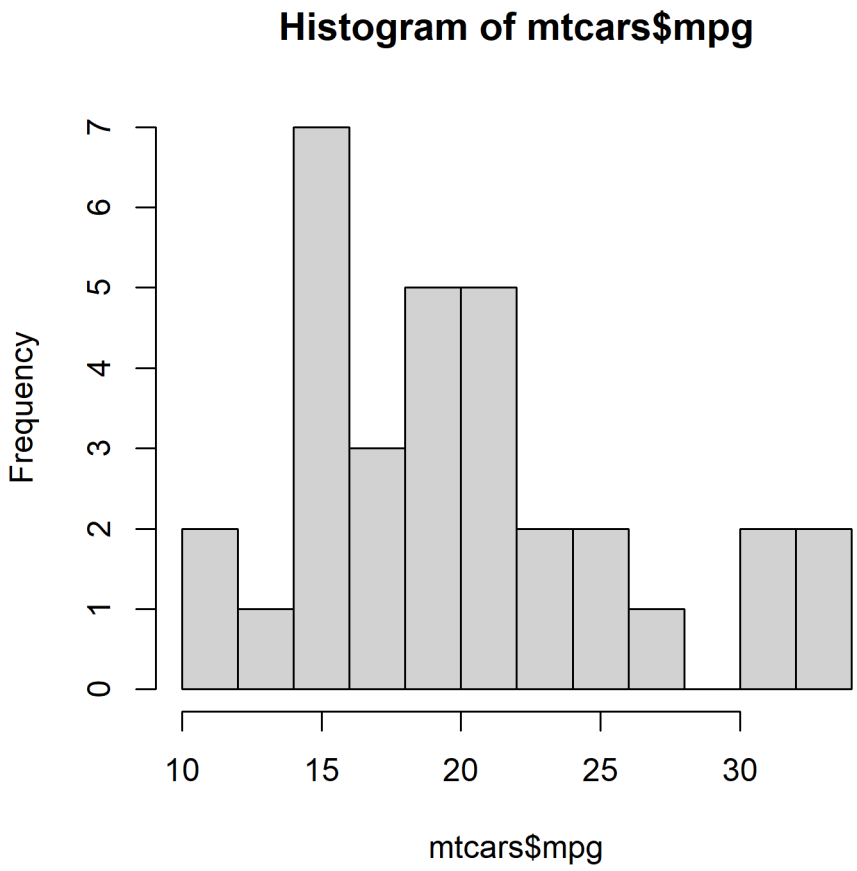



```
# display number of breaks
h$breaks
## [1] 10 15 20 25 30 35

# frequency of the intervals
h$counts
## [1]  6 12  8  2  4

# frequency density
h$density
## [1] 0.0375 0.0750 0.0500 0.0125 0.0250

# mid points of the intervals
h$mids
## [1] 12.5 17.5 22.5 27.5 32.5

# varible name
h$xname
## [1] "mtcars$mpg"

# whether intervals are of equal size
h$equidist
## [1] TRUE
```

Bins

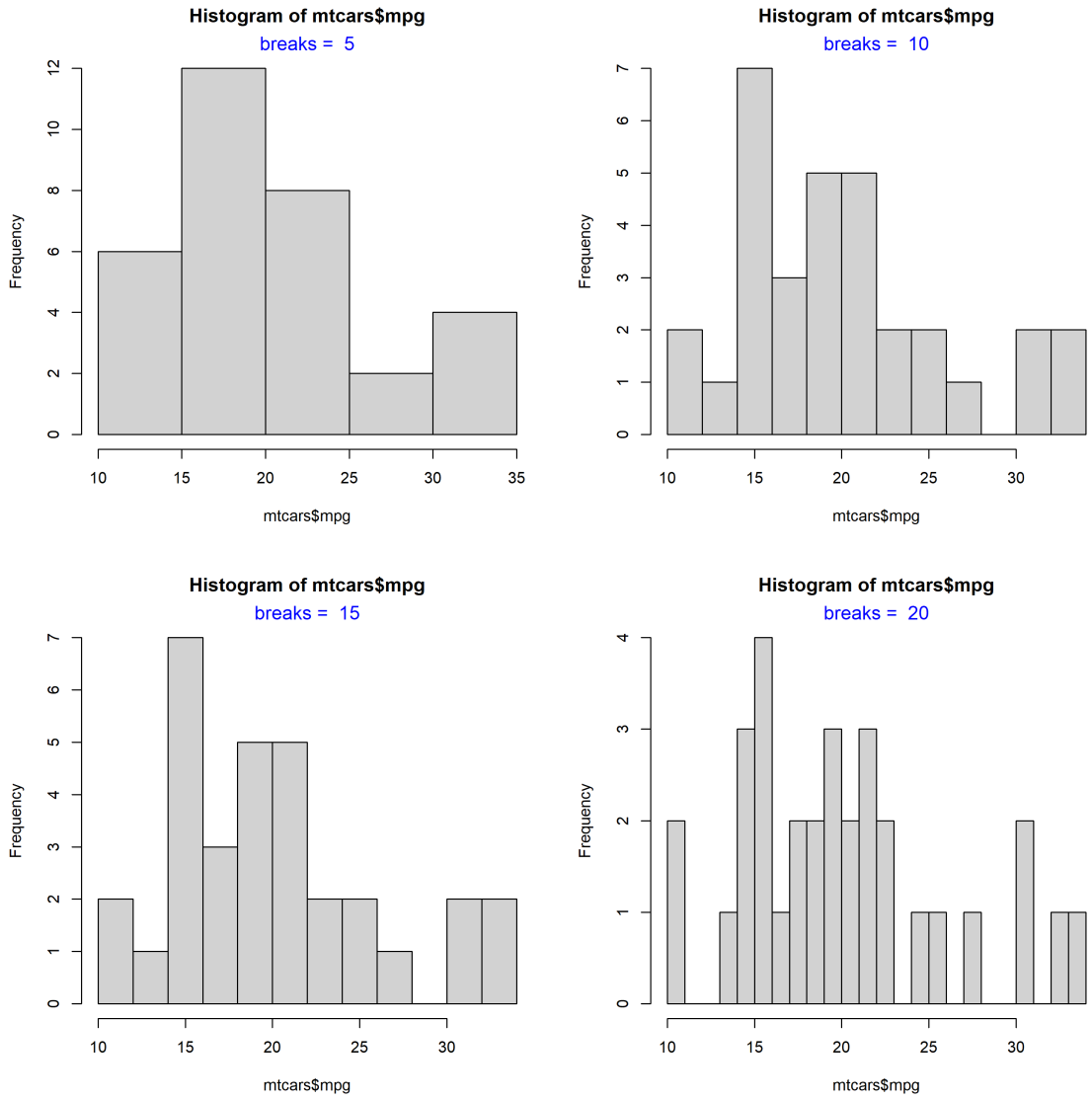
The `hist()` function creates equidistant intervals by default. We can specify the number of bins using the `breaks` argument.

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



The below plot displays histograms with different number of bins:

```r
init <- par(no.readonly = TRUE)
par(mfrow = c(2, 2))
values <- c(5, 10, 15, 20)
for (i in values) {
    hist(mtcars$mpg, breaks = i)
    mtext(paste("breaks = ", i), side = 3, col = "blue")
}
```

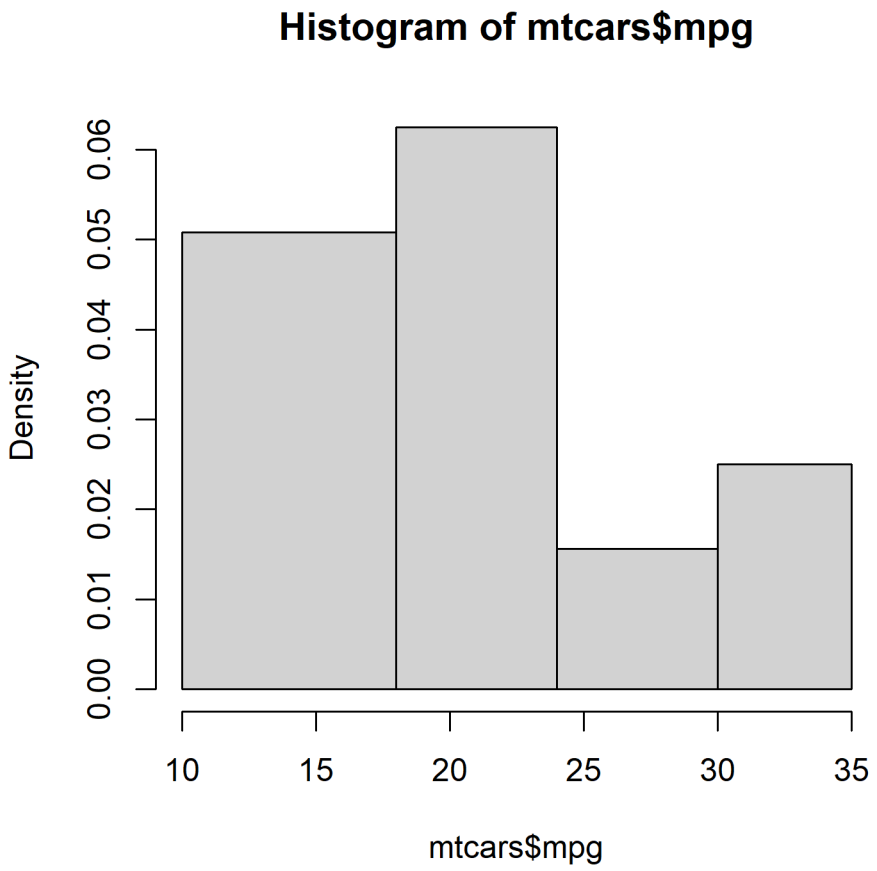



```
par(init)
```

Intervals

If we want to create histograms with specific intervals, the `breaks` argument can be supplied with the intervals.

```
hist(mtcars$mpg, breaks = c(10, 18, 24, 30, 35))
```



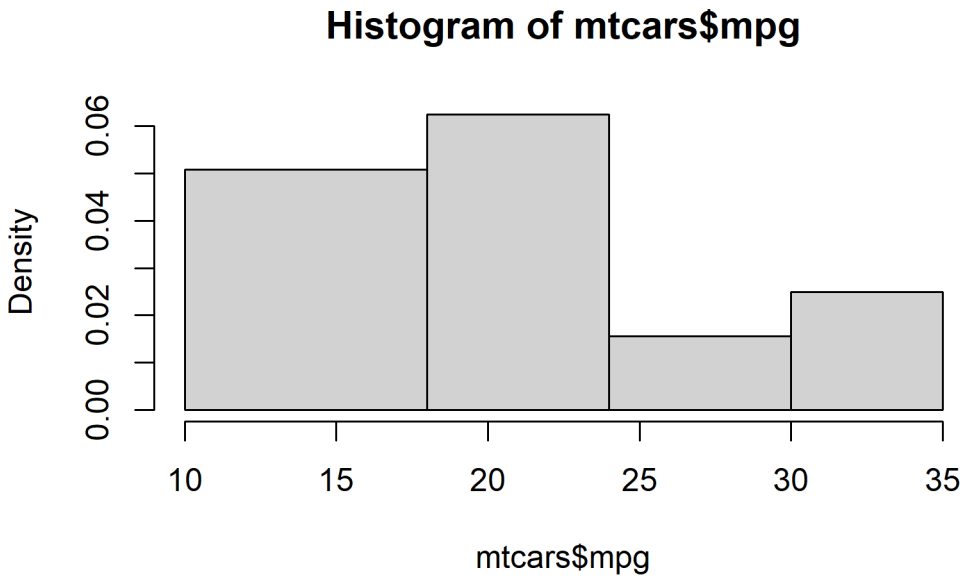
If you observe the Y axis, it does not represent frequency any more. Instead, it represents the frequency density. What is frequency density?

Frequency Density

Frequency Density = Relative Frequency / Class Width

Relative Frequency = Frequency / Total Observations

```
h <- hist(mtcars$mpg, breaks = c(10, 18, 24, 30, 35))
```




```
frequency <- h$counts
class_width <- c(8, 6, 6, 5)
rel_freq <- frequency / length(mtcars$mpg)
freq_density <- rel_freq / class_width
d <- data.frame(frequency = frequency, class_width = class_width, relative_frequency
= rel_freq, frequency_density = freq_density)
d
```

```
  frequency class_width relative_frequency frequency_density
1        13           8            0.40625        0.05078125
2        12           6            0.37500        0.06250000
3         3           6            0.09375        0.01562500
4         4           5            0.12500        0.02500000
```

When multiplied by the class width, the product will always sum upto 1.

```
sum(d$frequency_density * d$class_width)
```

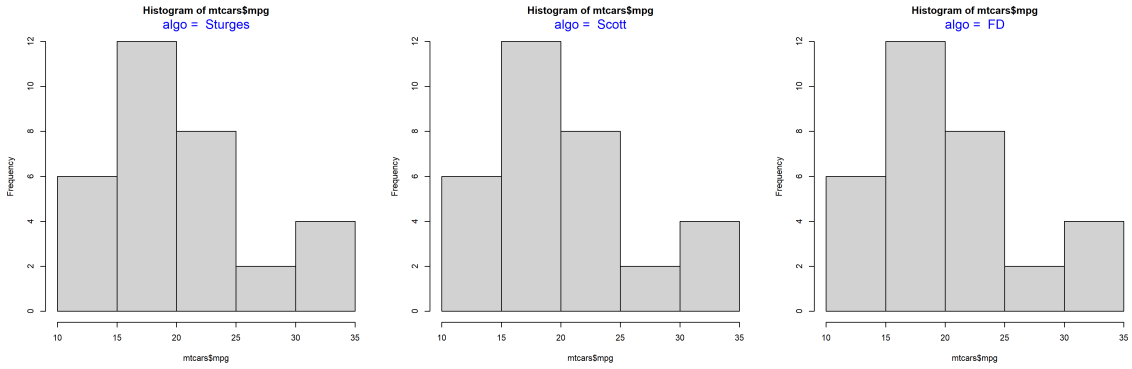
```
[1] 1
```

We will learn more about frequency density in a bit. Before we end this section, we need to learn about one more way to specify the intervals of the histogram, algorithms. The `hist()` function allows us to specify the following algorithms:

- Sturges (default)
- Scott
- Freedman-Diaconis (FD)

In the below plot, we examine how th algorithms work:

```
init <- par(no.readonly = TRUE)
par(mfrow = c(1, 3))
values <- c("Sturges", "Scott", "FD")
for (i in values) {
    hist(mtcars$mpg, breaks = i)
    mtext(paste("algo = ", i), side = 3, col = "blue")
}
```

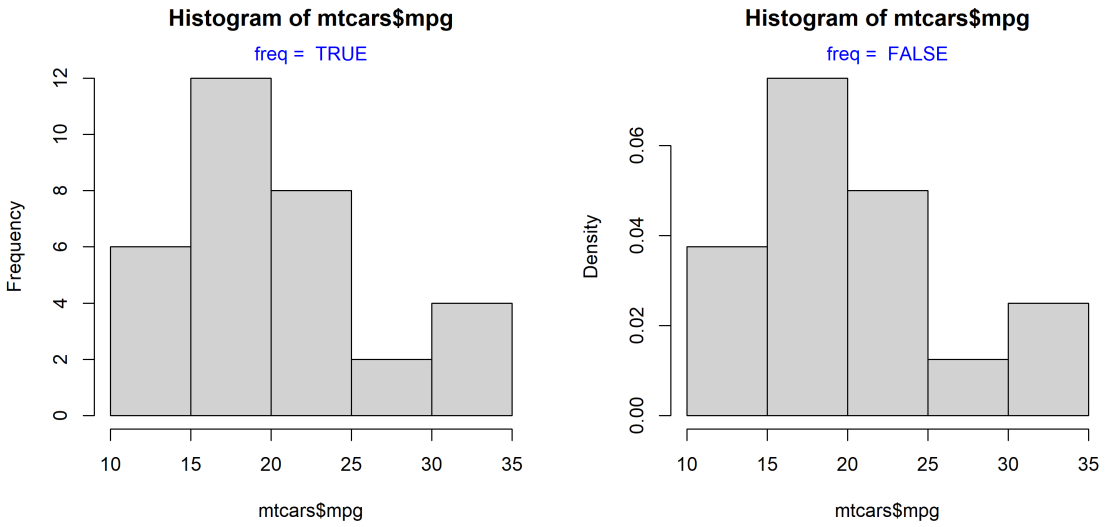



```
par(init)
```

Frequency Distribution II

Let us come back to frequency density. If you want the Y axis of the histogram to represent frequency density instead of counts, set the `freq` argument to `FALSE`.

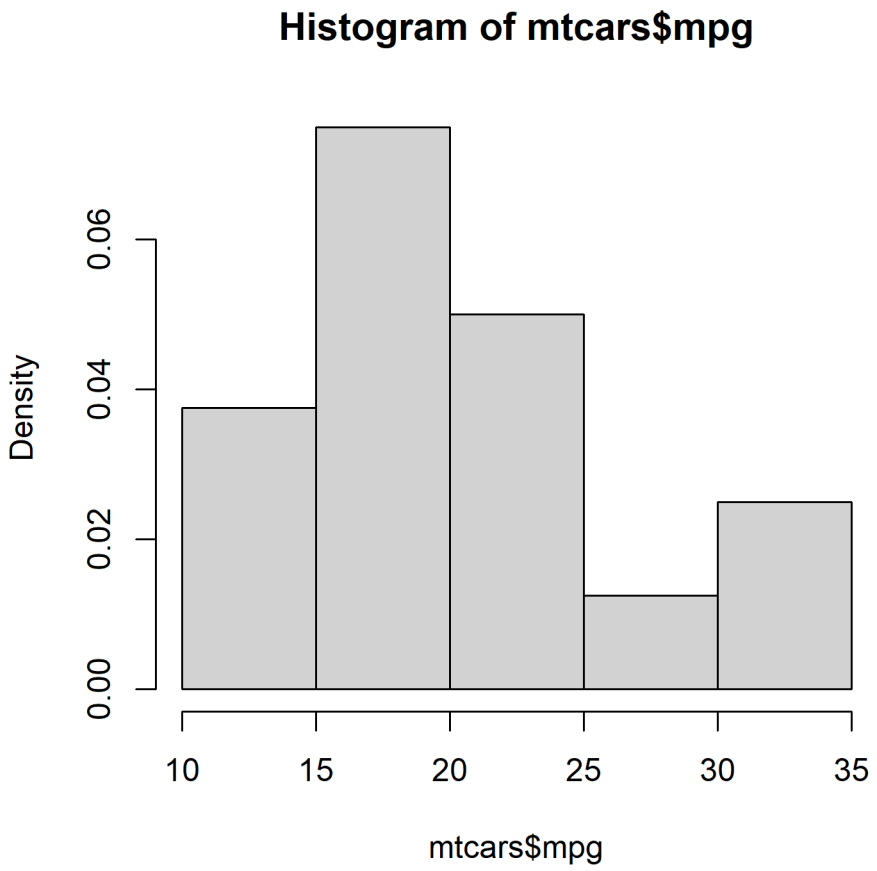
```
init <- par(no.readonly = TRUE)
par(mfrow = c(1, 2))
values <- c(TRUE, FALSE)
for (i in values) {
    hist(mtcars$mpg, freq = i)
    mtext(paste("freq = ", i), side = 3, col = "blue")
}
```




```
par(init)
```

The same result can be achieved by using the `probability` argument as well. It takes only logical values as inputs and the default is `FALSE`. If set to `TRUE`, the Y axis will represent the frequency density instead of counts.

```
hist(mtcars$mpg, probability = TRUE)
```

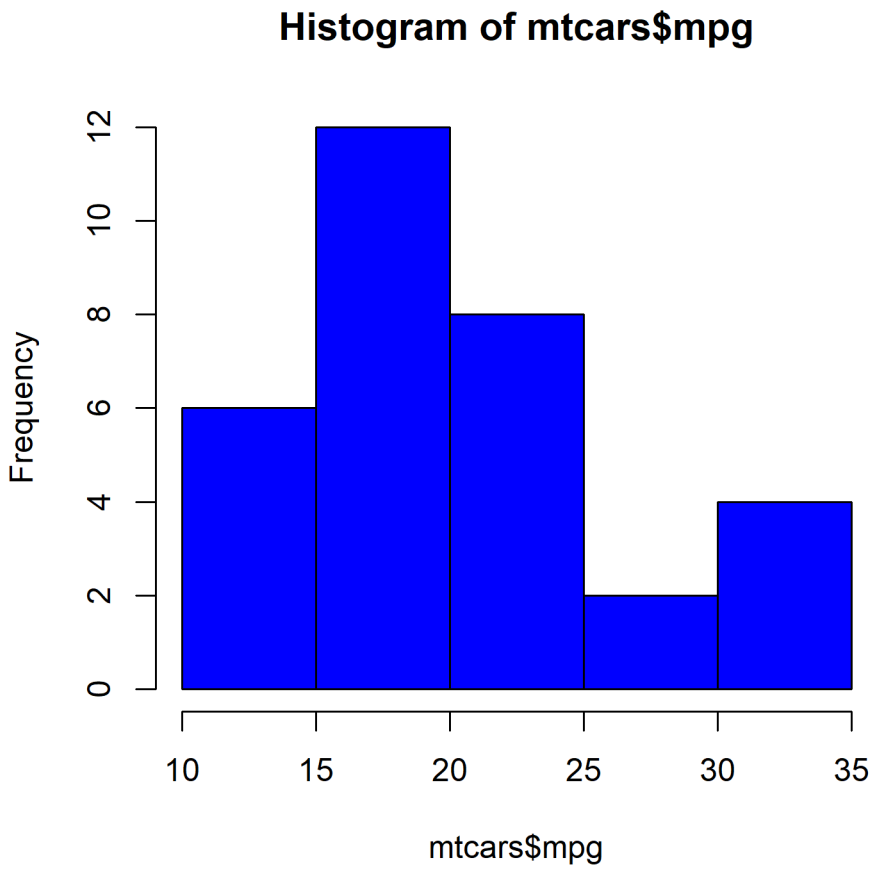


Color

To add colors to the bars of the histogram, use the `col` argument. If the number of colors specified is less than the number of bars, the colors are recycled. Below are a few examples:

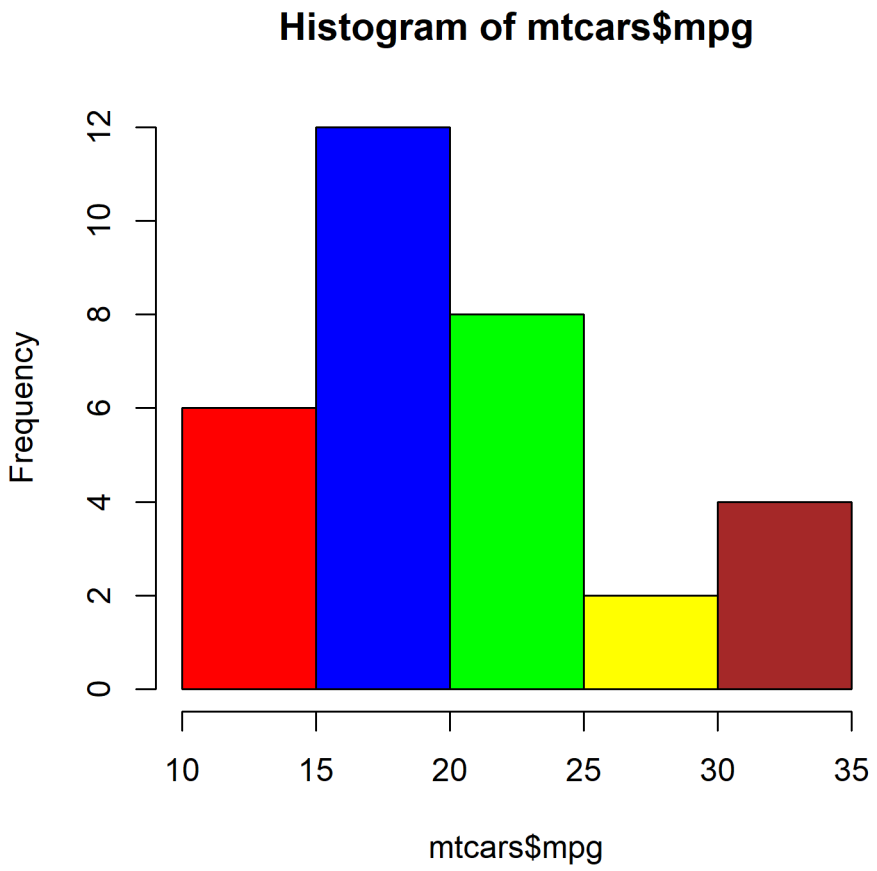
Single Color

```r
hist(mtcars$mpg, col = 'blue')
```



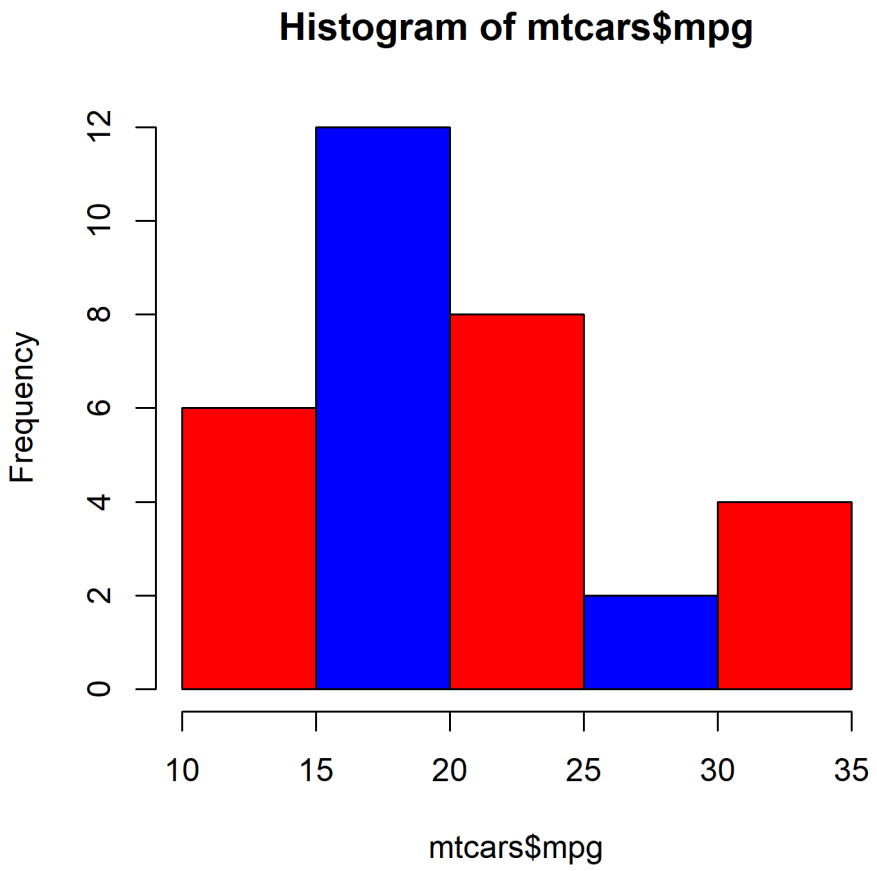
Different Colors

```
hist(mtcars$mpg, col = c('red', 'blue', 'green', 'yellow', 'brown'))
```



Recycled Colors

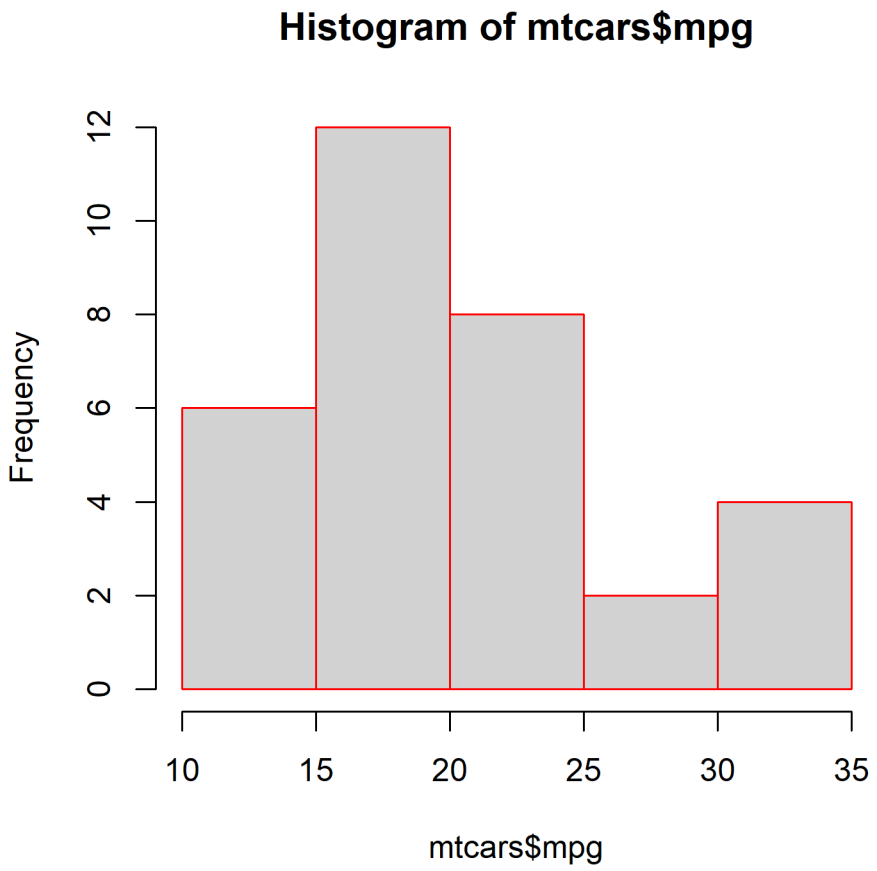
```
hist(mtcars$mpg, col = c('red', 'blue'))
```



Border Color

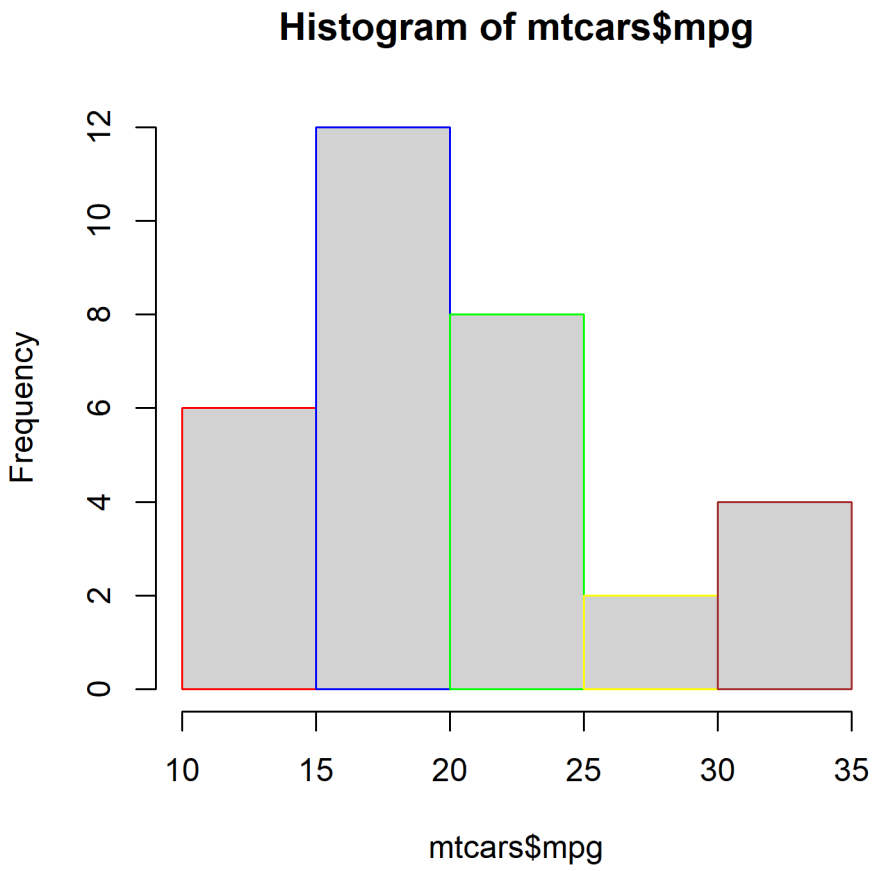
Colors can be specified for the borders of the histogrambars using the `border` argument.

```
hist(mtcars$mpg, border = 'red')
```



Different Colors

```
hist(mtcars$mpg, border = c('red', 'blue', 'green', 'yellow', 'brown'))
```



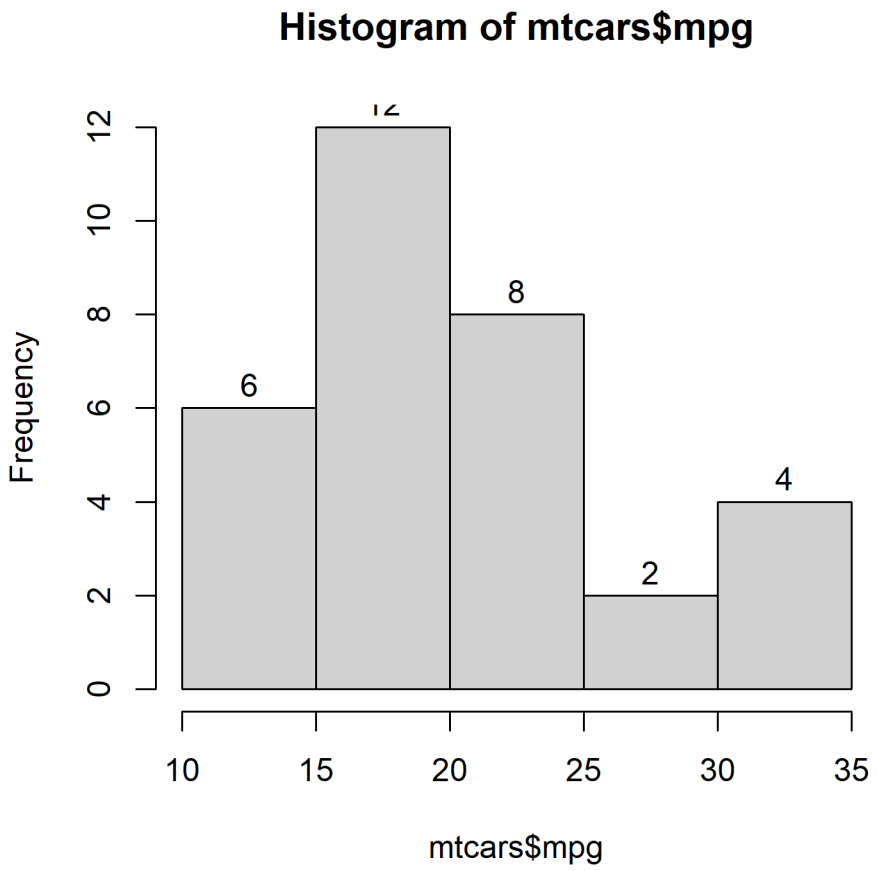
Labels

In certain cases, we might want to add the frequency counts on the histogram bars. It is easier for the user to know the frequencies of each bin when they are present on top of the bars. Let us add the frequency counts on top of the bars using the `labels` argument. We can either set it to `TRUE` or a character vector containing the label values. Let us look at both the methods.

Method 1

Set `labels` to `TRUE`.

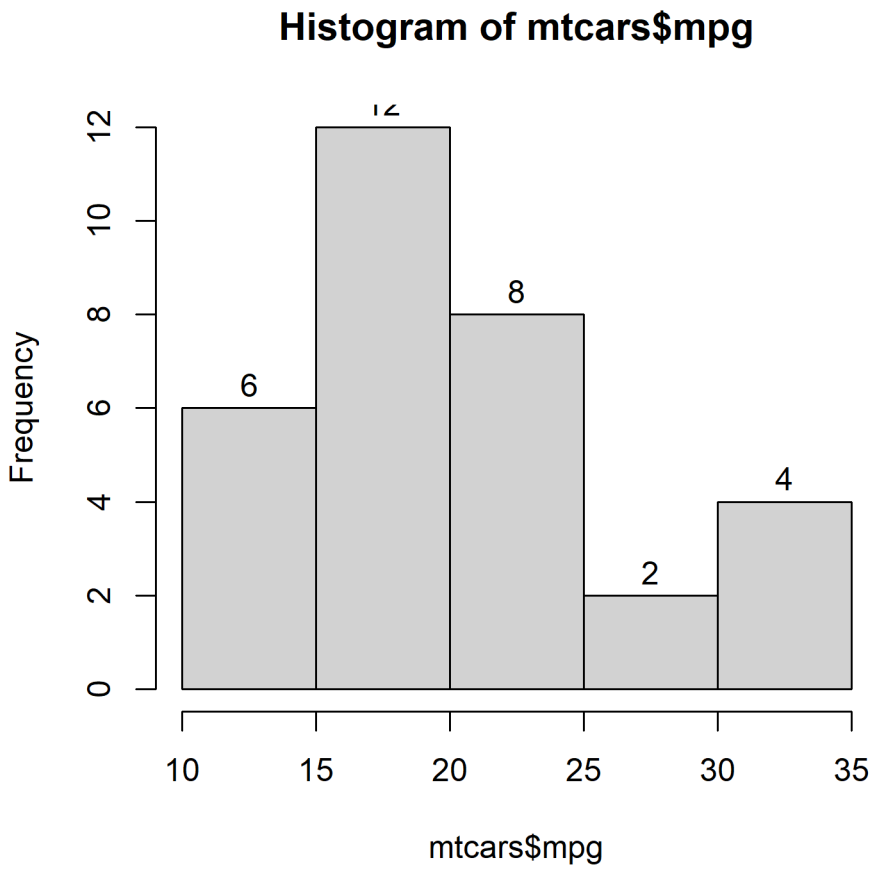
```
hist(mtcars$mpg, labels = TRUE)
```



Method 2

Specify the label values in a character vector.

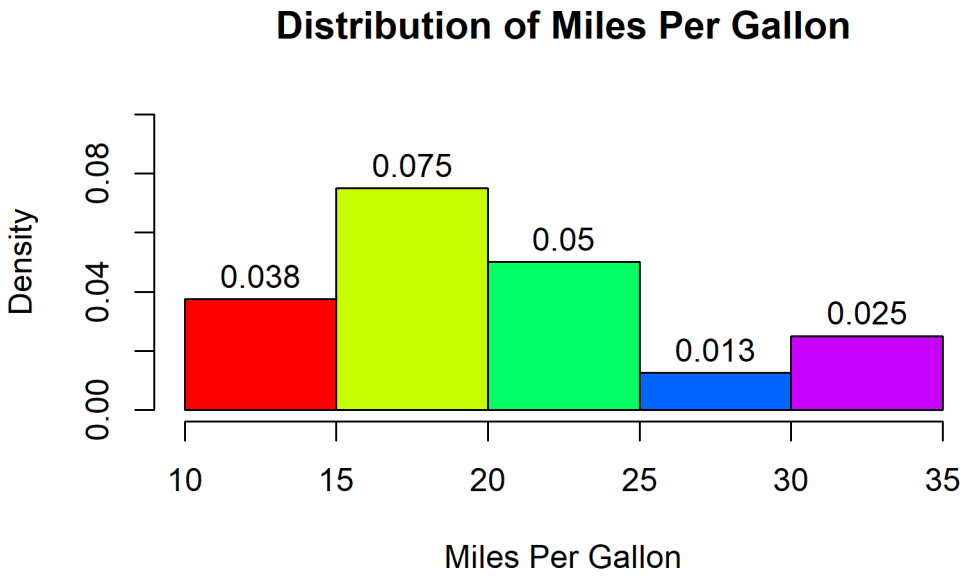
```
hist(mtcars$mpg, labels = c("6", "12", "8", "2", "4"))
```



Putting it all together..

Let us add a title and axis labels to the histogram.

```
hist(mtcars$mpg, labels = TRUE, prob = TRUE,
     ylim = c(0, 0.1), xlab = 'Miles Per Gallon',
     main = 'Distribution of Miles Per Gallon',
     col = rainbow(5))
```



Legends

Introduction

In this chapter, we will learn how to:

- position the legend within the plot
- modify the layout using `ncol` and `horiz` arguments
- add title using the `title` set of arguments
- modify the appearance and position of the legend box
- modify the appearance of the text in the legend box

Legends are used to convey information about the data being represented by a plot. To understand the importance of legends, let us look at the two plots below. In the first plot, would you be able to understand what the lines represent in the absence of a legend? May be yes but only if the author provides information in a textual form outside the plot. While such information will be useful, it will also be very inconvenient. Now look at the second plot, from the legend at the top right we can easily interpret what the lines represent. Would you agree that a legend is integral to plot representing multiple data? If yes, let us go ahead and learn how to add a legend to different plots.

Since we have looked at a line graph in the above example, we will learn how to add a legend to a line graph. After that, we will generalise the steps to different plots.

Data

Let us build a line graph that represents annual economic growth (GDP) data of the BRICS nations for the years 2010-14.

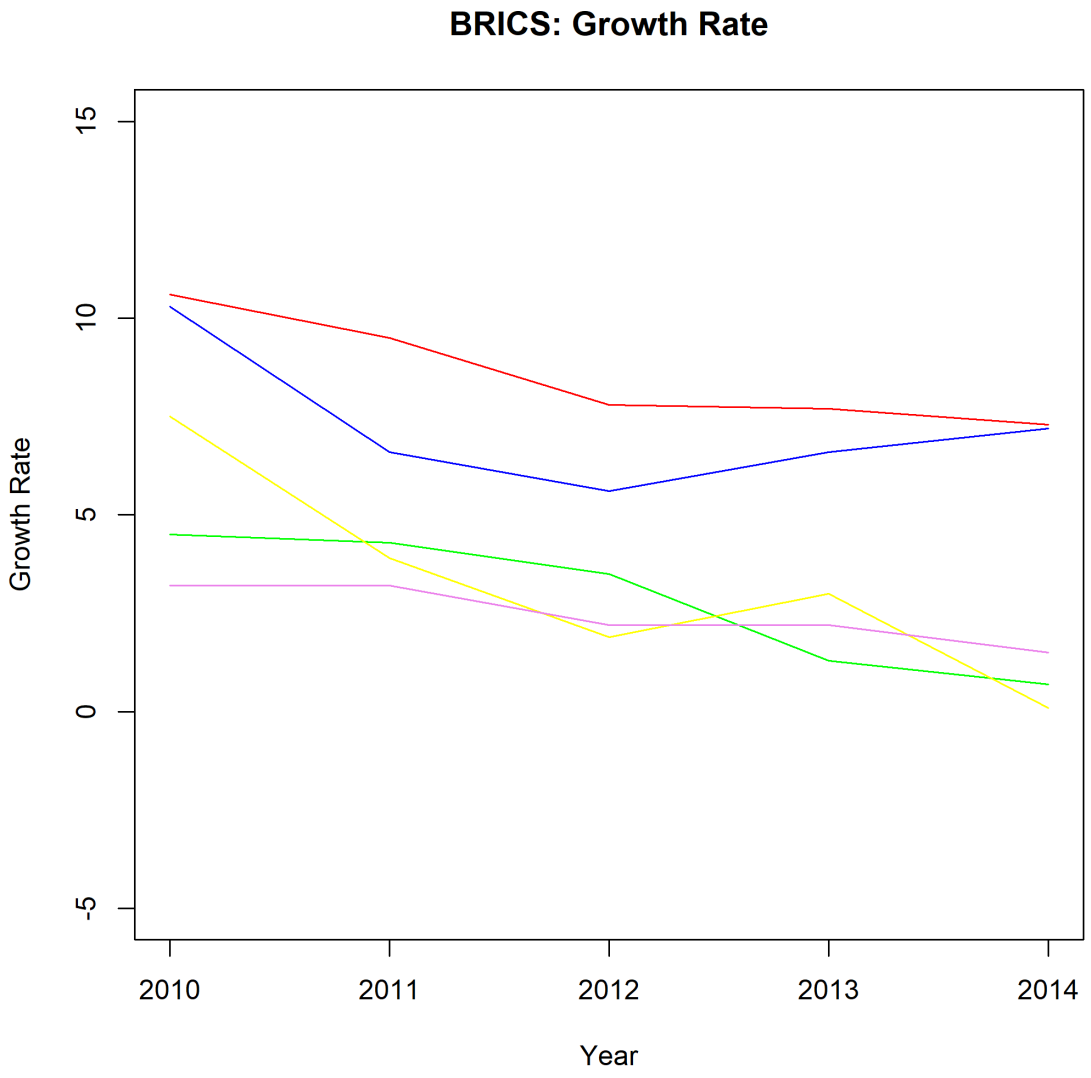
> The values below are a frozen teaching vintage, also shipped as `data/brics-gdp-2010-14.csv`. They differ slightly from the World Bank's current revised series, so treat them as illustrative, not citable statistics.

```
year <- seq(2010, 2014, 1)
india <- c(10.3, 6.6, 5.6, 6.6, 7.2)
china <- c(10.6, 9.5, 7.8, 7.7, 7.3)
russia <- c(4.5, 4.3, 3.5, 1.3, 0.7)
brazil <- c(7.5, 3.9, 1.9, 3.0, 0.1)
s_africa <- c(3.2, 3.2, 2.2, 2.2, 1.5)
gdp <- data.frame(year, india, china, russia, brazil, s_africa, stringsAsFactors =
FALSE)
gdp
```

```
  year india china russia brazil s_africa
1 2010  10.3  10.6    4.5    7.5      3.2
2 2011   6.6   9.5    4.3    3.9      3.2
3 2012   5.6   7.8    3.5    1.9      2.2
4 2013   6.6   7.7    1.3    3.0      2.2
5 2014   7.2   7.3    0.7    0.1      1.5
```

Line Graph

Below is the line graph that represents the above GDP data set:



Without a legend, it will be very difficult to map the lines to the BRICS nations. We will add a legend to the above plot using the `legend()` function and do so one step at a time.

Legend Location

In order to add a legend to the plot, the first thing we must specify is the location of the legend in the plot. There are 2 ways to do this:

- use x and y axis coordinates
- use keywords

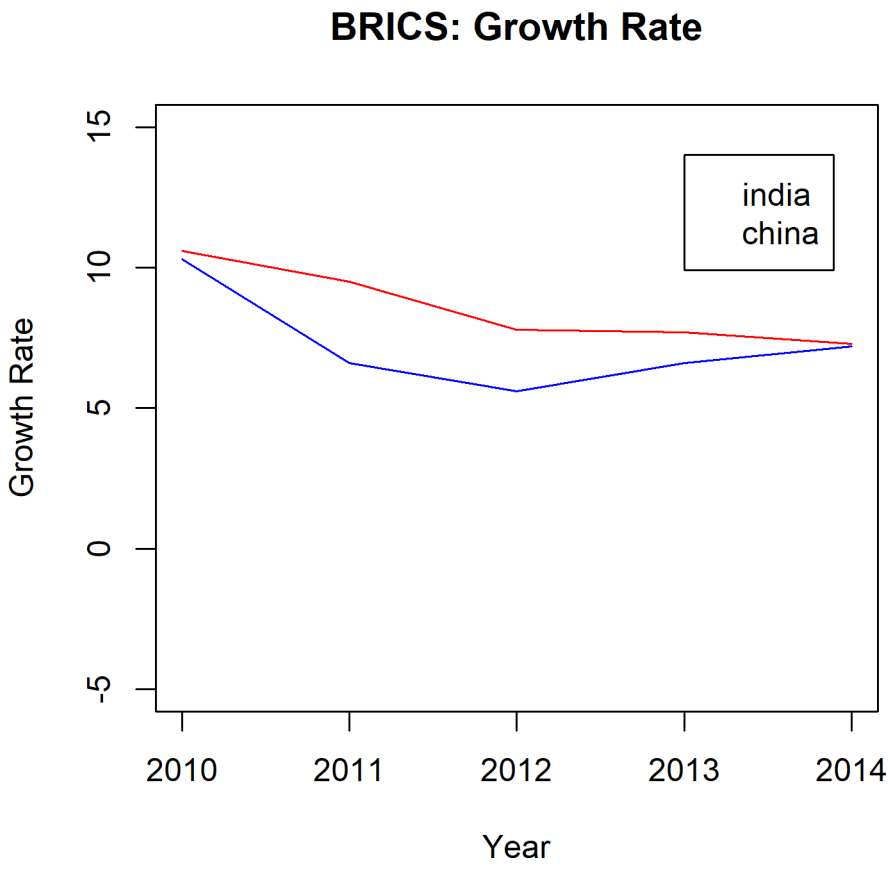
The list of keywords include:

- top
- bottom

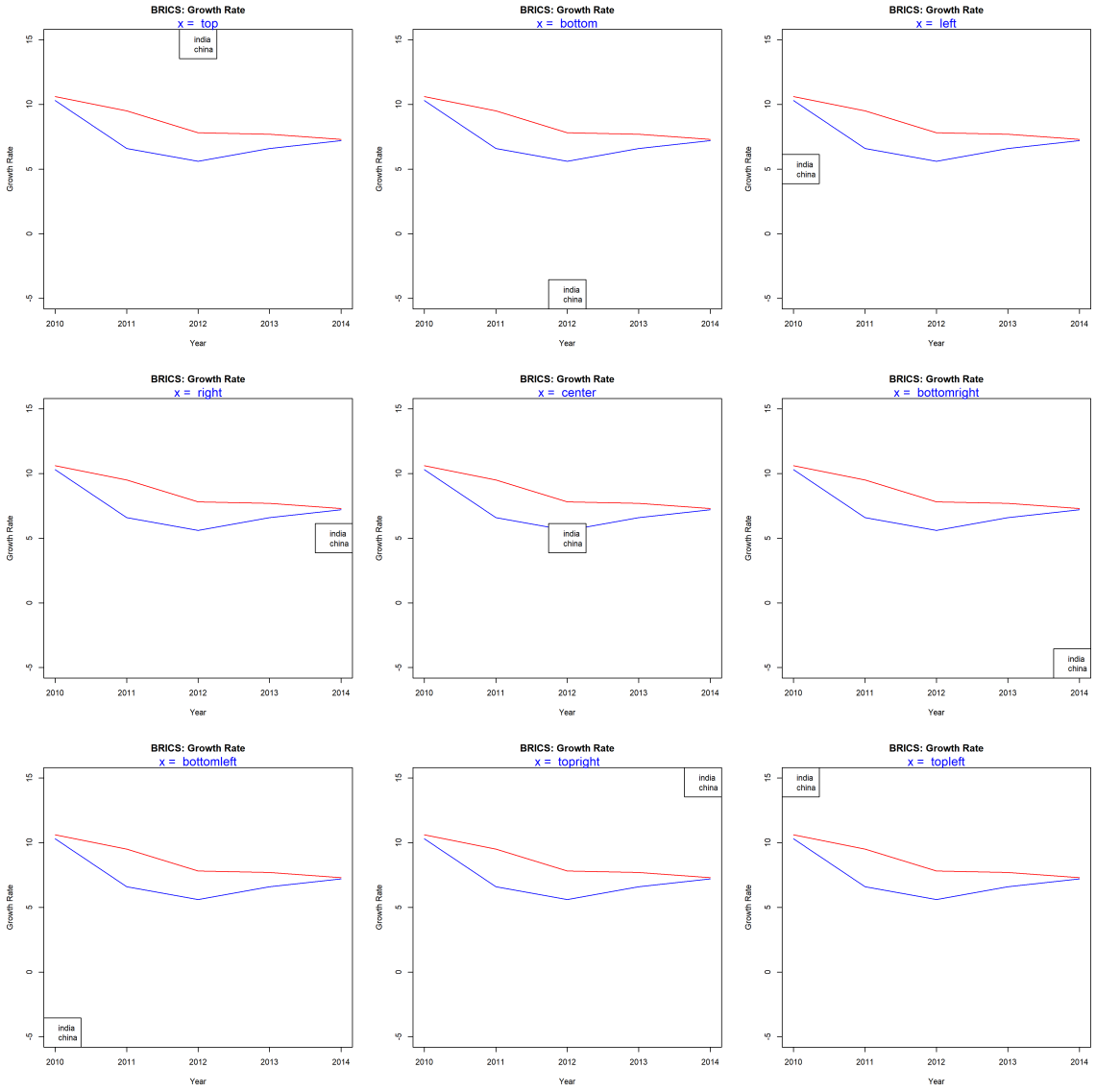
- left
- right
- center
- bottomright
- bottomleft
- topright
- topleft

But there is a glitch. If we do not specify what goes into the legend, the `legend()` function will return an error. Before we experiment with the location of the legend inside the plot, we need to learn about another argument used to specify the content of the legend. The argument is also named `legend`. It takes a vector as input. In the next example, we will plot the GDP data for India and China and add a basic legend.

```
{plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
legend(x = 2013, y = 14, legend = c('india', 'china'))}
```



You can see that a legend has been added bases on the X and Y axis coordinates we specified in the `legend()` function. But the legend is incomplete and a user still cannot map the lines to the countries using the legend. We will learn how to add lines inside the legend shortly but before that let us use keywords to position the legend inside the plot.



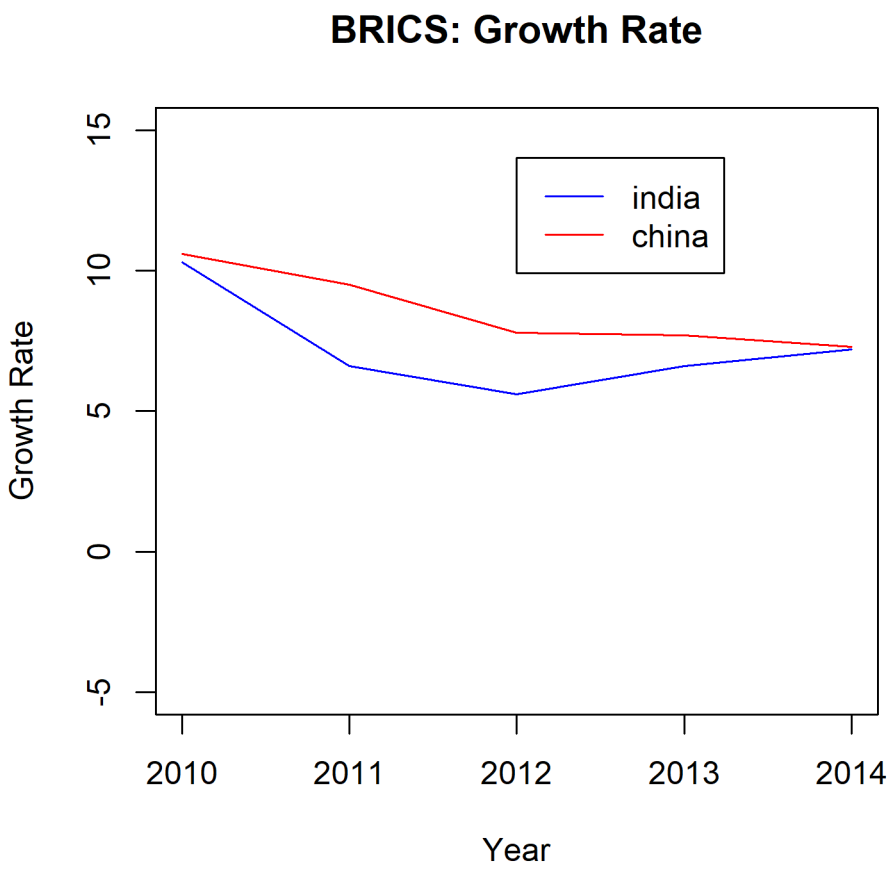
You can either use the keywords or the axis coordinates to position the legend inside the plot. Use the coordinates method if you want greater control over the position of the legend. Next step is to add lines inside the legend so that a user can map the lines in the plots to the countries.

Lines

Adding a line in the legend is very simple. Use the `lty` argument to specify the line type and the `col` argument to add color to the line.

```
{plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
```

```
legend(x = 2012, y = 14, legend = c('india', 'china'),
       lty = 1, col = c('blue', 'red'))}
```



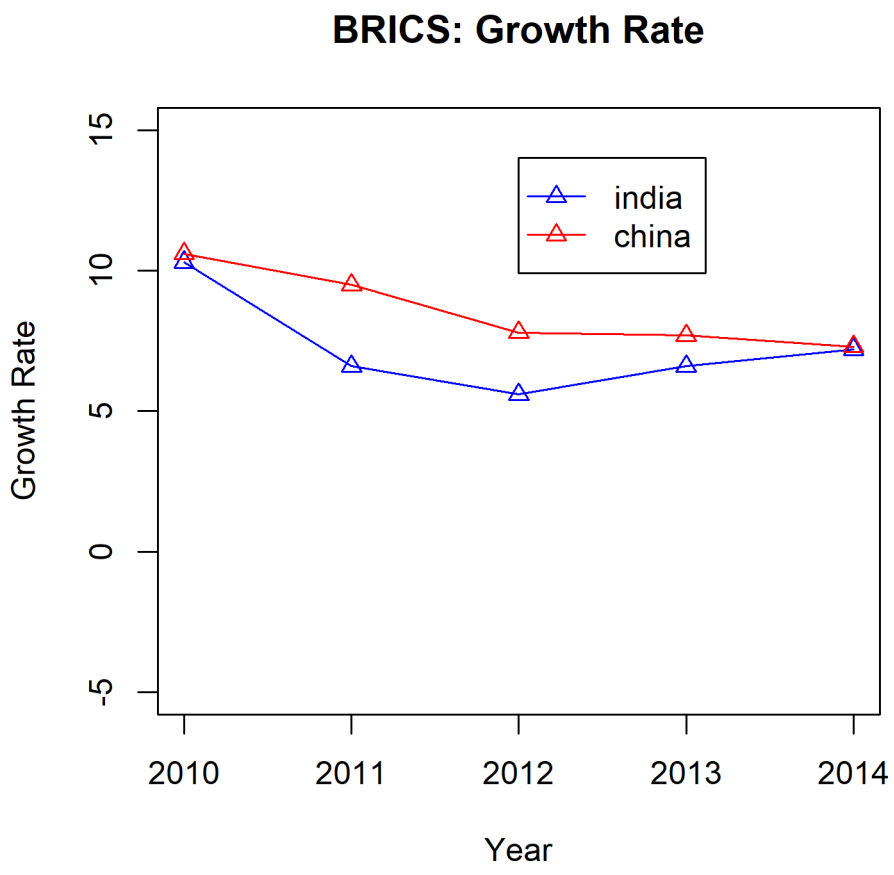
Now we can map the lines to the respective countries using the legend. But our legend looks very simple right. Let us explore the options available to modify and enhance the appearance of the legend.

Points

If the plot has both lines and points, we can use the `pch` argument in the `legend()` function to specify the shape of the point.

```
{plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
points(gdp$year, gdp$india, pch = 2, col = 'blue')
lines(gdp$year, gdp$china, col = 'red')
points(gdp$year, gdp$china, pch = 2, col = 'red')
```

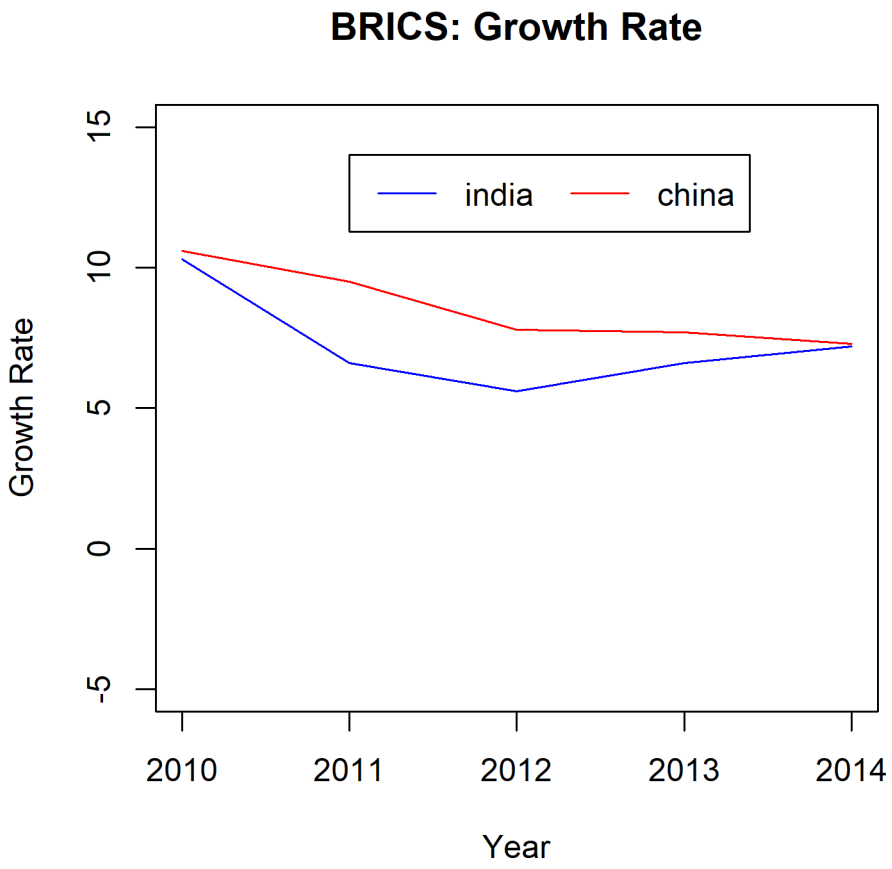
```
legend(x = 2012, y = 14, legend = c('india', 'china'),
       lty = 1, pch = 2, col = c('blue', 'red'))}
```



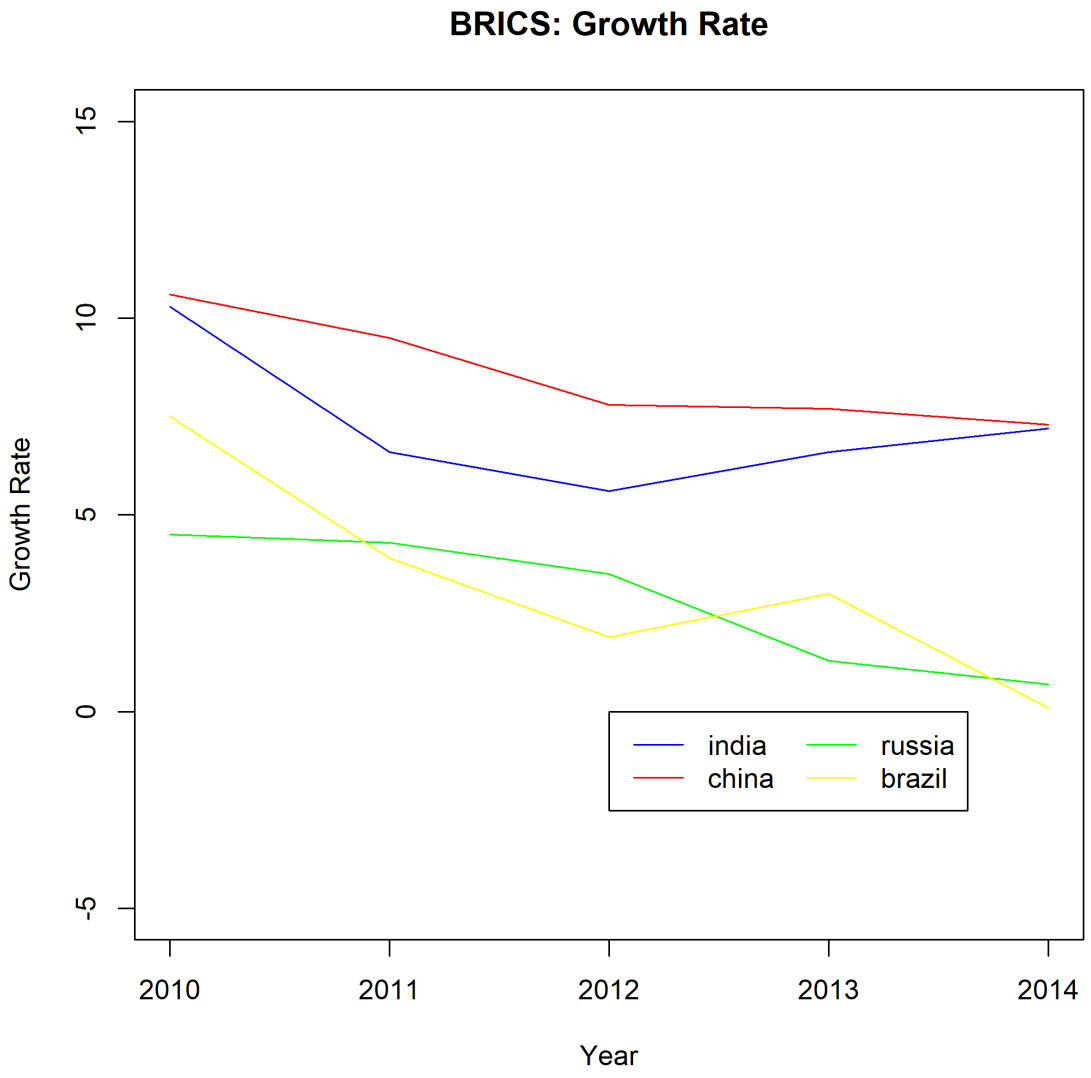
Text Placement

The contents of the legend can be positioned horizontally using the `horiz` argument. It takes logical values as inputs and the default is `FALSE`. Set it to `TRUE` to position the contents horizontally:

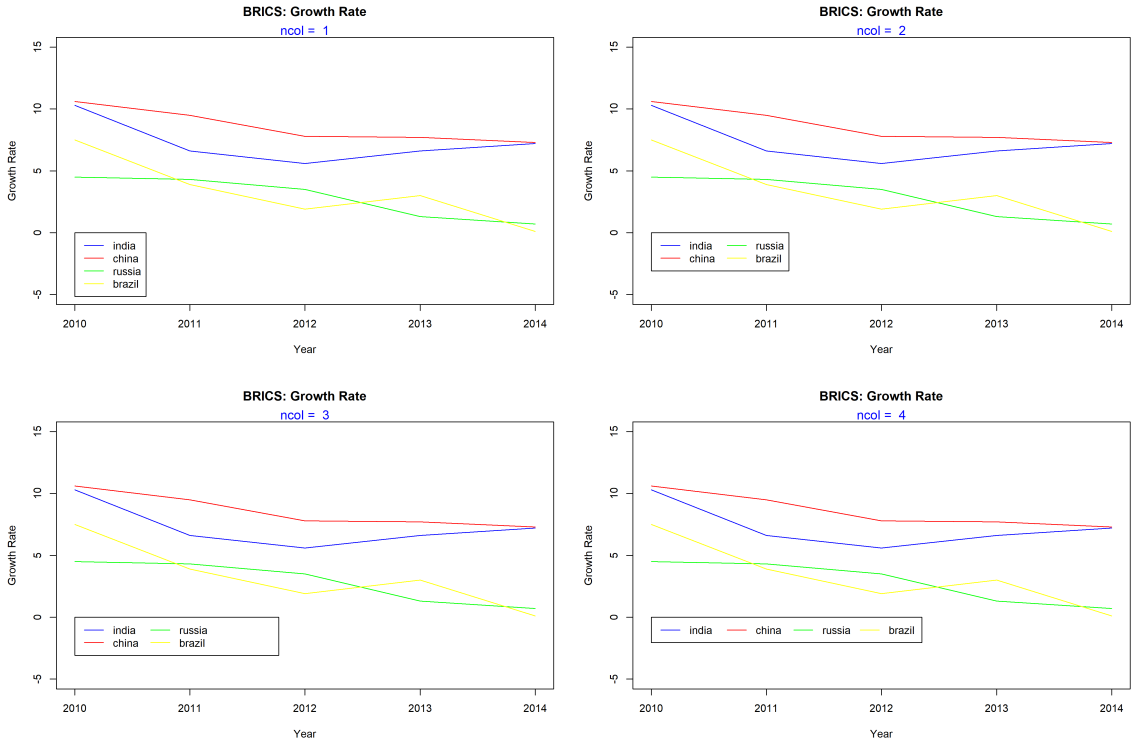
```
{plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
legend(x = 2011, y = 14, legend = c('india', 'china'),
       lty = 1, col = c('blue', 'red'),
       horiz = TRUE)}
```



Another way to position the content inside the legend is to use columns. In the below example, we use the `ncol` argument to split the contents of the legend into two columns instead of the default one.



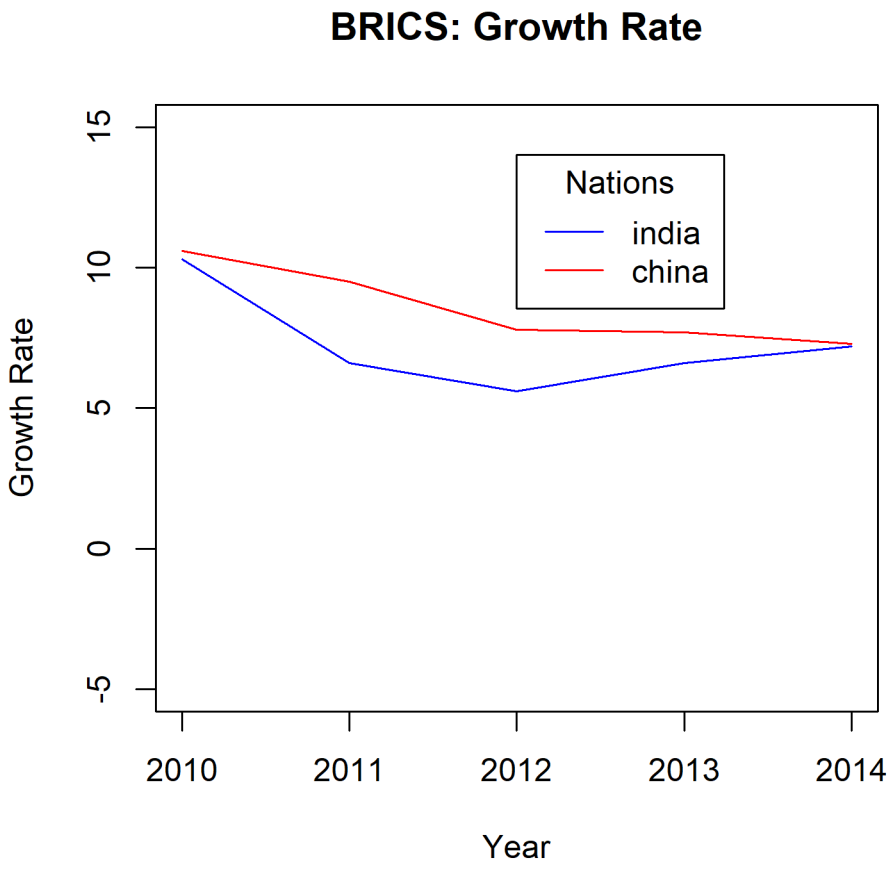
The below plots show the difference in appearance:



Title

Let us add a title to the legend using the `title` argument:

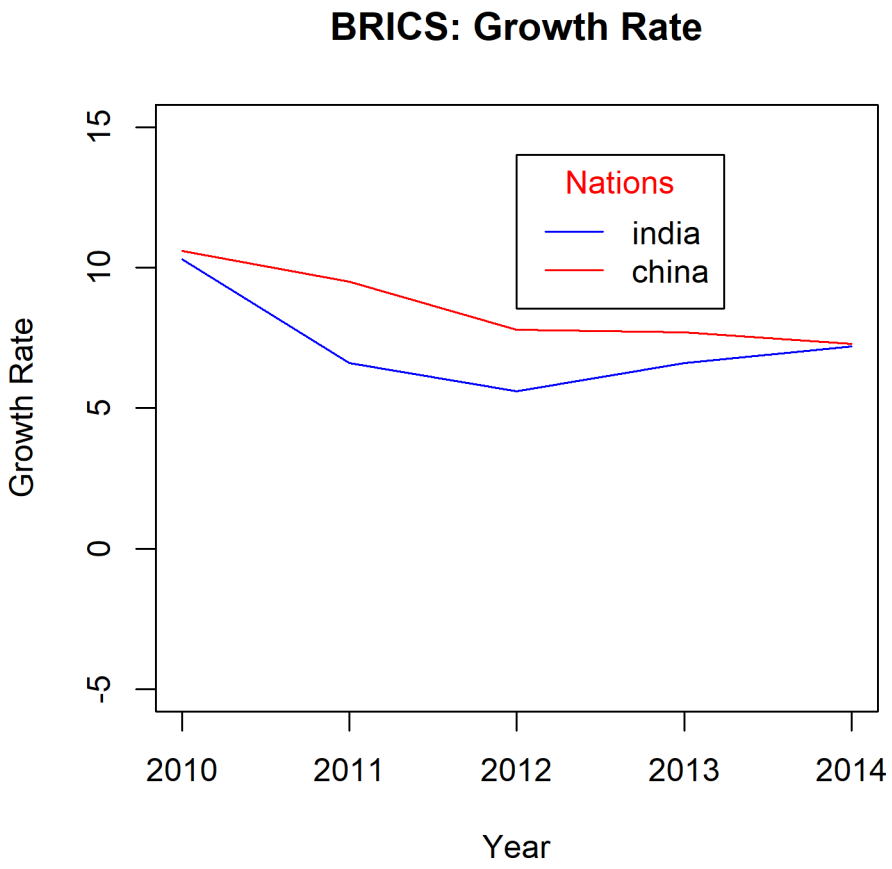
```
{plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
legend(x = 2012, y = 14, legend = c('india', 'china'),
       lty = 1, col = c('blue', 'red'),
       title = 'Nations')}
```



Title Color

The color of the title can be modified using the `title.col` argument:

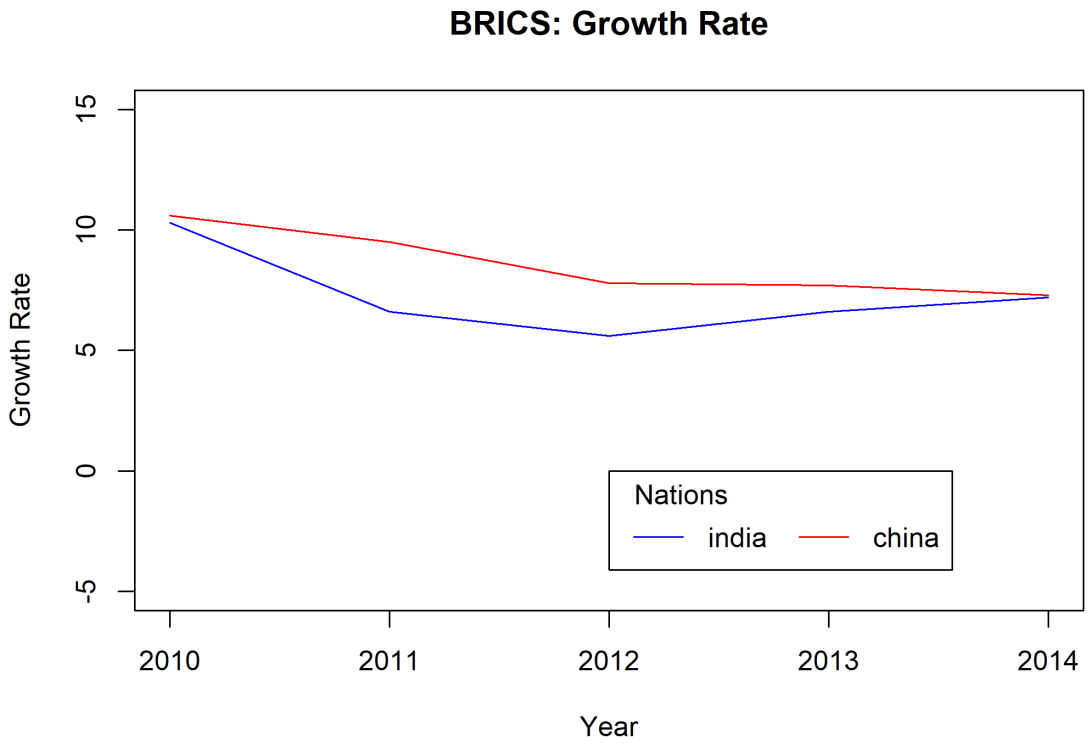
```
{plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
legend(x = 2012, y = 14, legend = c('india', 'china'),
       lty = 1, col = c('blue', 'red'),
       title = 'Nations', title.col = 'red')}
```



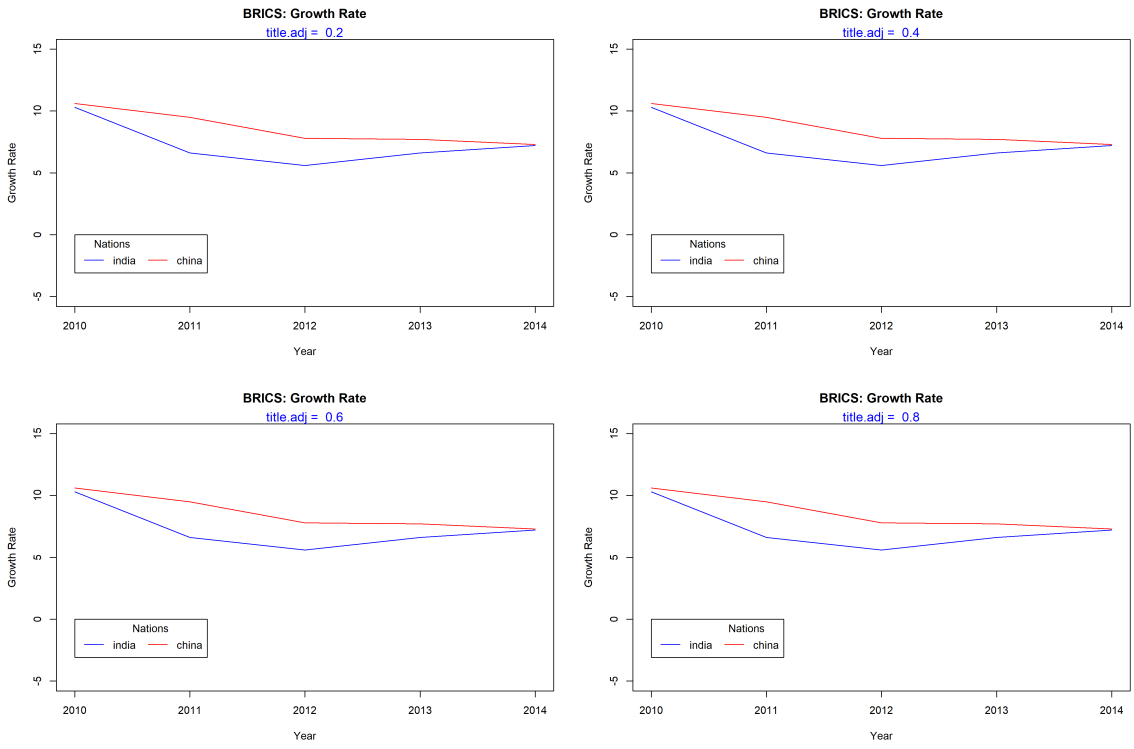
Title Position

The title can be positioned within the legend box using the `title.adj` argument. It will take values between 0 and 1. The default is 0.5 and the title is positioned in the middle of the box. As the value moves away from 0.5, the position of the title moves to the left or right respectively.

```
{plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
legend(x = 2012, y = 0, legend = c('india', 'china'),
       lty = 1, col = c('blue', 'red'), horiz = TRUE,
       title = 'Nations', title.adj = 0.1)}
```



The below plots show the relative position of the title within the legend box for different values between 0 and 1.



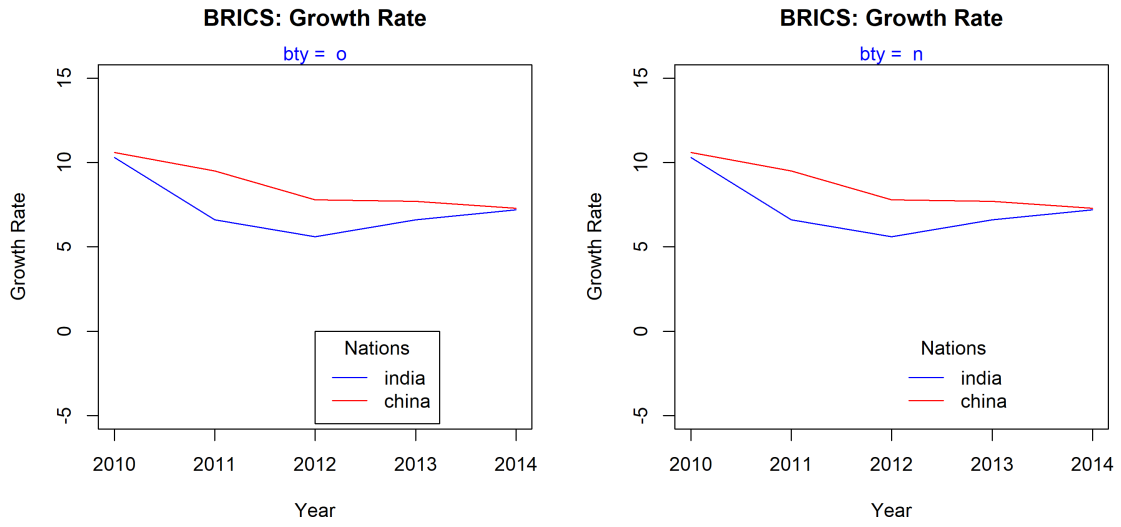
Box Appearance

There are a lot of options to modify the appearance of the legend box. The below table displays the arguments and their descriptions. Let us look at them one by one:

option	argument	values
Box Type	bty	o, n
Background Color	bg	blue, #0000ff
Border Line Type	box.lty	1:5
Border Line Width	box.lwd	0.5, 1, 1.5
Border Line Color	box.col	blue, #0000ff
Horizontal Justification	xjust	0:1
Vertical Justification	yjust	0:1
Text Color	text.col	blue, #0000ff
Text Font	text.font	1:5

Box Type

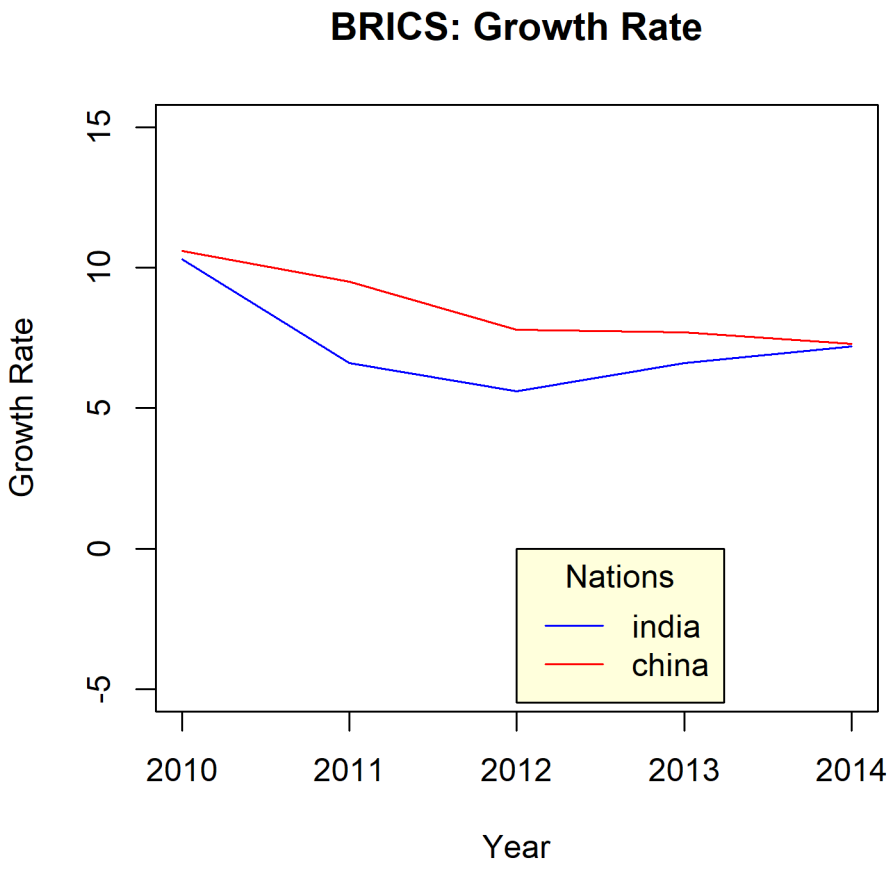
The `bty` argument takes two values, `o` and `n`. If set to `n`, there will be no box around the legend.



Background Color

A background color can be added to the legend box using the `bg` argument. Below is an example:

```
plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
legend(x = 2012, y = 0, legend = c('india', 'china'),
       lty = 1, col = c('blue', 'red'), title = 'Nations',
       bg = 'lightyellow')
```

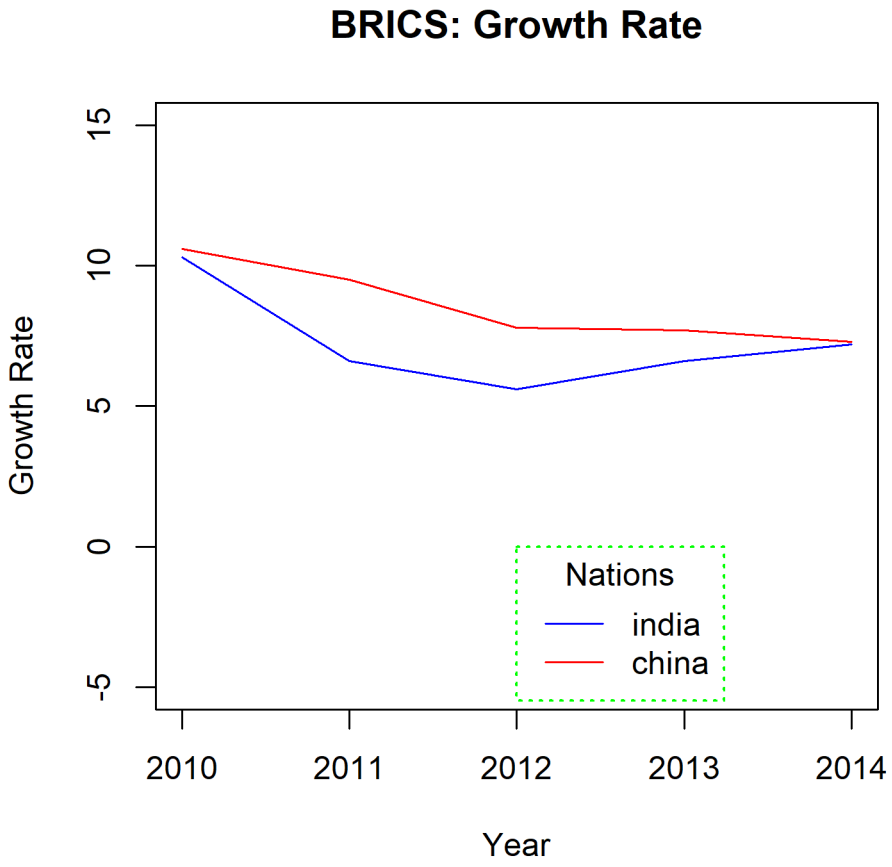


Border

The following arguments can be used to modify the border of the legend box:

- `box.lty`: line type
- `box.lwd`: line width
- `box.col`: color

```
plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
legend(x = 2012, y = 0, legend = c('india', 'china'),
       lty = 1, col = c('blue', 'red'), title = 'Nations',
       box.lty = 3, box.lwd = 1.5, box.col = 'green')
```



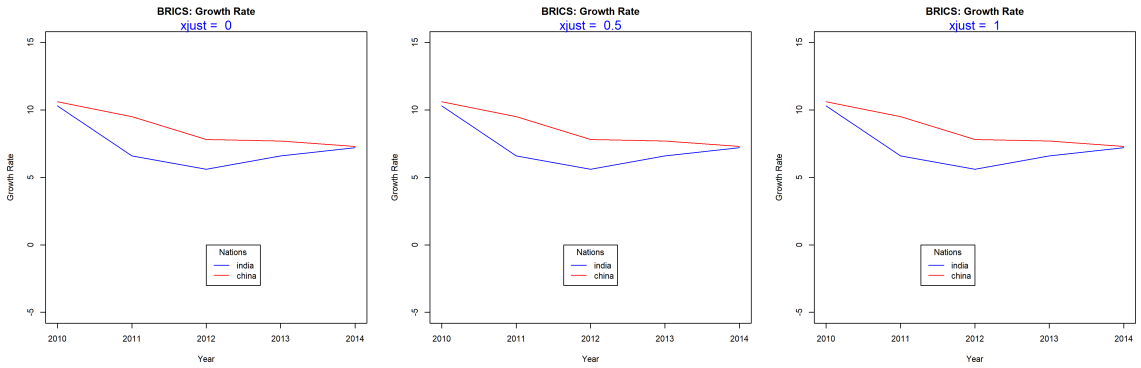
Justification

The `xjust` and `yjust` arguments can be used to position the legend relative to the **X** and **Y** axis respectively. Listed below is the value and the respective justification:

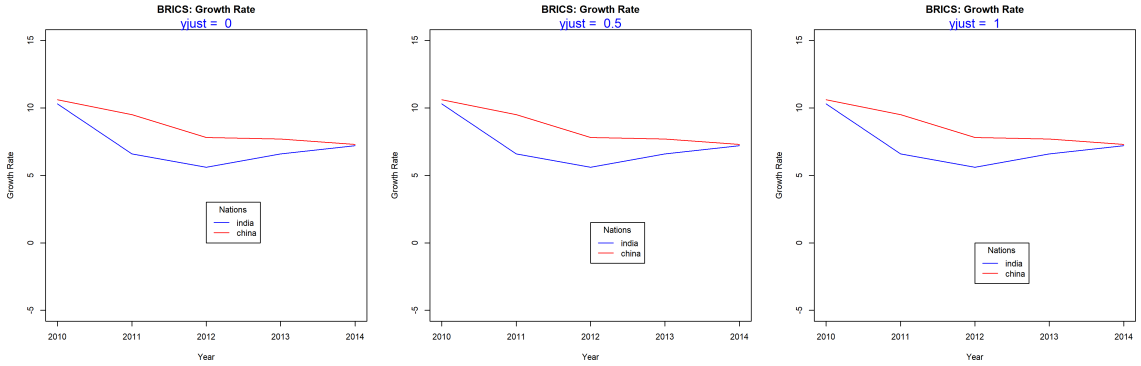
- `0`: left justified
- `0.5`: centered
- `1`: right justified

Let us look at a few examples to understand how it works.

Horizontal Justification



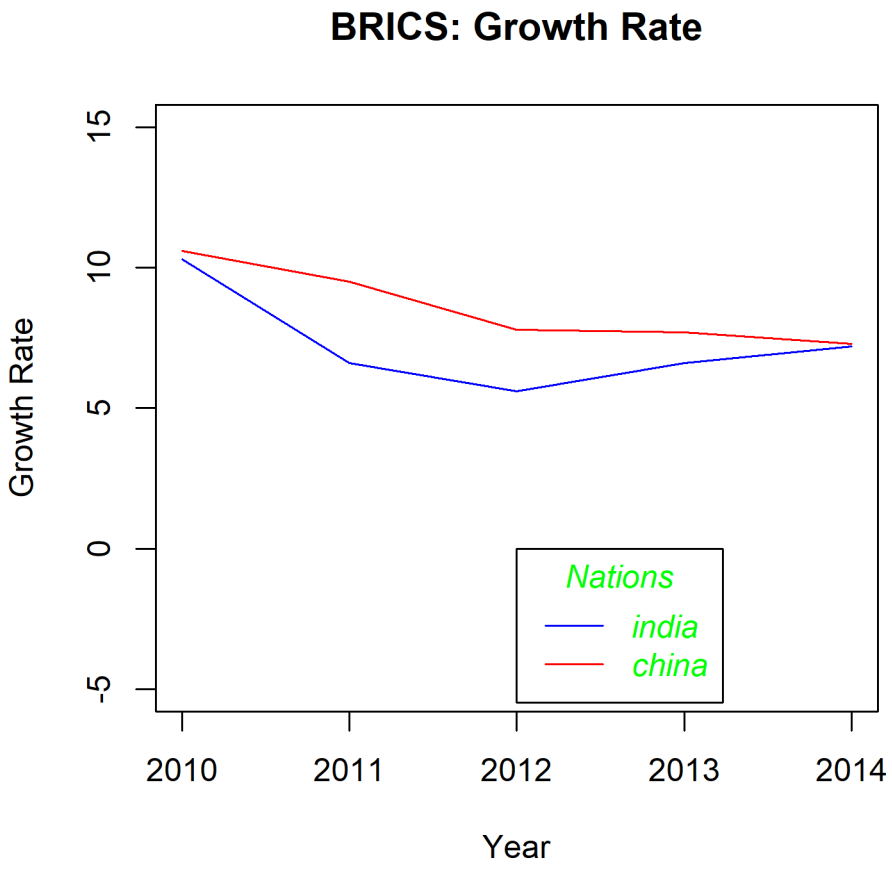
Vertical Justification



Text Appearance

The last topic we will explore is the appearance of the text inside the legend box. We will modify the color and font of text using the `text.col` and `text.font` arguments.

```
{plot(gdp$year, gdp$india, type = 'l',
      ylim = c(-5, 15), xlab = 'Year',
      ylab = 'Growth Rate', col = 'blue',
      main = 'BRICS: Growth Rate')
lines(gdp$year, gdp$china, col = 'red')
legend(x = 2012, y = 0, legend = c('india', 'china'),
       lty = 1, col = c('blue', 'red'), title = 'Nations',
       text.col = 'green', text.font = 3)}
```



Text Annotations

Introduction

In this chapter, we will learn to add text annotations. There are occassions when you want to display additional information in a plot. This is usually achieved by adding text either inside the plot or on the margins. For example, you might want to label a line/bar or add formulas to better communicate what is shown in the plot. The idea is to use the available space within/outside the plot to provide additional information that can be useful to the end users. We will learn to add text inside as well as on the margins of the plot. This is accomplished using the following two functions:

- `text()` : add text inside the plot
- `mtext()` : add text on the margins of the plot

Syntax

Let us take a quick look at the syntax of both the functions:

```
text(x, y = NULL, labels = seq_along(x$x), adj = NULL,
     pos = NULL, offset = 0.5, vfont = NULL,
     cex = 1, col = NULL, font = NULL, ...)

mtext(text, side = 3, line = 0, outer = FALSE, at = NA,
      adj = NA, padj = NA, cex = NA, col = NA, font = NA, ...)
```

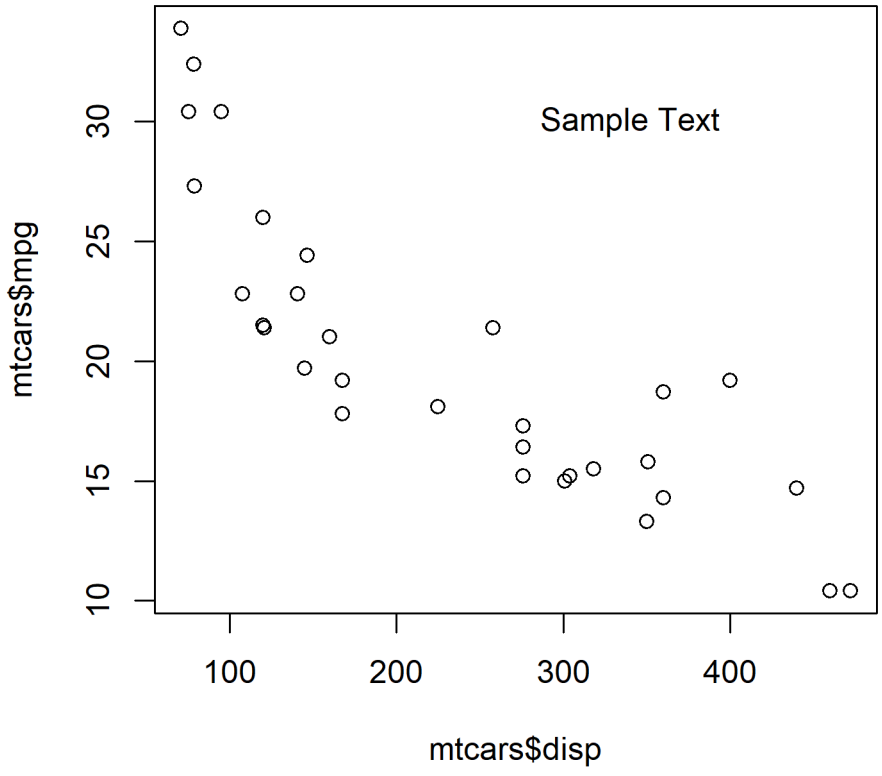
Text Inside the Plot

To add text inside a plot, the following arguments must be supplied to the `text()` function:

- `labels` : the text to be displayed
- `x` : x axis coordinate
- `y` : y axis coordinate

Below is a simple example:

```
plot(mtcars$disp, mtcars$mpg)
text(x = 340, y = 30, labels = 'Sample Text')
```

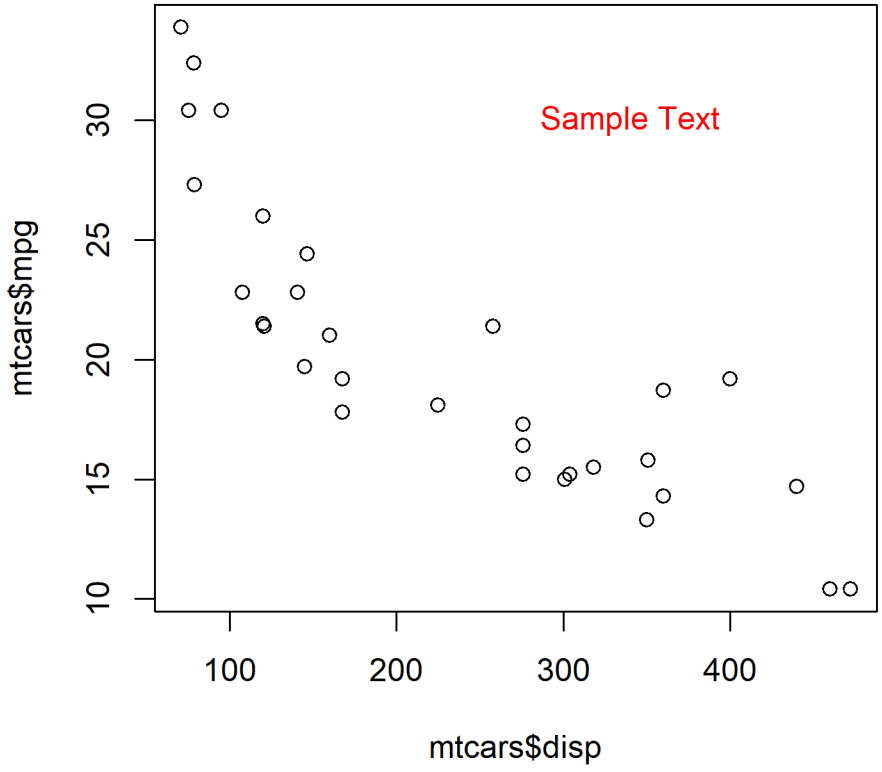


The text appears at the coordinates (340, 30). Ensure that the text is enclosed in single/double quotes and the coordinates provided are within the range of the **X** and **Y** axis variables.

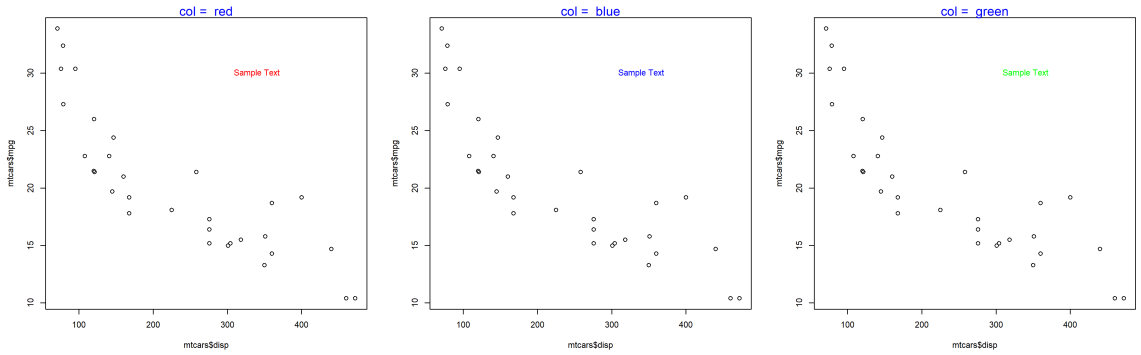
Color

The color of the text can be modified using the `col` argument in the `text()` function.

```
plot(mtcars$disp, mtcars$mpg)
text(x = 340, y = 30, labels = 'Sample Text', col = 'red')
```



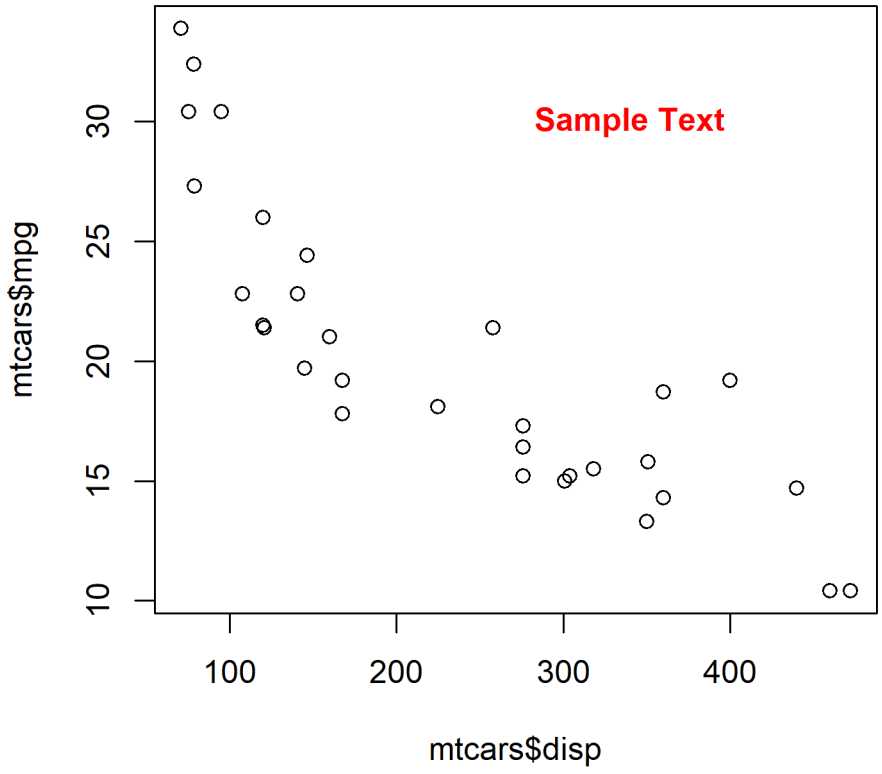
The below plot depicts the appearance of the text for different values of the `col` argument:



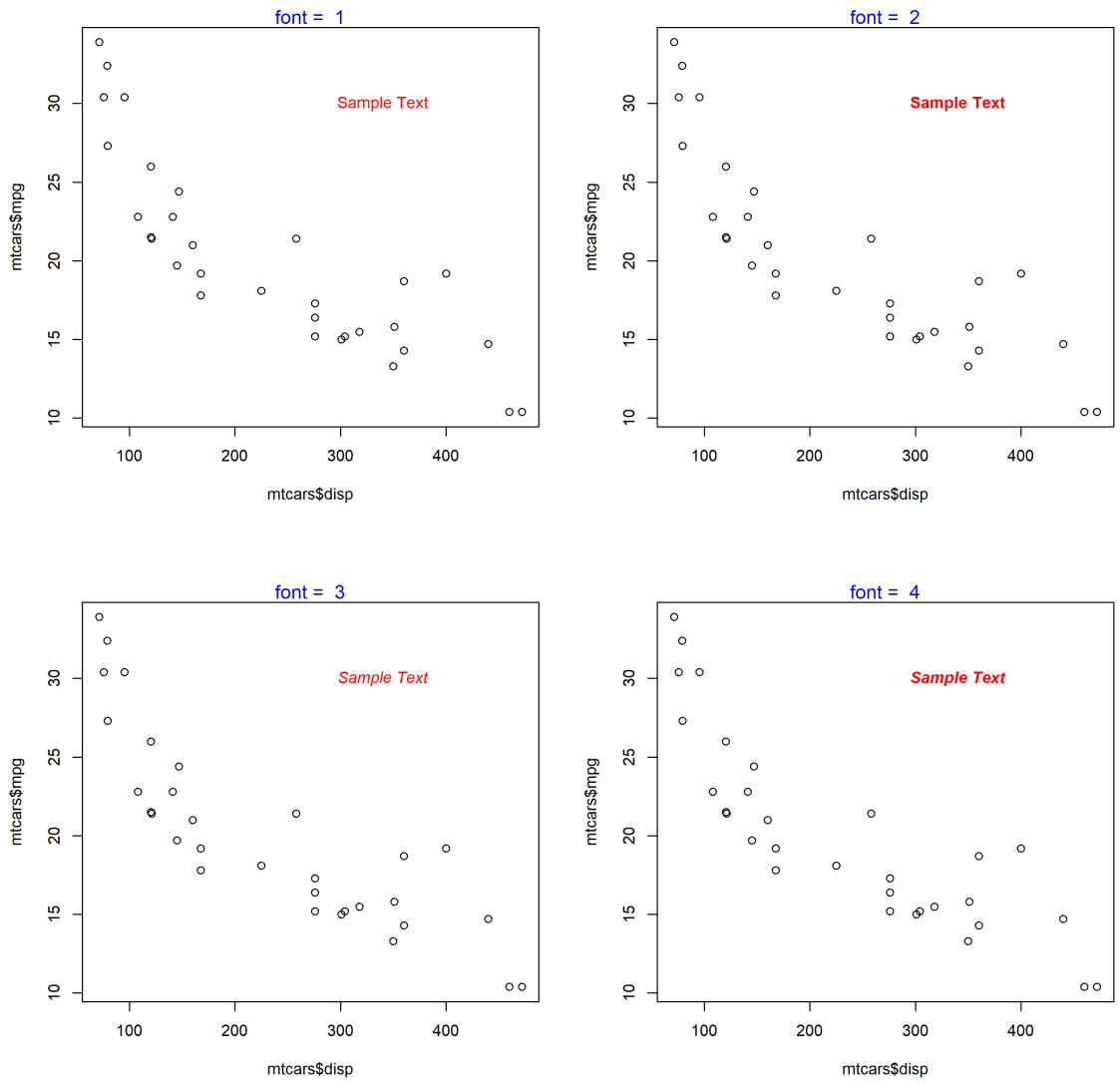
Font

The font of the text can be modified using the `font` argument in the `text()` function.

```
plot(mtcars$disp, mtcars$mpg)
text(x = 340, y = 30, labels = 'Sample Text', col = 'red', font = 2)
```



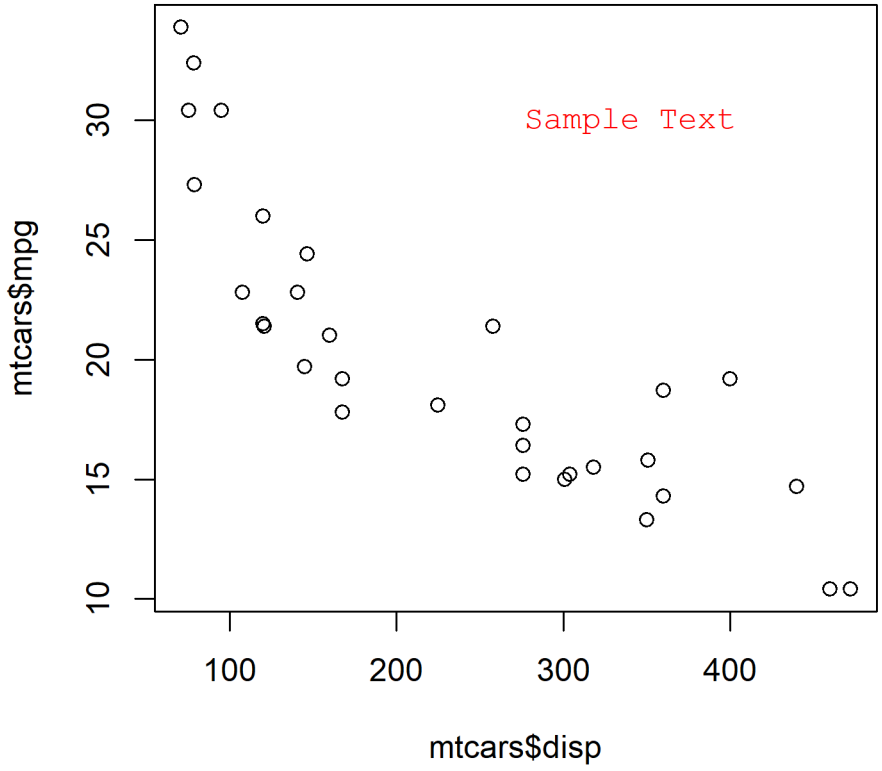
The below plot depicts the appearance of the text for different values of the `font` argument:



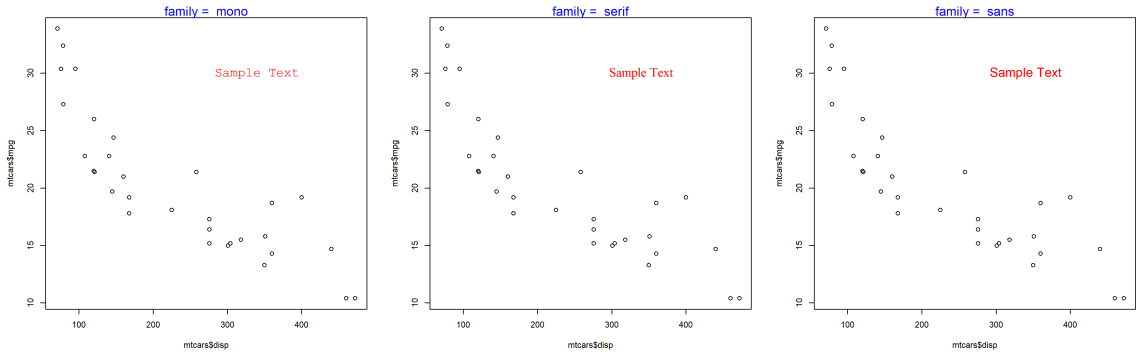
Font Family

The font family of the text can be modified using the `family` argument in the `text()` function.

```
plot(mtcars$disp, mtcars$mpg)
text(x = 340, y = 30, labels = 'Sample Text', col = 'red', family = 'mono')
```



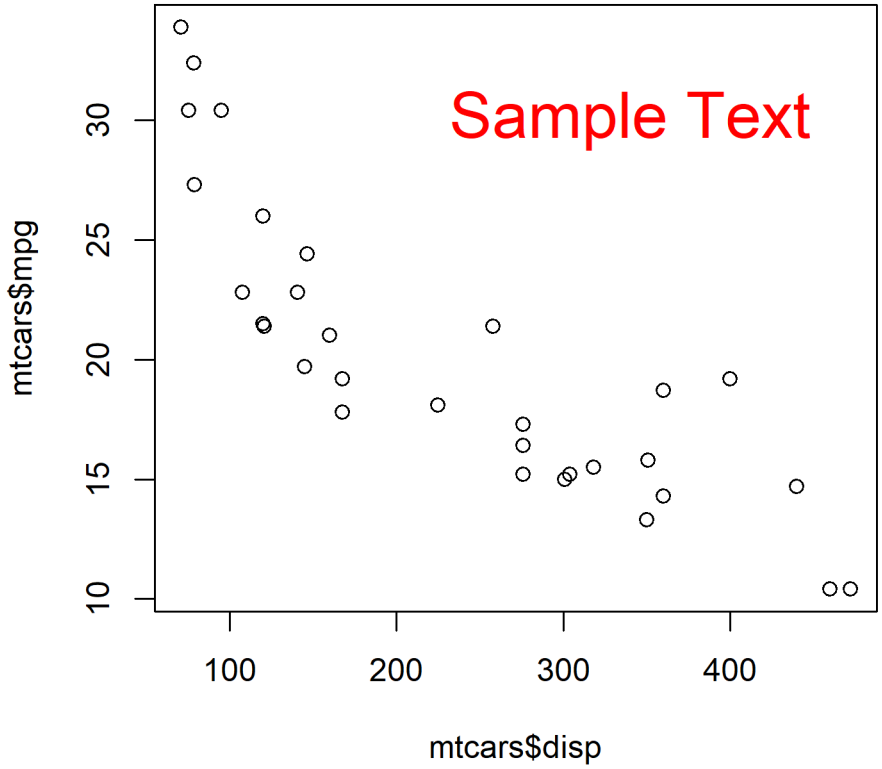
The below plot depicts the appearance of the text for different values of the `family` argument:



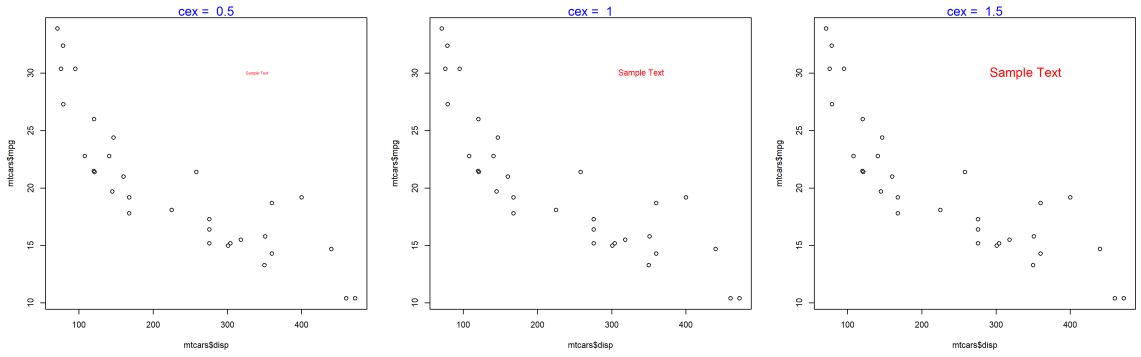
Font Size

The font size of the text can be modified using the `cex` argument in the `text()` function.

```
plot(mtcars$disp, mtcars$mpg)
text(x = 340, y = 30, labels = 'Sample Text', col = 'red', cex = 2)
```



The below plot depicts the appearance of the text for different values of the `cex` argument:

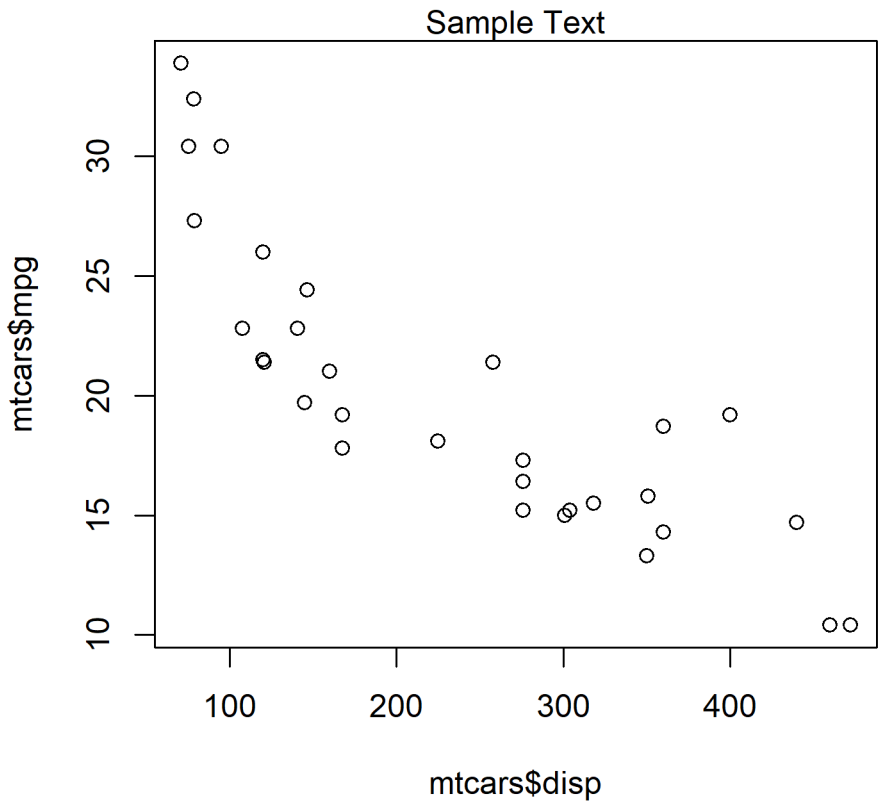


Text on the Margins

The `mtext()` function allows the user to place the text on the margins of the plot. It allows the user to modify the location of the text in multiple ways and we will explore them one by one. To

beign with, let us add text to the plot using the `mtext()` function. The minimum input you need to provide is the text itself. Below is a simple example:

```
plot(mtcars$disp, mtcars$mpg)
mtext('Sample Text')
```

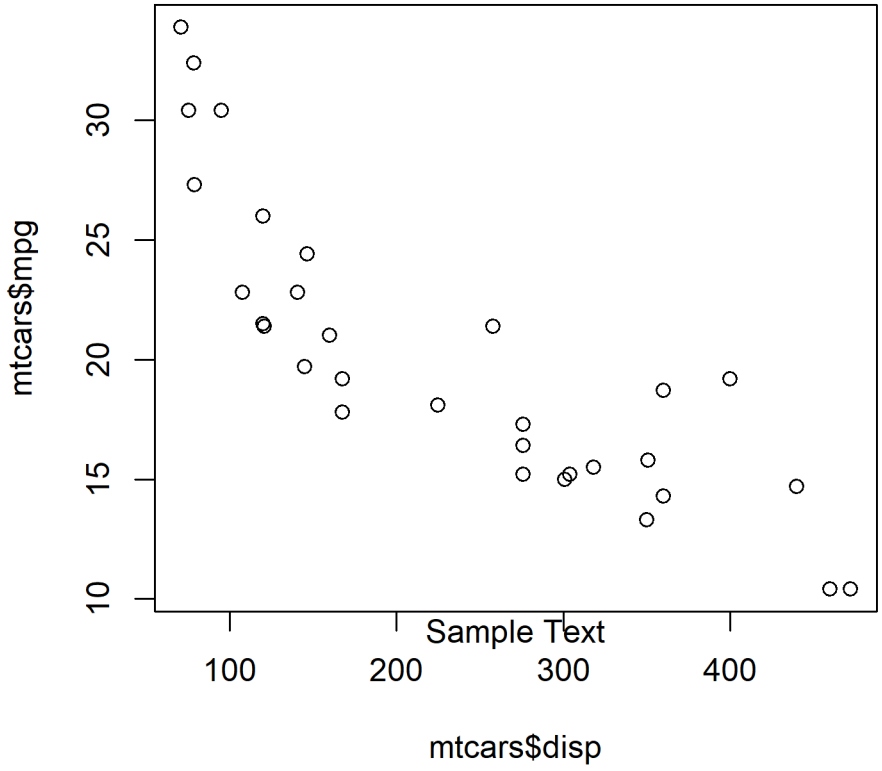


As you can see, the text is placed on the margin of the plot and not inside the plot. Next, we will specify the margin on which to place the text.

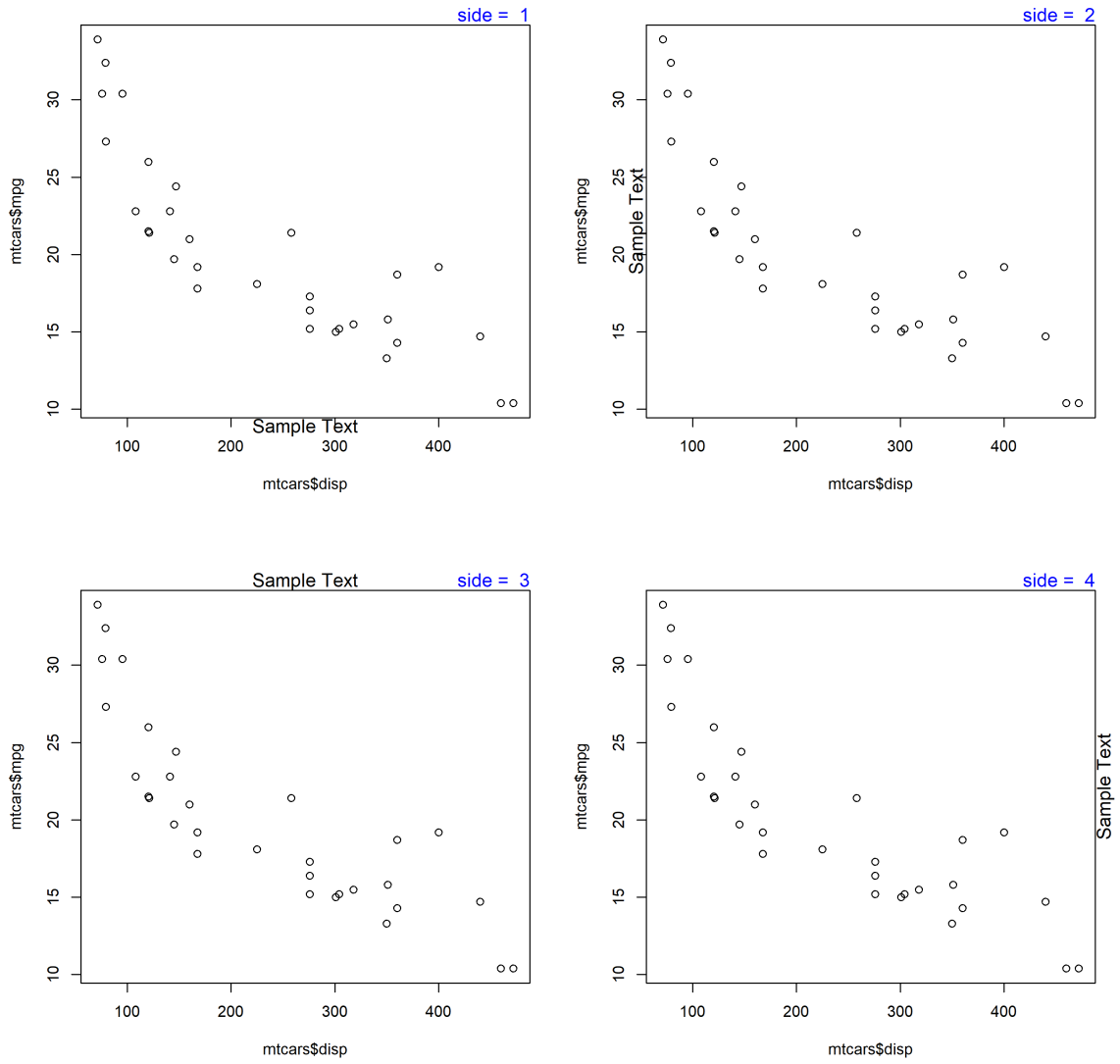
Specify Margin

Use the `side` argument to specify the margin on which you want to place the text. If takes values 1 to 4, each representing one side of the plot.

```
plot(mtcars$disp, mtcars$mpg)
mtext('Sample Text', side = 1)
```



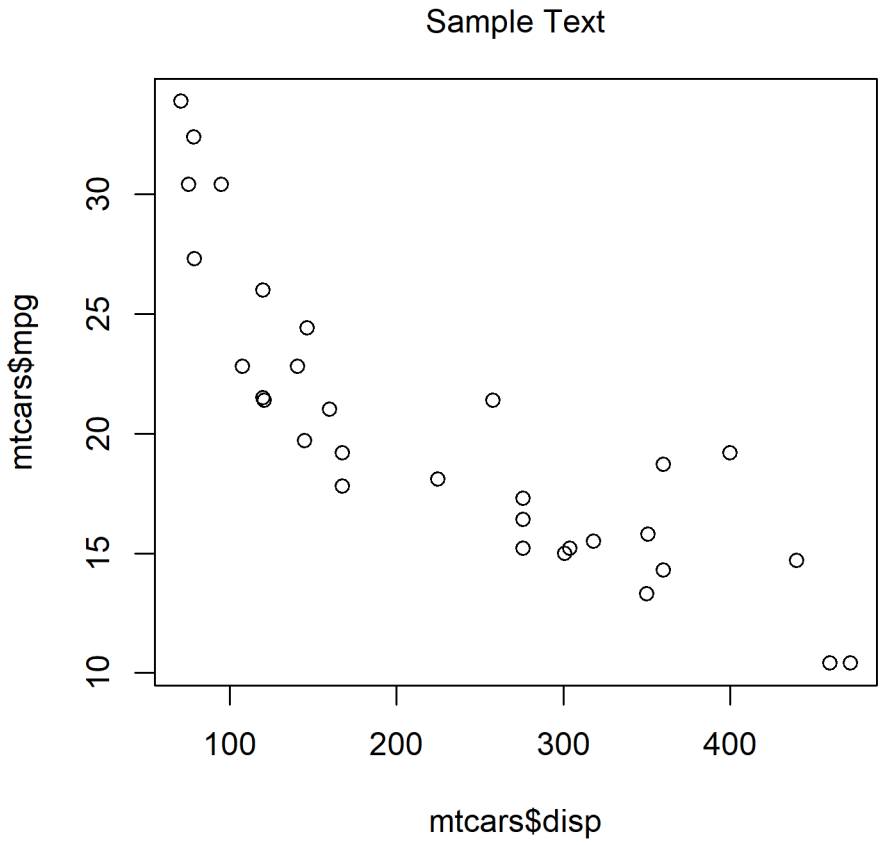
The below plot displays the appearance of the text when differnt options for `side` argument are supplied:



Line

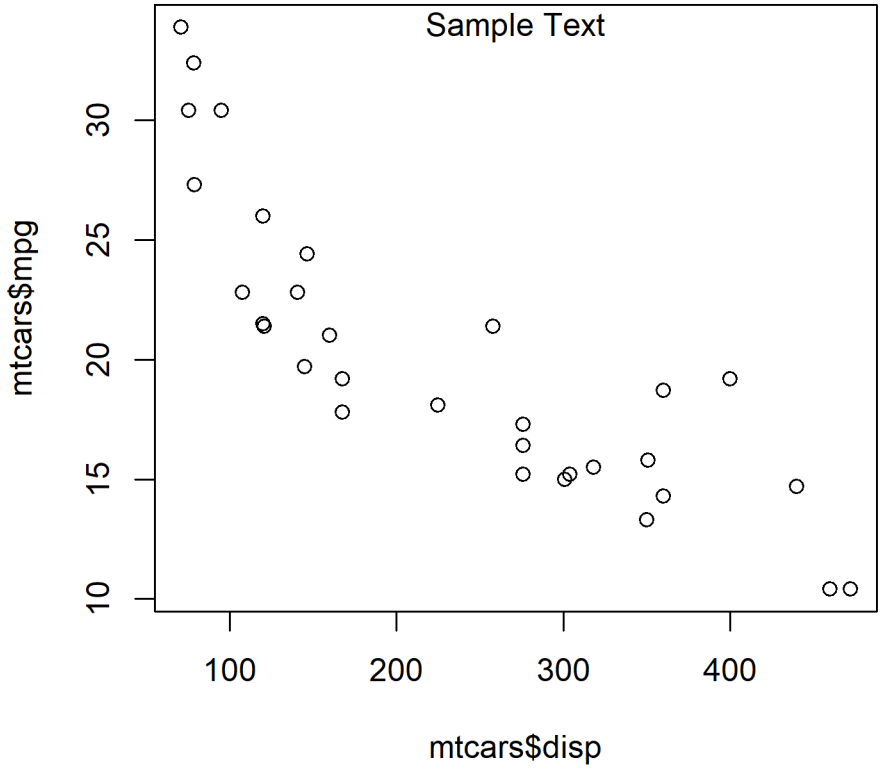
The `line` argument places the text at a specified distance from the margin. The default value is `0`. As the value increases, the text is placed farther from the margin and outside the plot. As the value decreases, the text is placed inside the plot and farther from the margin. Below is a example where the text is placed outside the plot as the value is greater than 1.

```
plot(mtcars$disp, mtcars$mpg)
mtext('Sample Text', line = 1)
```

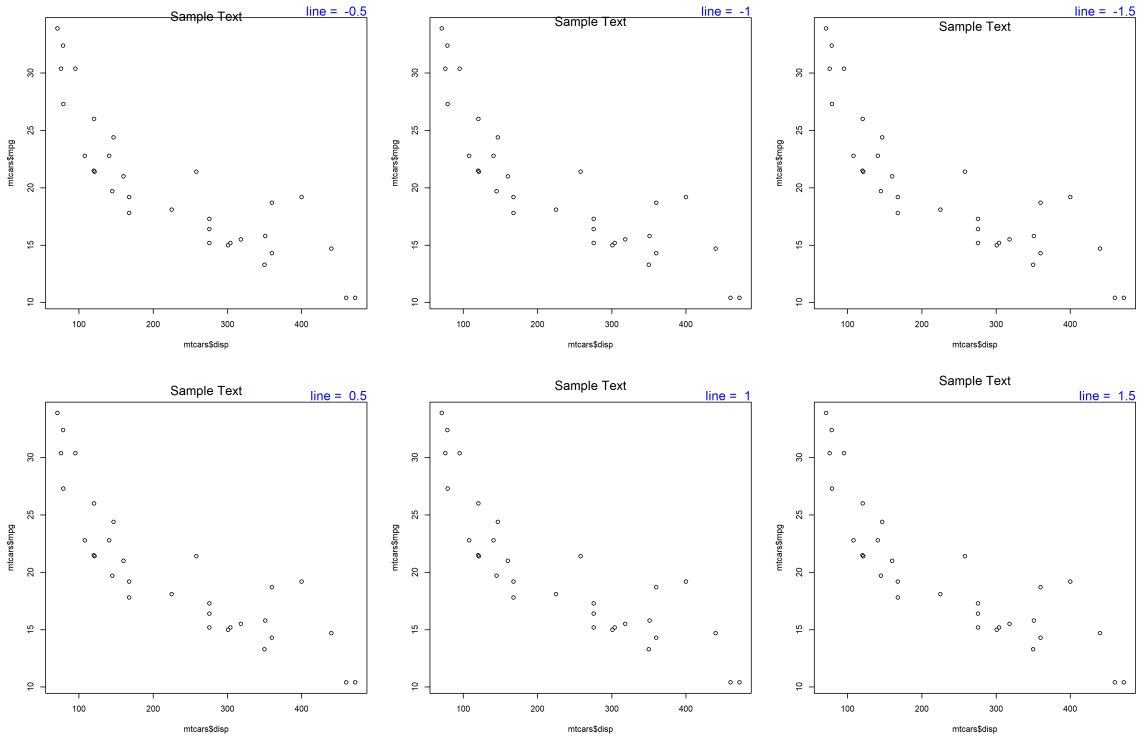


When the value is less than `0`, the line argument places the text inside the plot.

```
plot(mtcars$disp, mtcars$mpg)
mtext('Sample Text', line = -1)
```



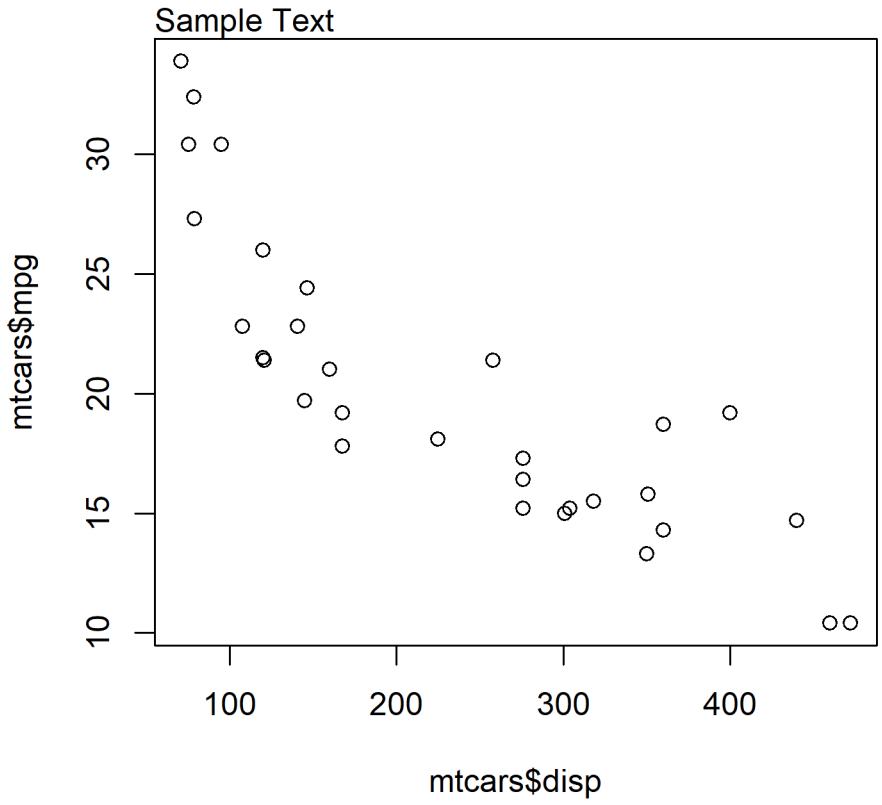
The below plot displays the appearance of the text when different values are supplied to the `line` argument:



Alignment

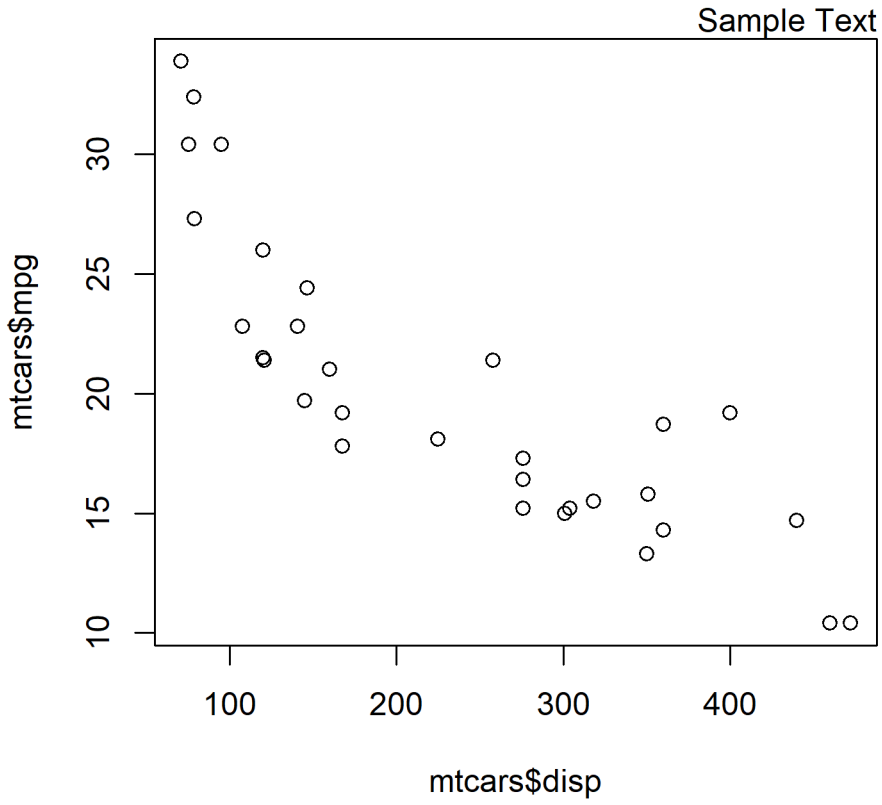
The `adj` argument is used for horizontal alignment of the text. It takes values between 0 and 1. If set to `0`, the text will be left aligned and at `1`, it will be right aligned. Below is a example where the text is left aligned as `adj` is set to `0`.

```
plot(mtcars$disp, mtcars$mpg)
mtext('Sample Text', adj = 0)
```

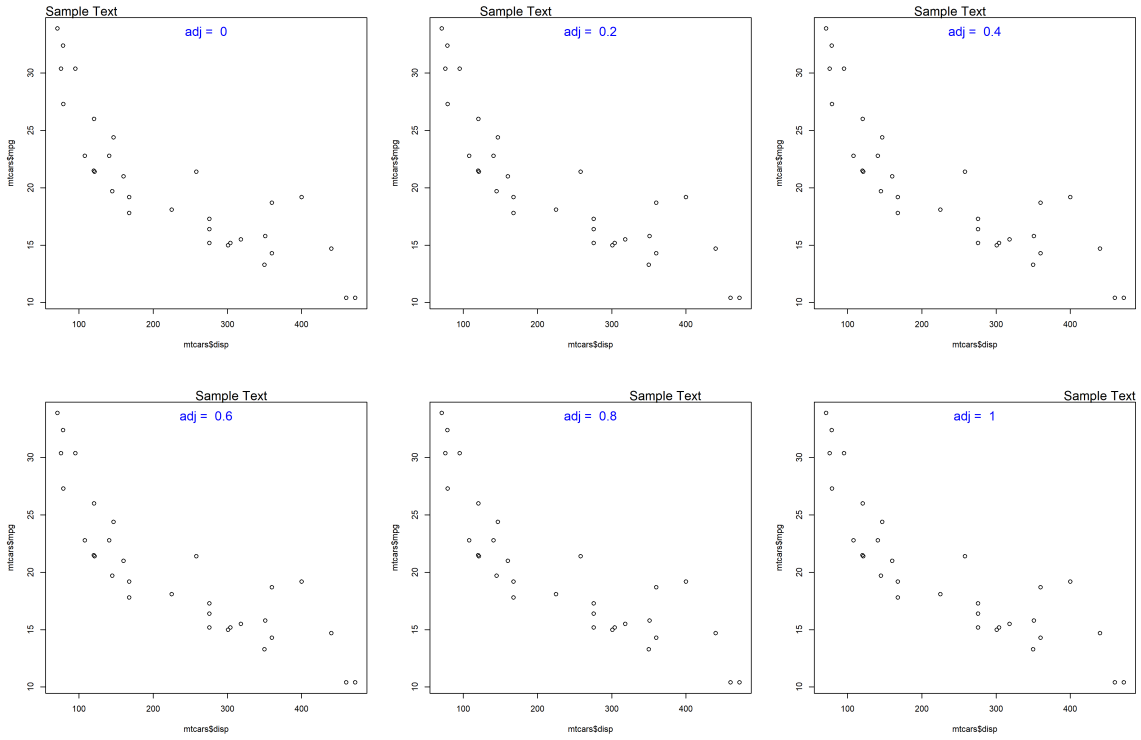


When the value is set to 1, the text is right aligned.

```
plot(mtcars$disp, mtcars$mpg)
mtext('Sample Text', adj = 1)
```



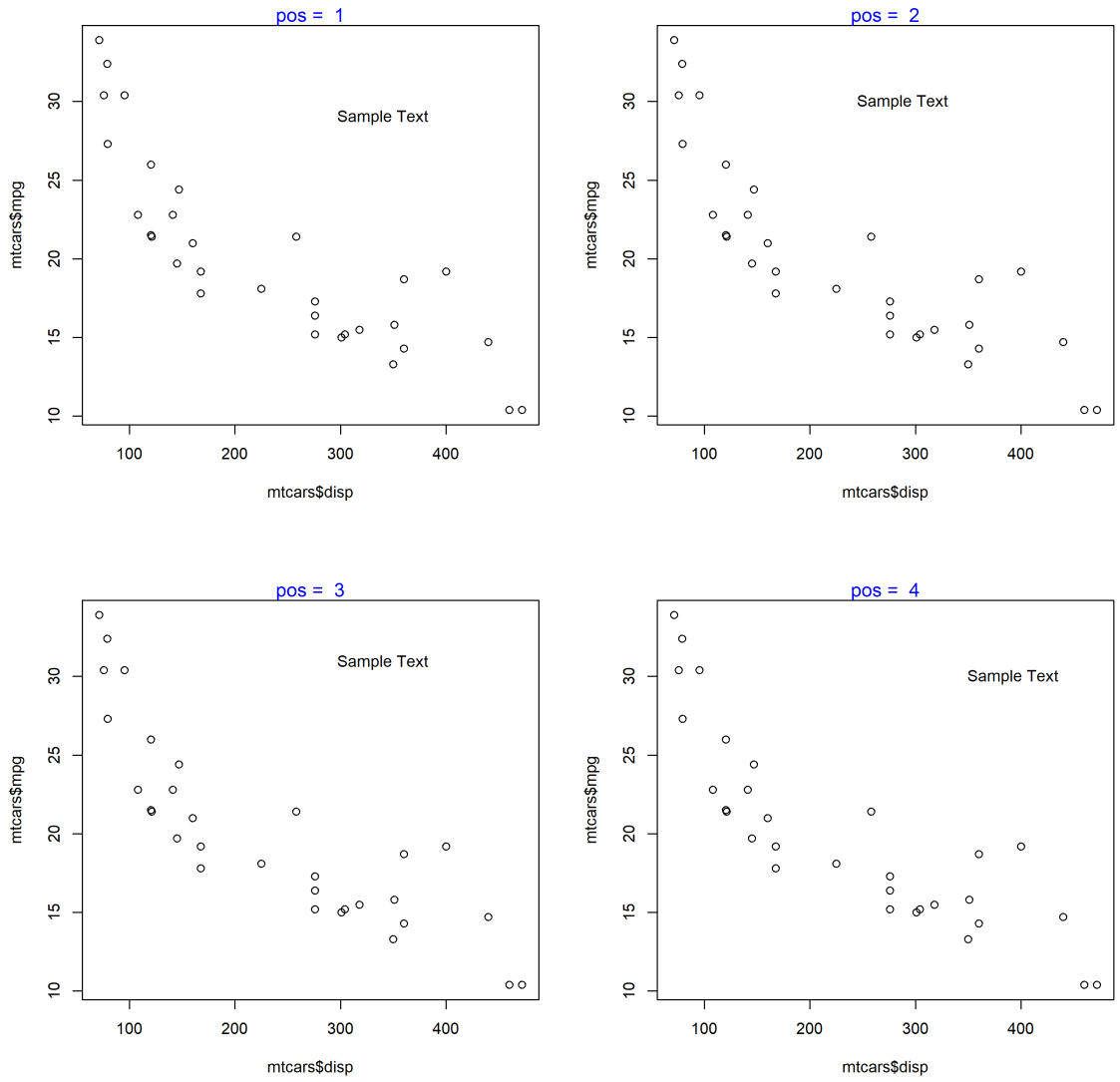
The below plot displays the appearance of the text when different values are supplied to the `adj` argument:



Position

The `pos` argument of the `text()` function places the text relative to the specified coordinates without needing `adj`. It takes values 1 to 4, each representing a position around the point:

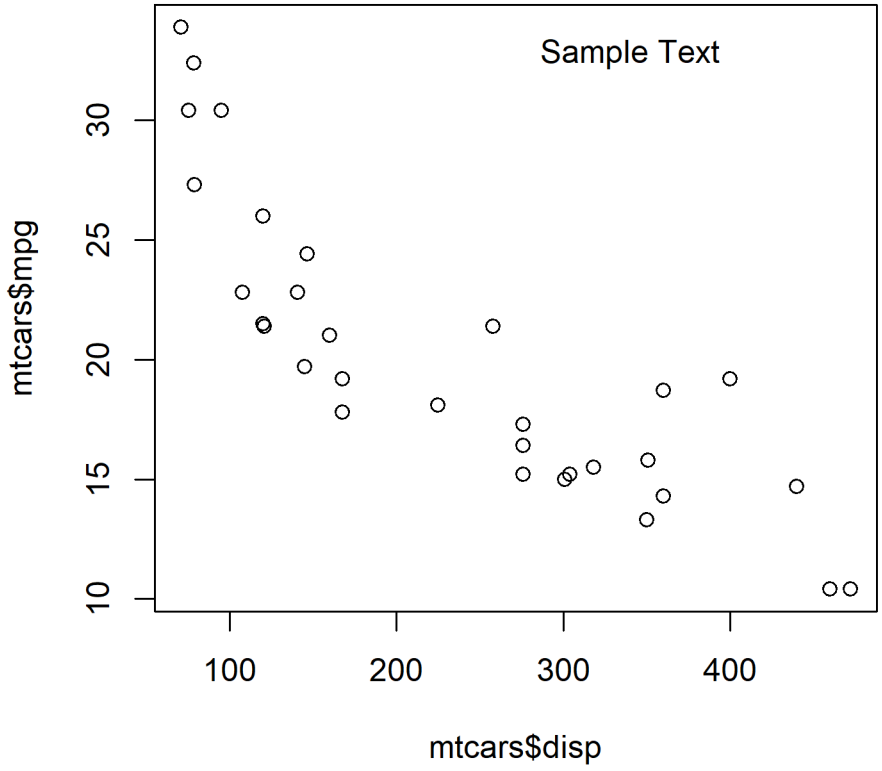
- 1: below
- 2: to the left
- 3: above
- 4: to the right



Offset

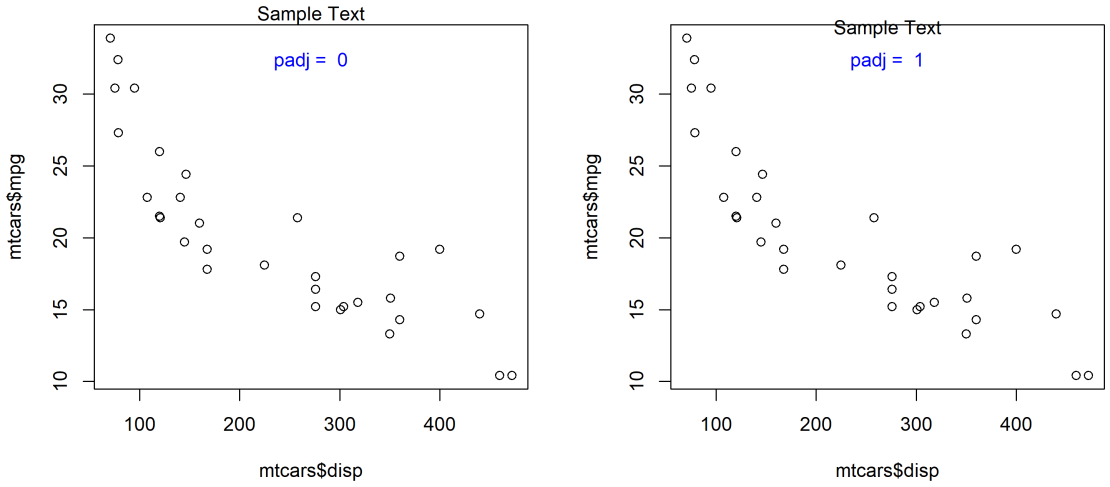
The `offset` argument of the `text()` function controls how far the text is placed from the coordinates when `pos` is used. The distance is measured in character widths and the default value is `0.5`.

```
plot(mtcars$disp, mtcars$mpg)
text(x = 340, y = 30, labels = 'Sample Text', pos = 3, offset = 1.5)
```



Perpendicular Adjustment

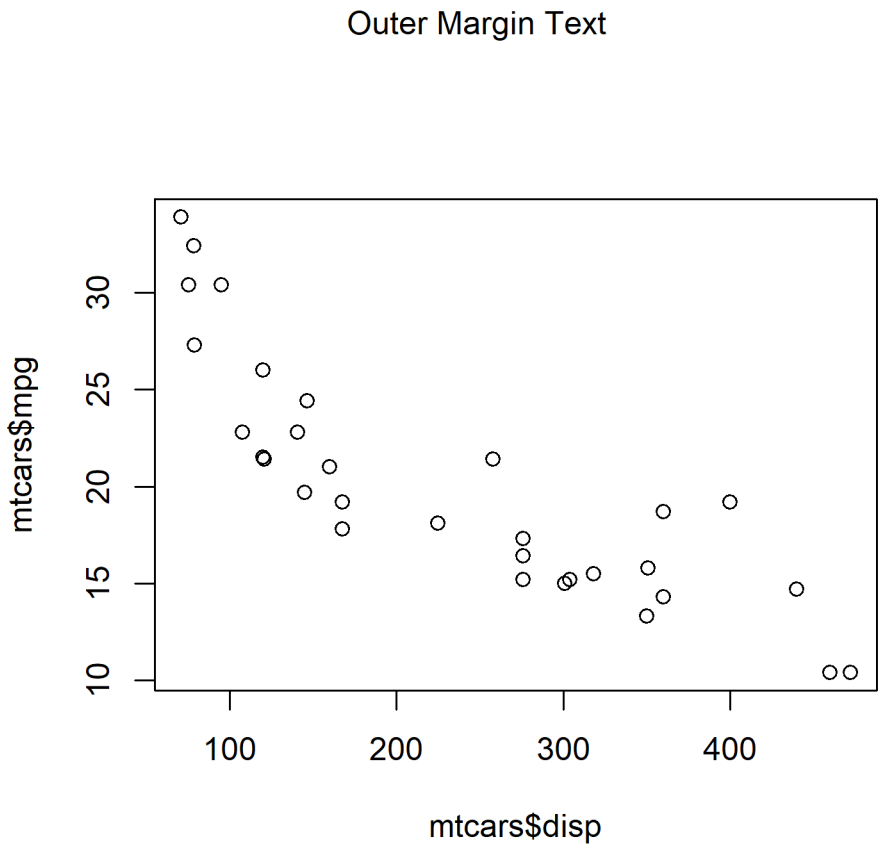
The `padj` argument of the `mtext()` function fine-tunes the placement of the text in the direction perpendicular to `adj`. It takes values between 0 and 1. Compare the two plots below and observe the nudge:



Outer Margin

By default, `mtext()` places text on the inner margin of the current plot. Set `outer = TRUE` to place the text on the outer margin of the whole figure region instead. This is useful with multi-panel layouts, but first reserve outer space with `par(oma = ...)`:

```
init <- par(no.readonly = TRUE)
par(oma = c(0, 0, 3, 0))
plot(mtcars$disp, mtcars$mpg)
mtext('Outer Margin Text', outer = TRUE)
```

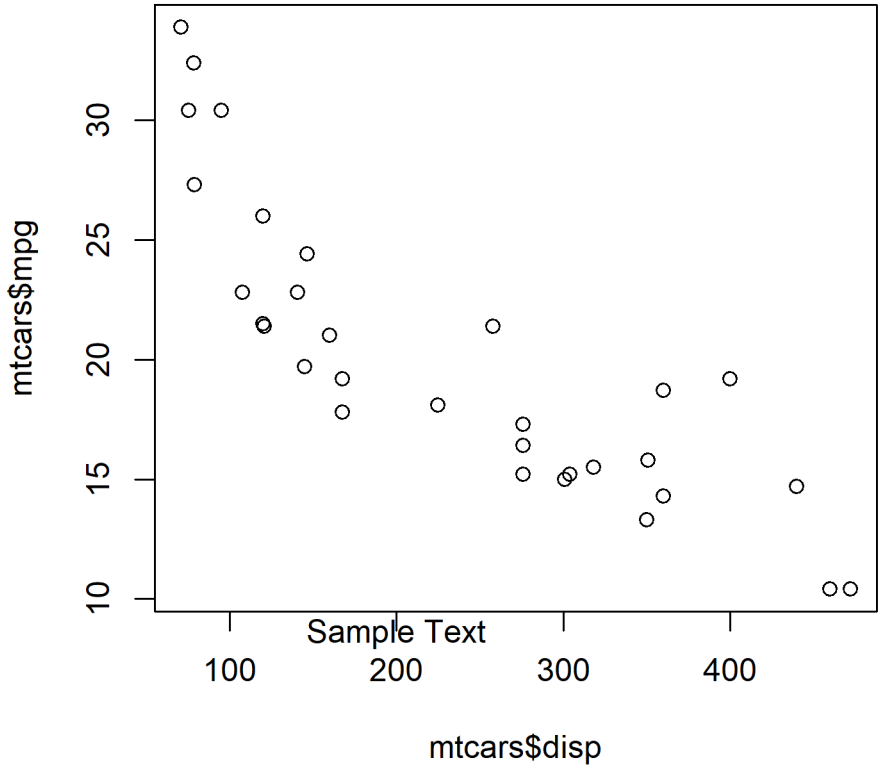



```
par(init)
```

Exact Position

The `at` argument of the `mtext()` function places the text at an exact position along the margin, expressed in the user coordinates of the corresponding axis. Below, the text is anchored at `disp = 200` on the bottom margin instead of the default center:

```
plot(mtcars$disp, mtcars$mpg)
mtext('Sample Text', side = 1, at = 200)
```



Faceting

Introduction

In this chapter, we will learn how to combine multiple plots. Often, it is useful to have multiple plots in the same frame as it allows us to get a comprehensive view of a particular variable or compare among different variables. The Graphics package offers two methods to combine multiple plots. `par()` can be used to set graphical parameters regarding plot layout using the mfcol and mfrow arguments. `layout()` serves the same purpose but offers more flexibility by allowing us to modify the height and width of rows and columns.

`par()` allows us to customize the graphical parameters(title, axis, font, color, size) for a particular session. For combining multiple plots, we can use the graphical parameters mfrow and mfcol. These two parameters create a matrix of plots filled by rows and columns respectively. Let us combine plots using both the above parameters.

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

Row

mfrow combines plots filled by rows i.e it takes two arguments, the number of rows and number of columns and then starts filling the plots by row. Below is the syntax for mfrow.

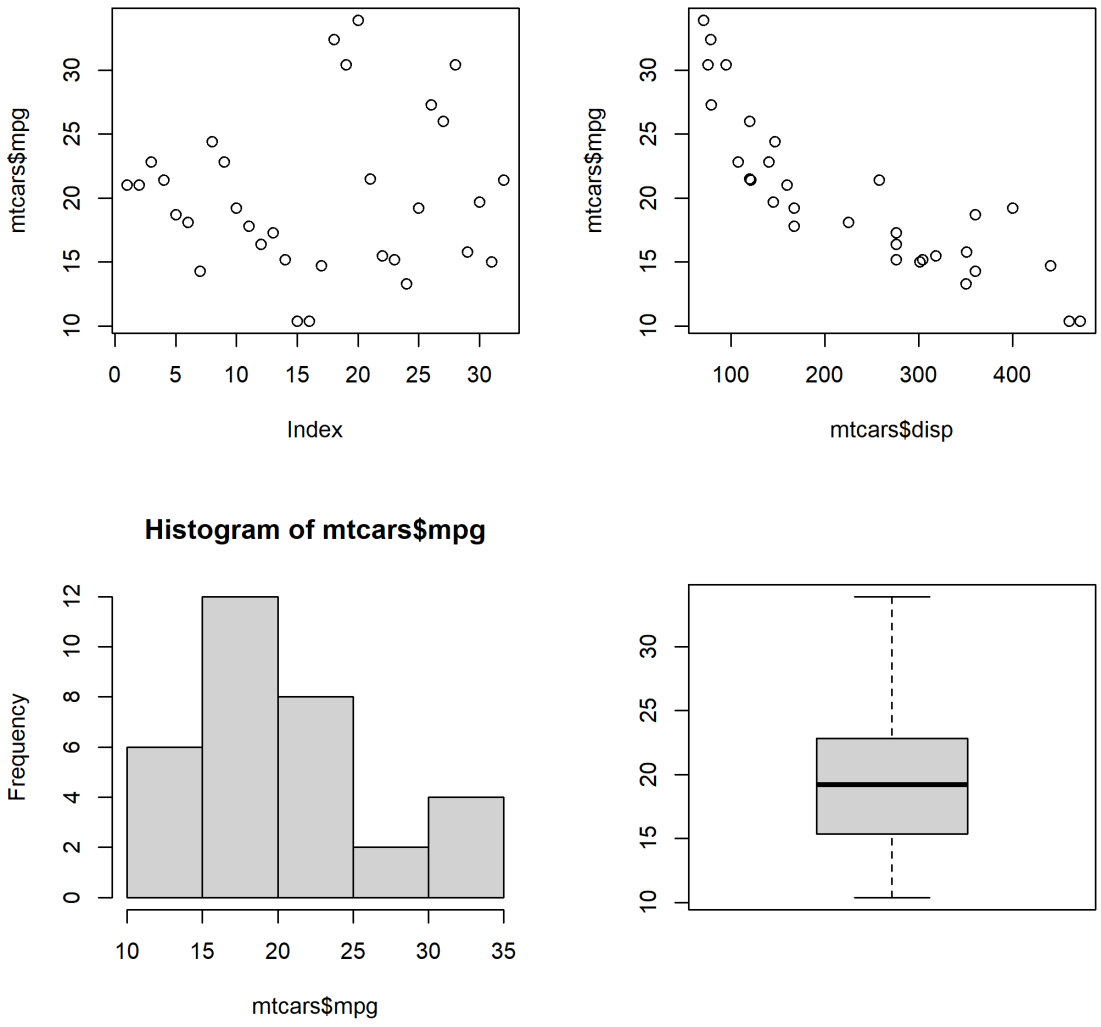
```
# mfrow syntax
par(mfrow = c(number of rows, number of columns))
```

Let us begin by combining 4 plots in 2 rows and 2 columns:

Case Study 1

Let us begin by combining 4 plots in 2 rows and 2 columns. The plots will be filled by rows as we are using the mfrow function:

```
# store the current parameter settings in init
init <- par(no.readonly=TRUE)

# specify that 4 graphs to be combined and filled by rows
par(mfrow = c(2, 2))

# specify the graphs to be combined
plot(mtcars$mpg)
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```



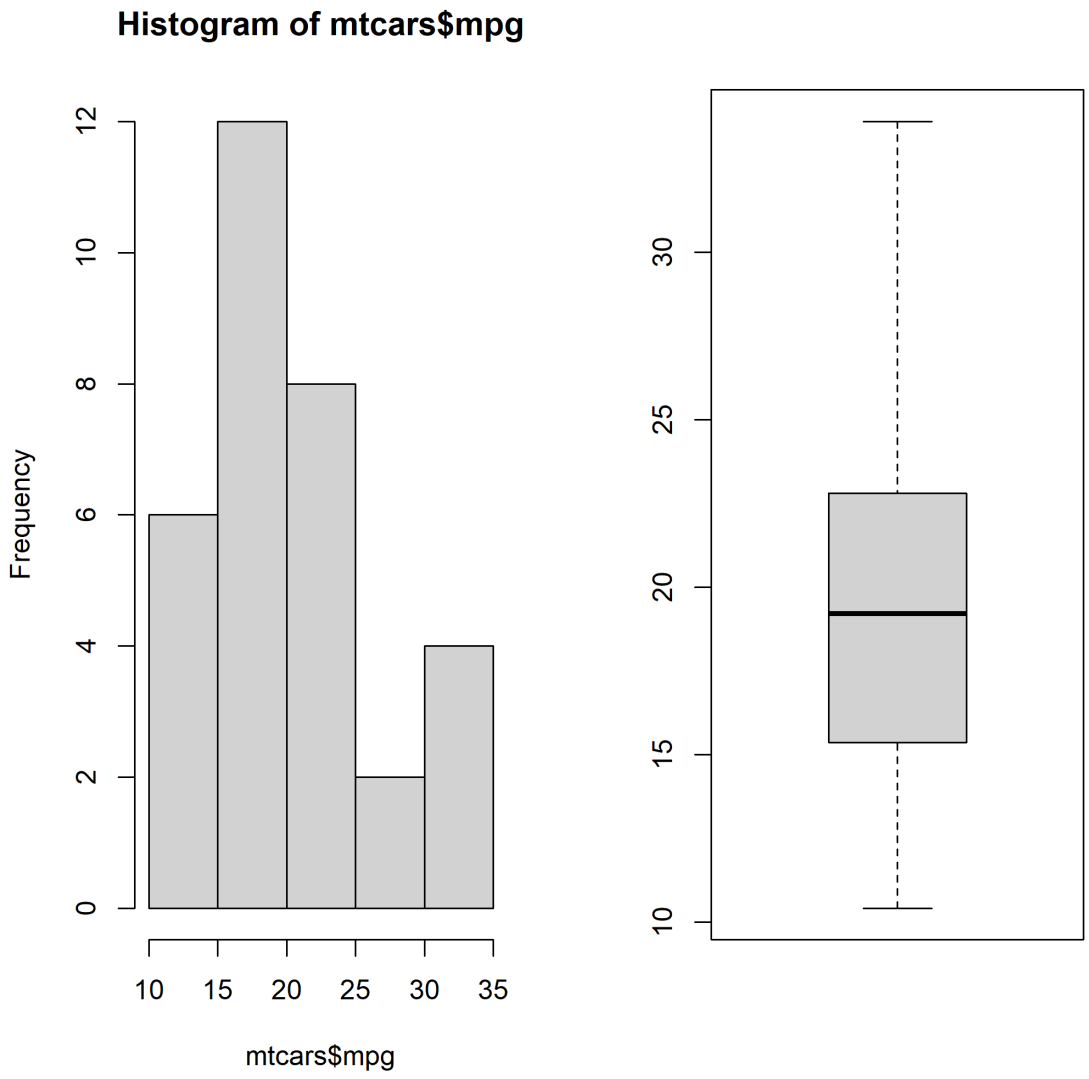

```
# restore the setting stored in init
par(init)
```

Case Study 2

Combine 2 plots in 1 row and 2 columns.

```
# store the current parameter settings in init
init <- par(no.readonly=TRUE)

# specify that 2 graphs to be combined and filled by rows
par(mfrow = c(1, 2))

# specify the graphs to be combined
```

```r
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```



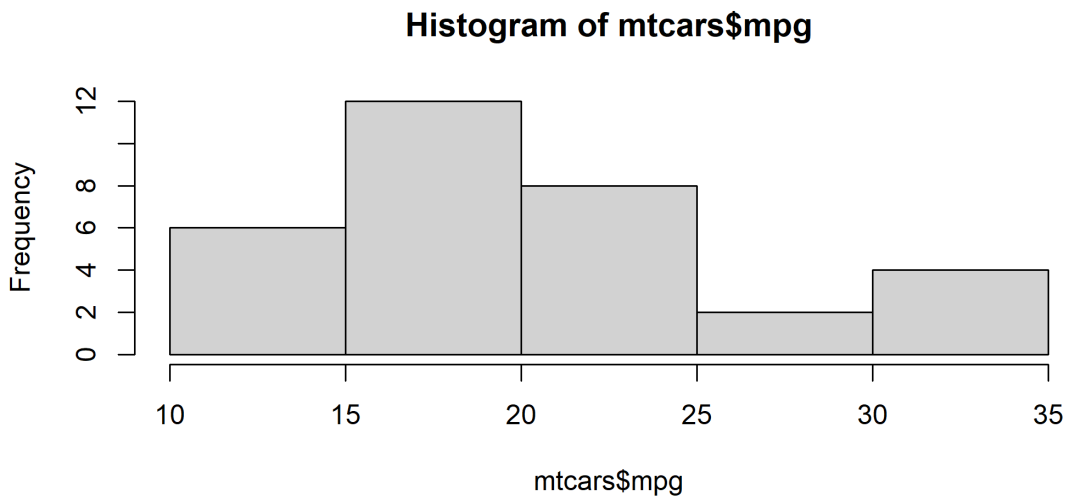

```r
# restore the setting stored in init
par(init)
```

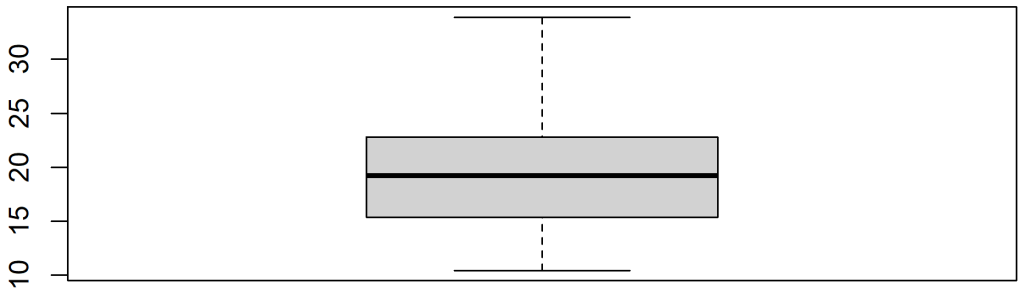
Case Study 3

Combine 2 plots in 2 rows and 1 column.

```r
# store the current parameter settings in init
init <- par(no.readonly=TRUE)

# specify that 2 graphs to be combined and filled by rows
```

```r
par(mfrow = c(2, 1))

# specify the graphs to be combined
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```



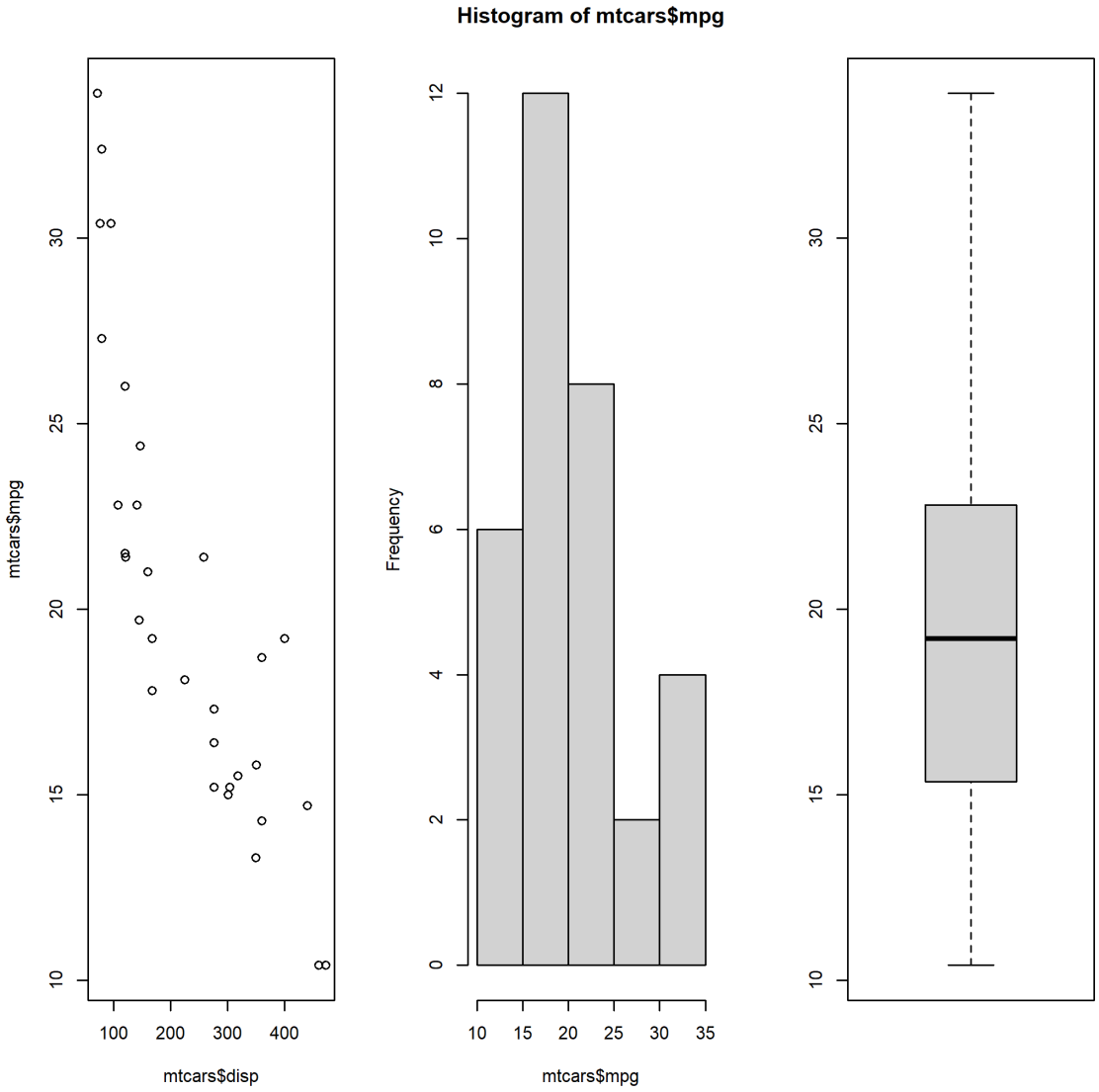



```r
# restore the setting stored in init
par(init)
```

Case Study 4

Combine 3 plots in 1 row and 3 columns.

```r
# store the current parameter settings in init
init <- par(no.readonly=TRUE)

# specify that 3 graphs to be combined and filled by rows
par(mfrow = c(1, 3))

# specify the graphs to be combined
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```

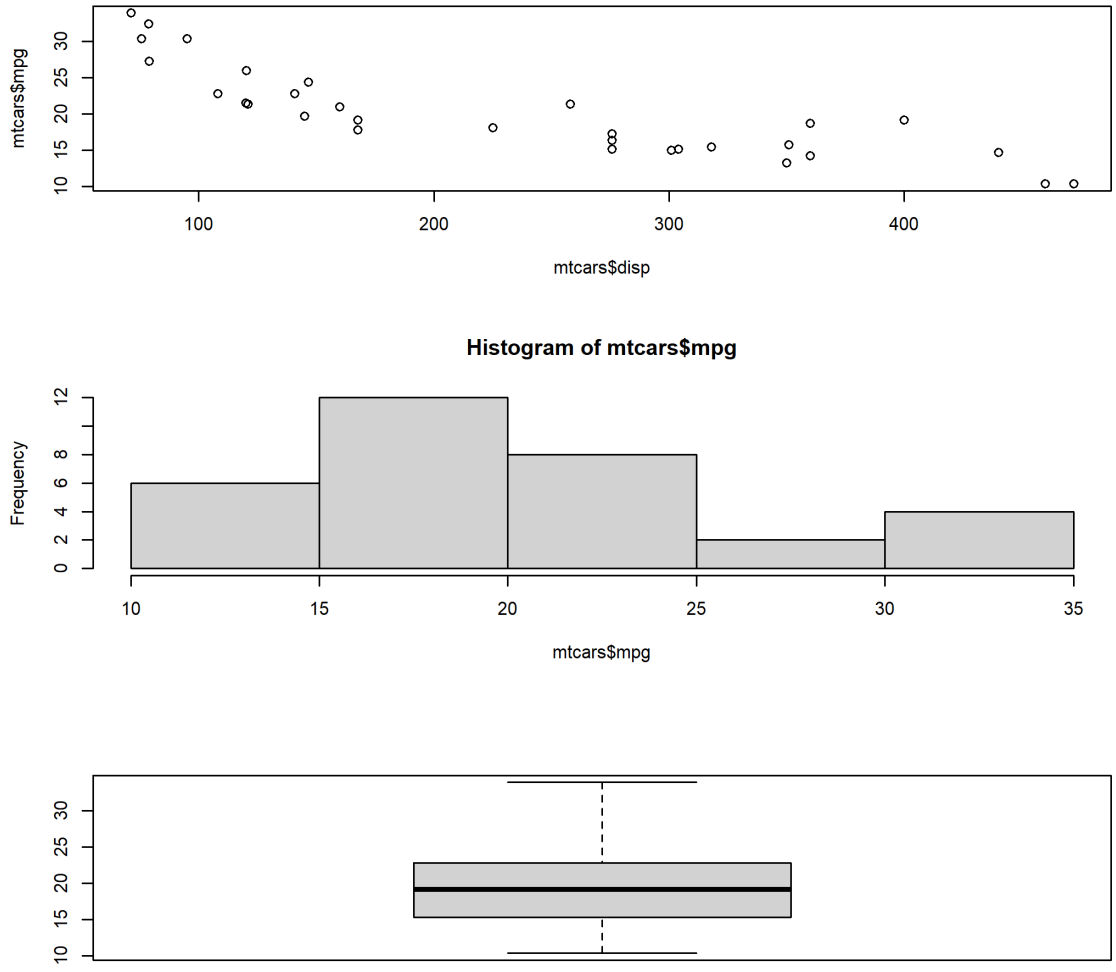



```r
# restore the setting stored in init
par(init)
```

Case Study 5

Combine 3 plots in 3 rows and 1 column.

```
# store the current parameter settings in init
init <- par(no.readonly=TRUE)

# specify that 3 graphs to be combined and filled by rows
par(mfrow = c(3, 1))

# specify the graphs to be combined
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```



```
# restore the setting stored in init
par(init)
```

Column

mfcol combines plots filled by columns i.e it takes two arguments, the number of rows and number of columns and then starts filling the plots by columns. Below is the syntax for mfcol:

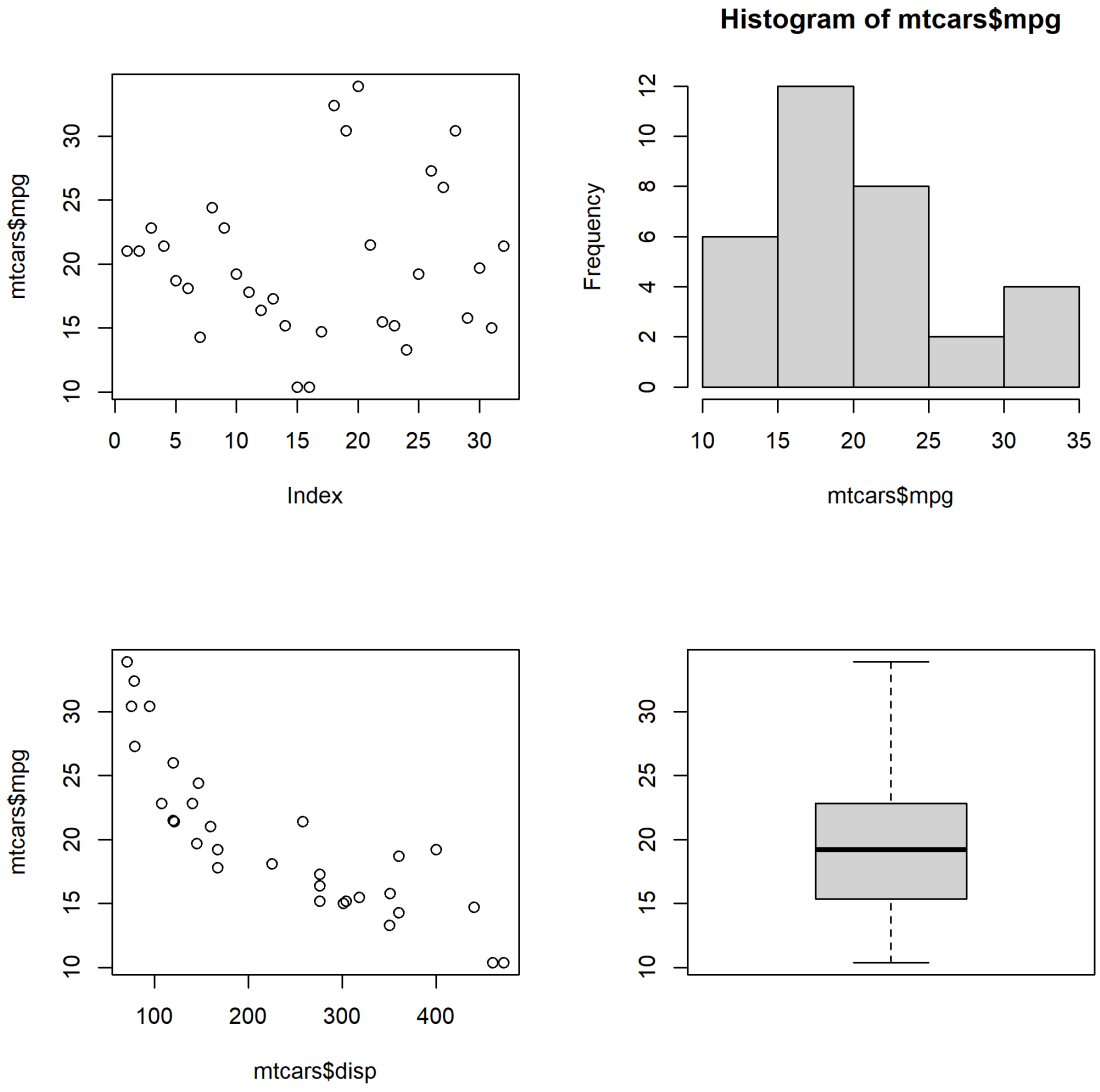
```
# mfcol syntax
par(mfcol = c(number of rows, number of columns))
```

Let us begin by combining 4 plots in 2 rows and 2 columns:

Case Study 6

Combine 4 plots in 2 rows and 2 columns, filled by columns.

```
# store the current parameter settings in init
init <- par(no.readonly=TRUE)

# specify that 4 graphs to be combined and filled by columns
par(mfcol = c(2, 2))

# specify the graphs to be combined
plot(mtcars$mpg)
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```




```
# restore the setting stored in init
par(init)
```

Special Cases

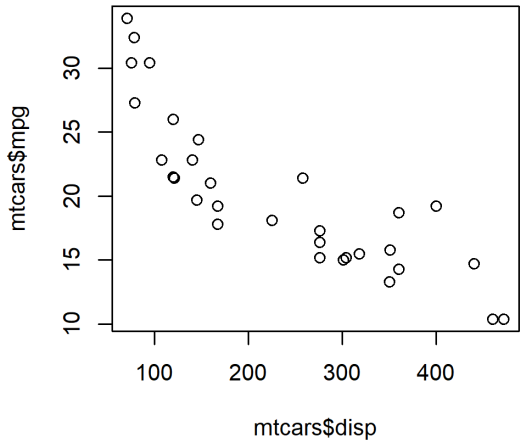
What happens if we specify lesser or more number of graphs? In the next two examples, we will specify lesser or more number of graphs than we ask the par() function to combine. Let us see what happens in such instances:

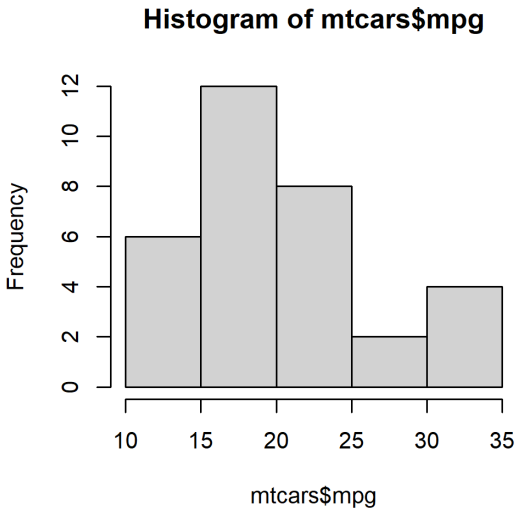
Case 1: Lesser number of graphs specified We will specify that 4 plots need to be combined in 2 rows and 2 columns but provide only 3 graphs.

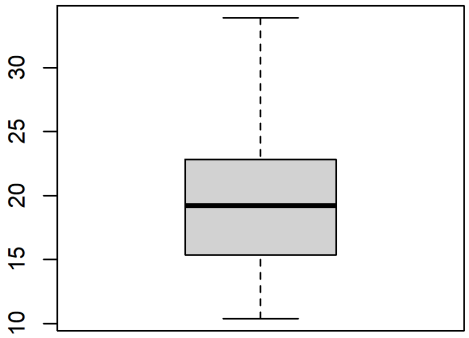
Case 2: Extra graph specified We will specify that 4 plots need to be combined in 2 rows and 2 columns but specify 6 graphs instead of 4.

Case Study 7

```
# store the current parameter settings in init
init <- par(no.readonly=TRUE)

# specify that 4 graphs to be combined and filled by rows
par(mfrow = c(2, 2))

# specify the graphs to be combined
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)

# restore the setting stored in init
par(init)
```

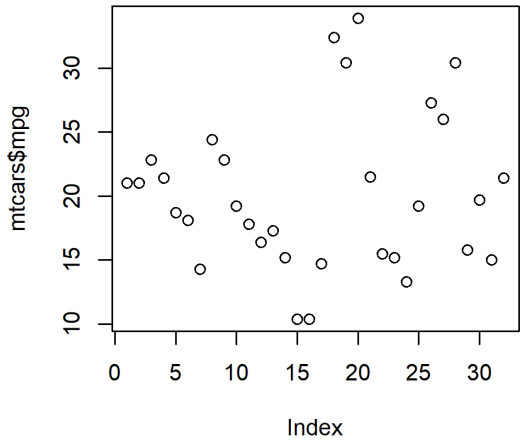


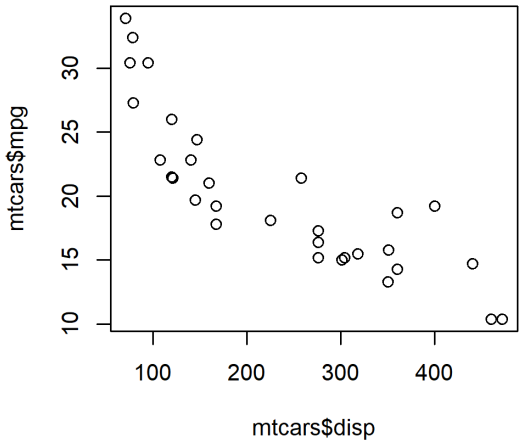


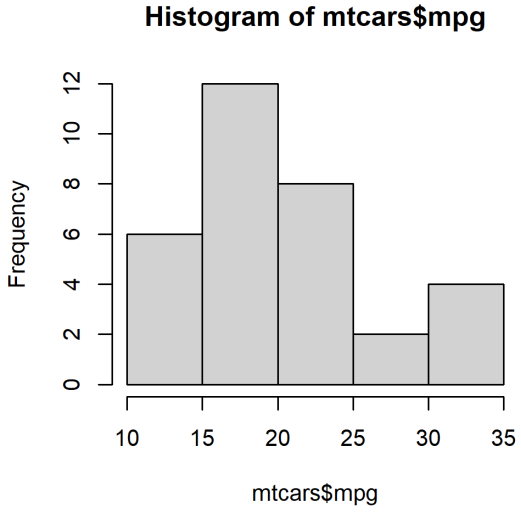


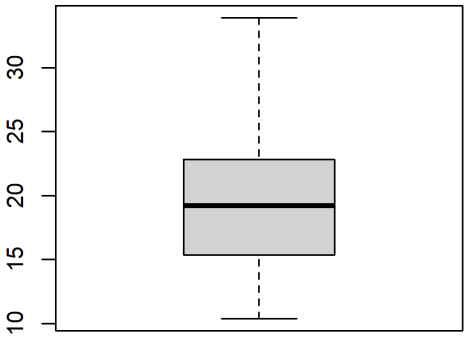
Case Study 8

```
# store the current parameter settings in init
init <- par(no.readonly=TRUE)

# specify that 4 graphs to be combined and filled by rows
par(mfrow = c(2, 2))

# specify the graphs to be combined
plot(mtcars$mpg)
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```

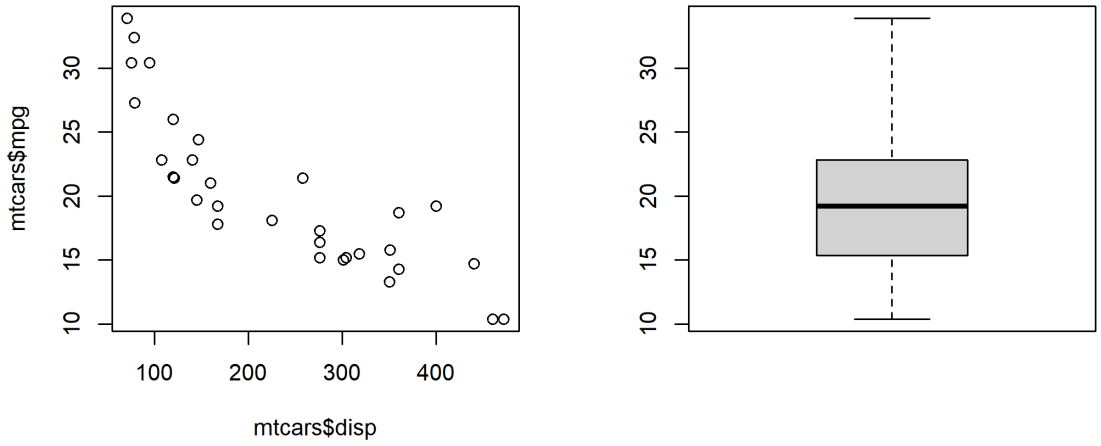








```r
plot(mtcars$disp, mtcars$mpg)
boxplot(mtcars$mpg)

# restore the setting stored in init
par(init)
```



Layout

At the core of the layout() function is a matrix. We communicate the structure in which the plots must be combined using a matrix. As such, the layout function is more flexible compared to the par() function.

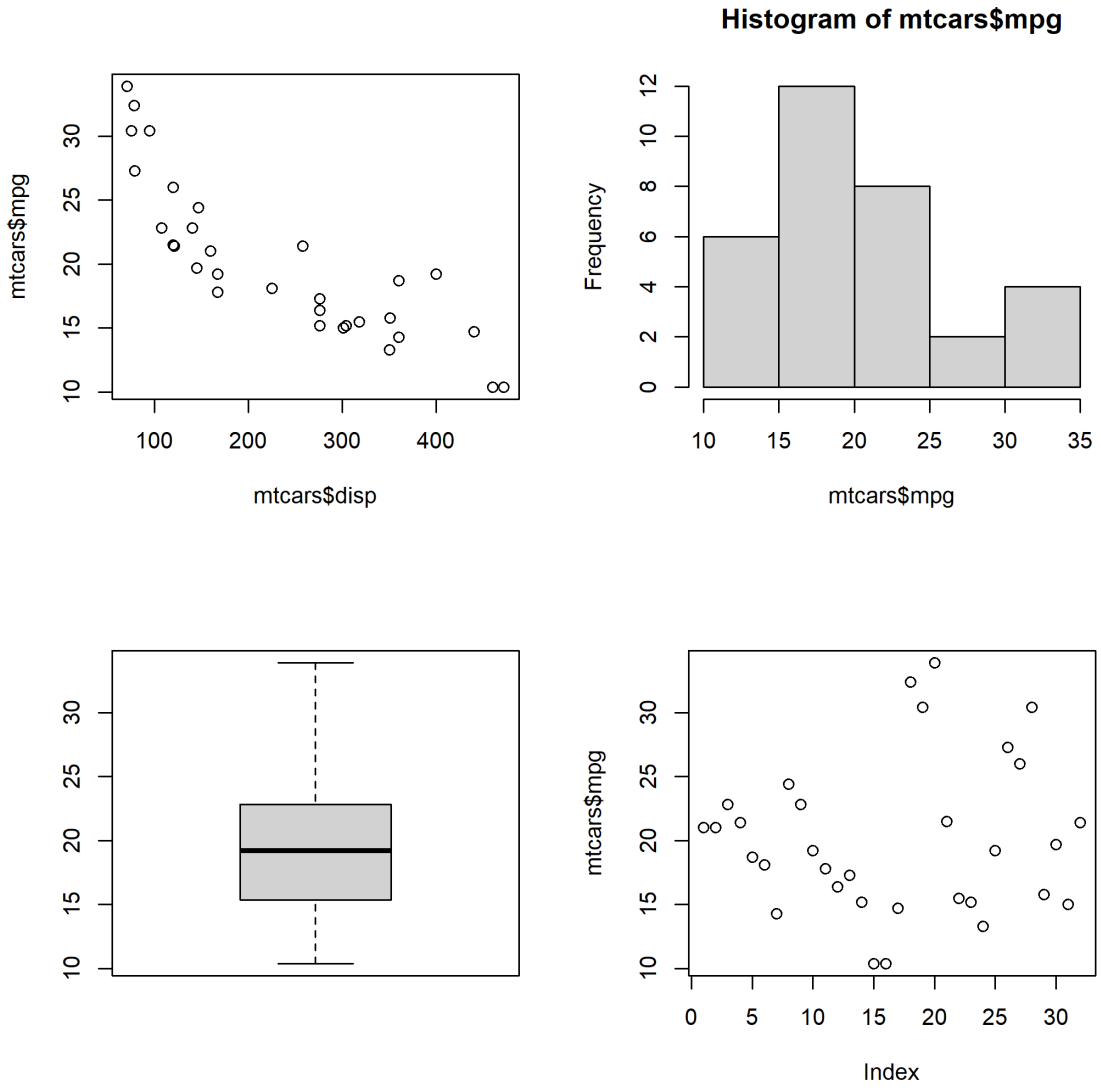
Option	Description	Value
matrix	matrix specifying location of plots	matrix
widths	width of columns	vector
heights	height of rows	vector

Let us begin by combining 4 plots in a 2 row/2 column structure. We do this by creating a layout using the matrix function.

Case Study 1

Combine 4 plots in 2 rows/2 columns filled by rows.

```
# specify the layout
# 4 plots to be combined in 2 row/ 2 columns and arranged by row
layout(matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE))

# specify the 4 plots
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
plot(mtcars$mpg)
```




```
layout(1)
```

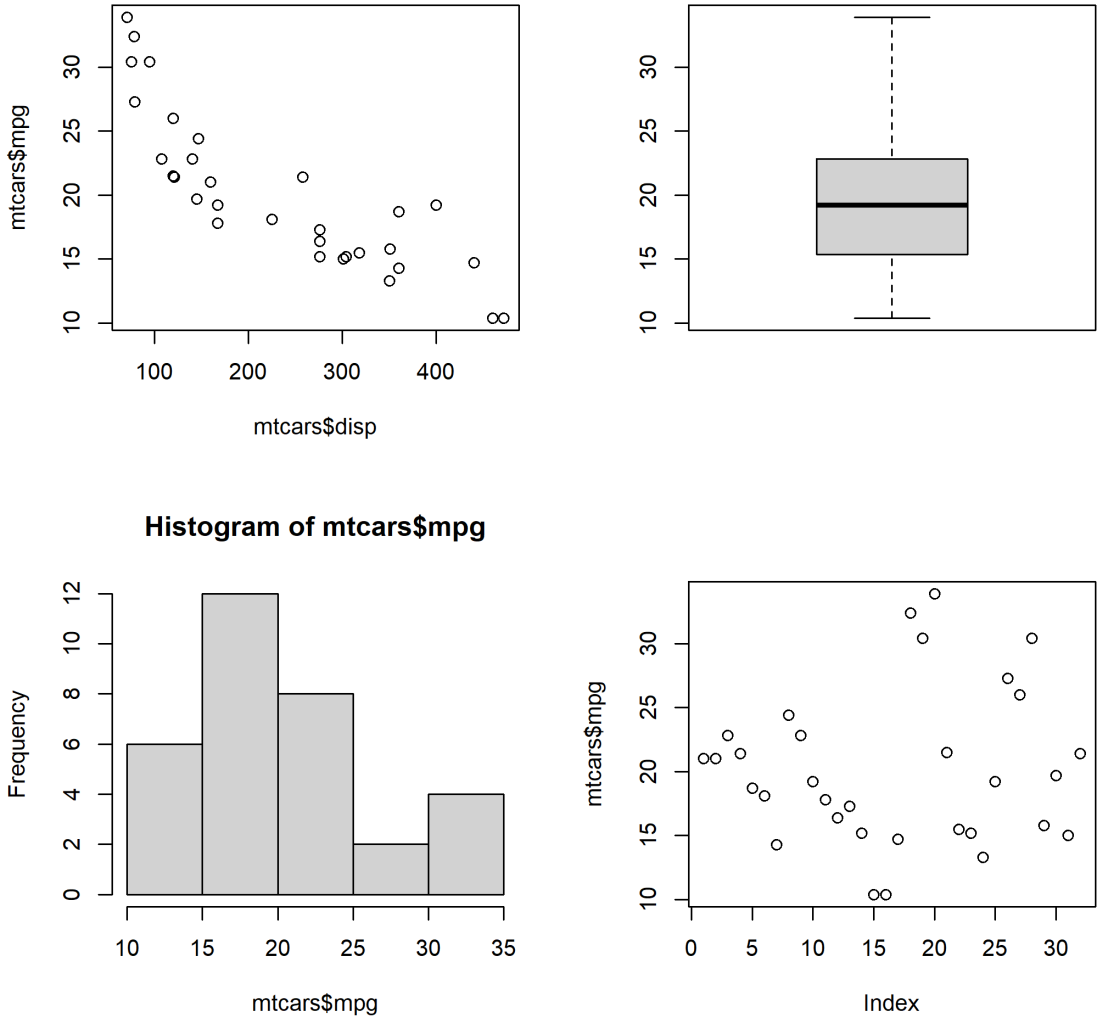
Case Study 2

Combine 4 plots in 2 rows/2 columns filled by columns

To fill the plots by column, we specify byrow = FALSE in the matrix.

```
# specify the layout
# 4 plots to be combined in 2 row/ 2 columns and filled by columns
layout(matrix(c(1, 2, 3, 4), nrow = 2, byrow = FALSE))

# specify the 4 plots
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
```

```r
boxplot(mtcars$mpg)
plot(mtcars$mpg)
```



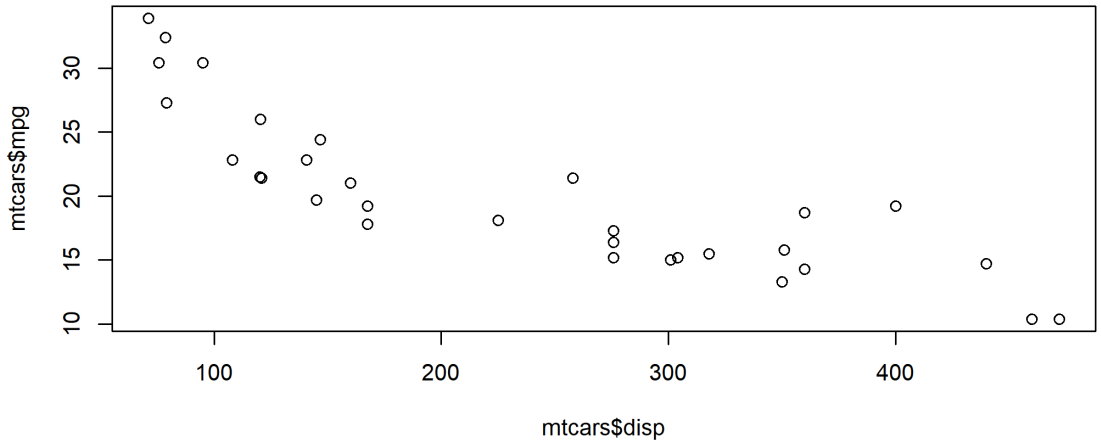

```r
layout(1)
```

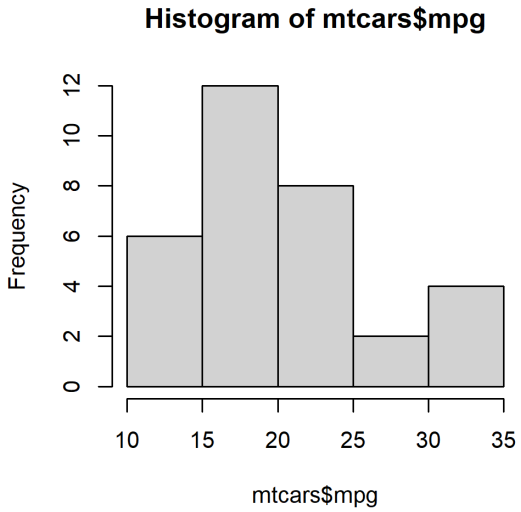
Case Study 3

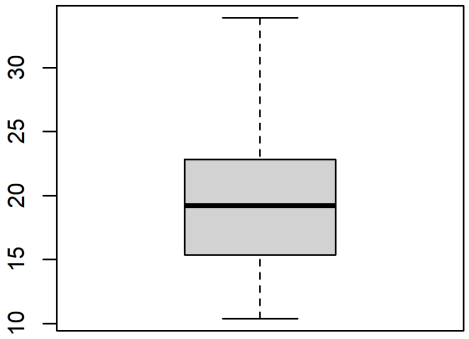
Combine 3 plots in 2 rows/2 columns filled by rows

The magic of the layout() function begins here. We want to combine 3 plots and the first plot should occupy both the columns in row 1 and the next 2 plots should be in row 2. If you look at the matrix below, 1 is specified twice and since the matrix is filled by row, it will occupy both the columns in the first row. Similarly the first plot will occupy the entire first row. It will be crystal clear when you see the plot.

```r
# specify the matrix
matrix(c(1, 1, 2, 3), nrow = 2, byrow = TRUE)
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    3

# 3 plots to be combined in 2 row/ 2 columns and arranged by row
layout(matrix(c(1, 1, 2, 3), nrow = 2, byrow = TRUE))

# specify the 3 plots
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```







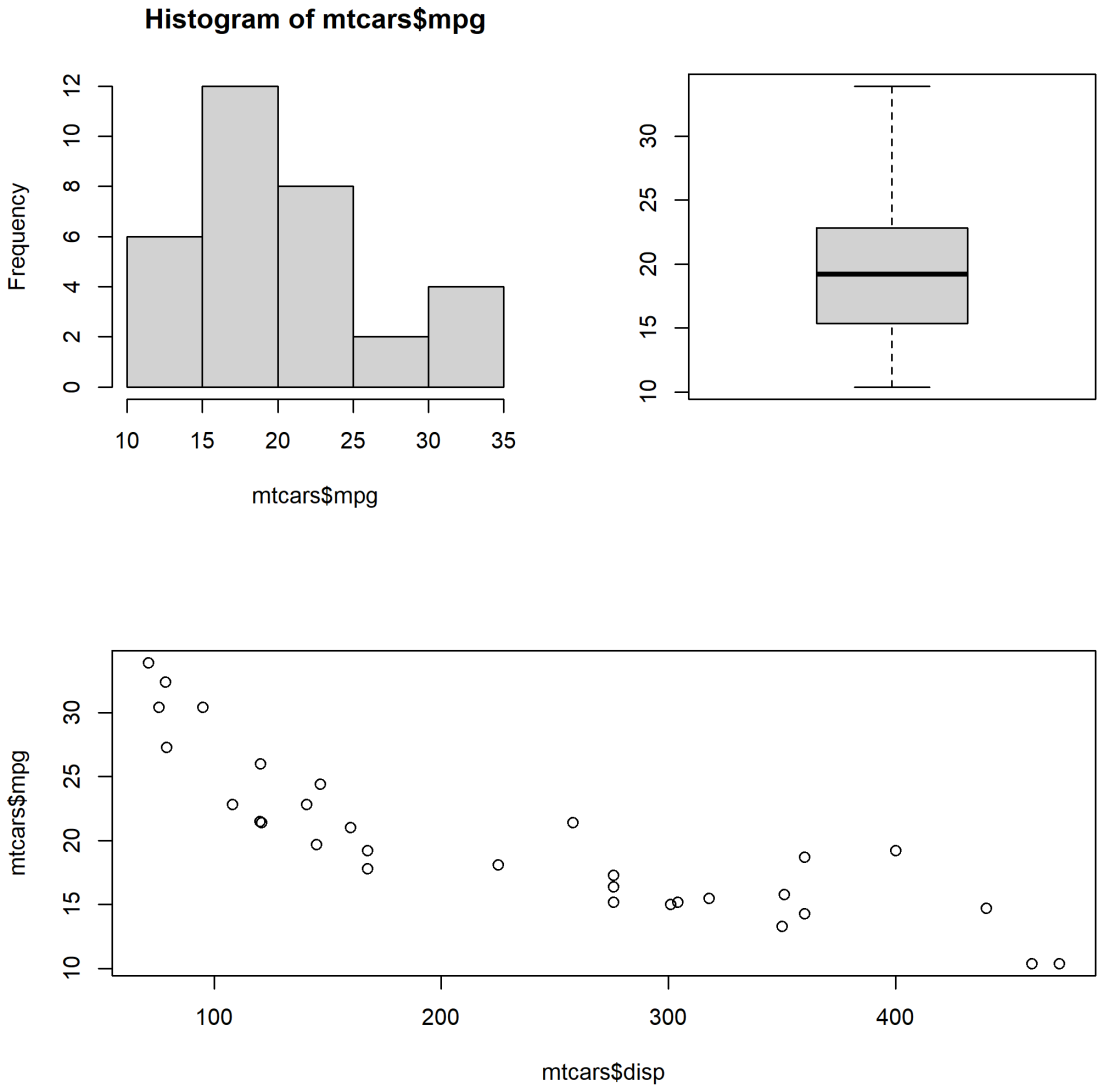
```
layout(1)
```

Case Study 4

Combine 3 plots in 2 rows/2 columns filled by rows

The plots must be filled by rows and the third plot must occupy both the columns of the second row while the other two plots will be placed in the first row. The matrix would look like this:

```
# specify the matrix
matrix(c(1, 2, 3, 3), nrow = 2, byrow = TRUE)
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    3

# 3 plots to be combined in 2 row/ 2 columns and arranged by row
layout(matrix(c(1, 2, 3, 3), nrow = 2, byrow = TRUE))

# specify the 3 plots
hist(mtcars$mpg)
boxplot(mtcars$mpg)
plot(mtcars$disp, mtcars$mpg)
```




```
layout(1)
```

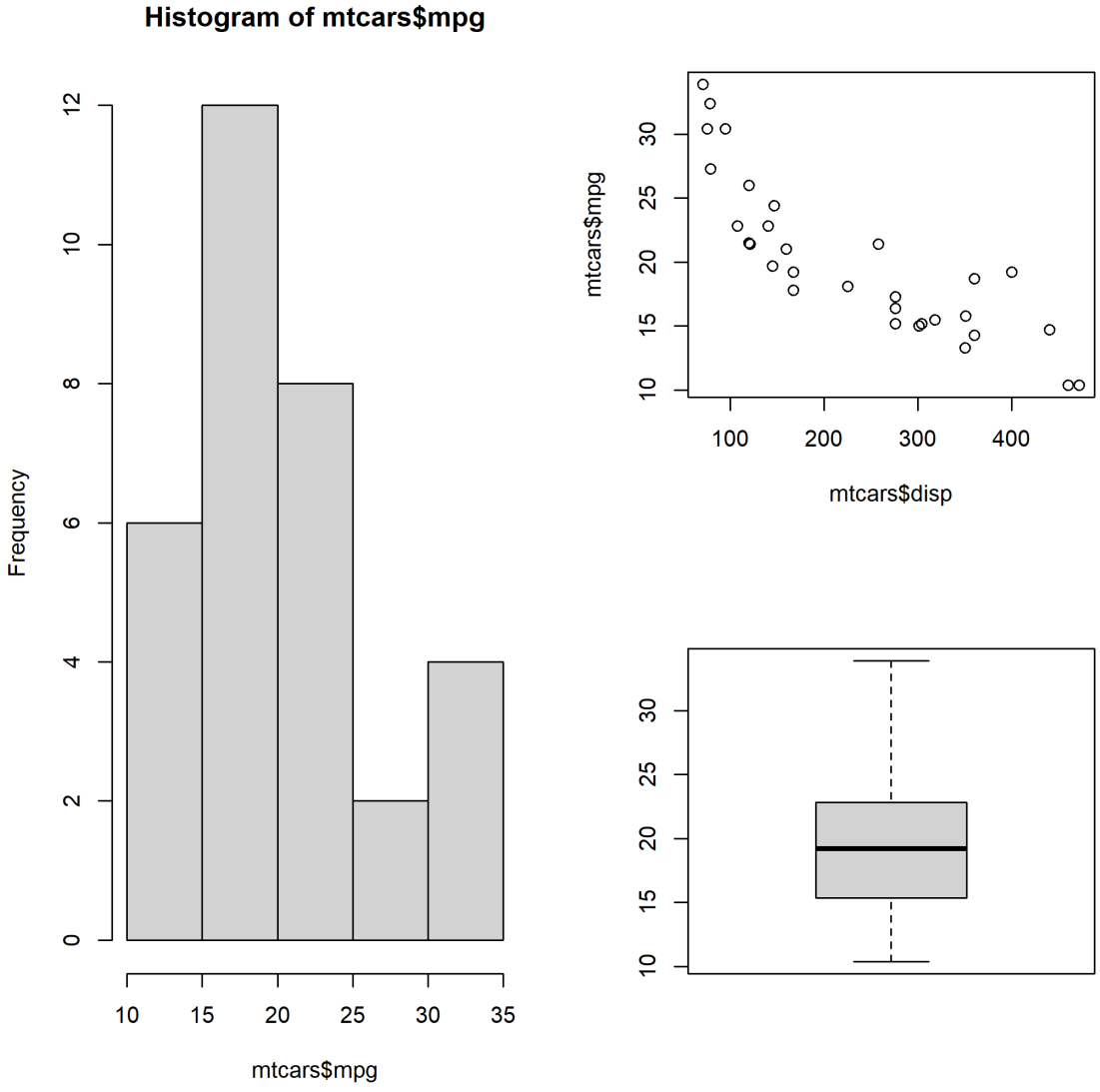
Case Study 5

Combine 3 plots in 2 rows/2 columns filled by columns

The plots must be filled by columns and the first plot must occupy both the rows of the first column while the other two plots will be placed in the second column in two rows. The matrix would look like this:

```
# specify the matrix
matrix(c(1, 1, 2, 3), nrow = 2, byrow = FALSE)
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    3
```

```r
# 3 plots to be combined in 2 row/ 2 columns and arranged by columns
layout(matrix(c(1, 1, 2, 3), nrow = 2, byrow = FALSE))

# specify the 3 plots
hist(mtcars$mpg)
plot(mtcars$disp, mtcars$mpg)
boxplot(mtcars$mpg)
```



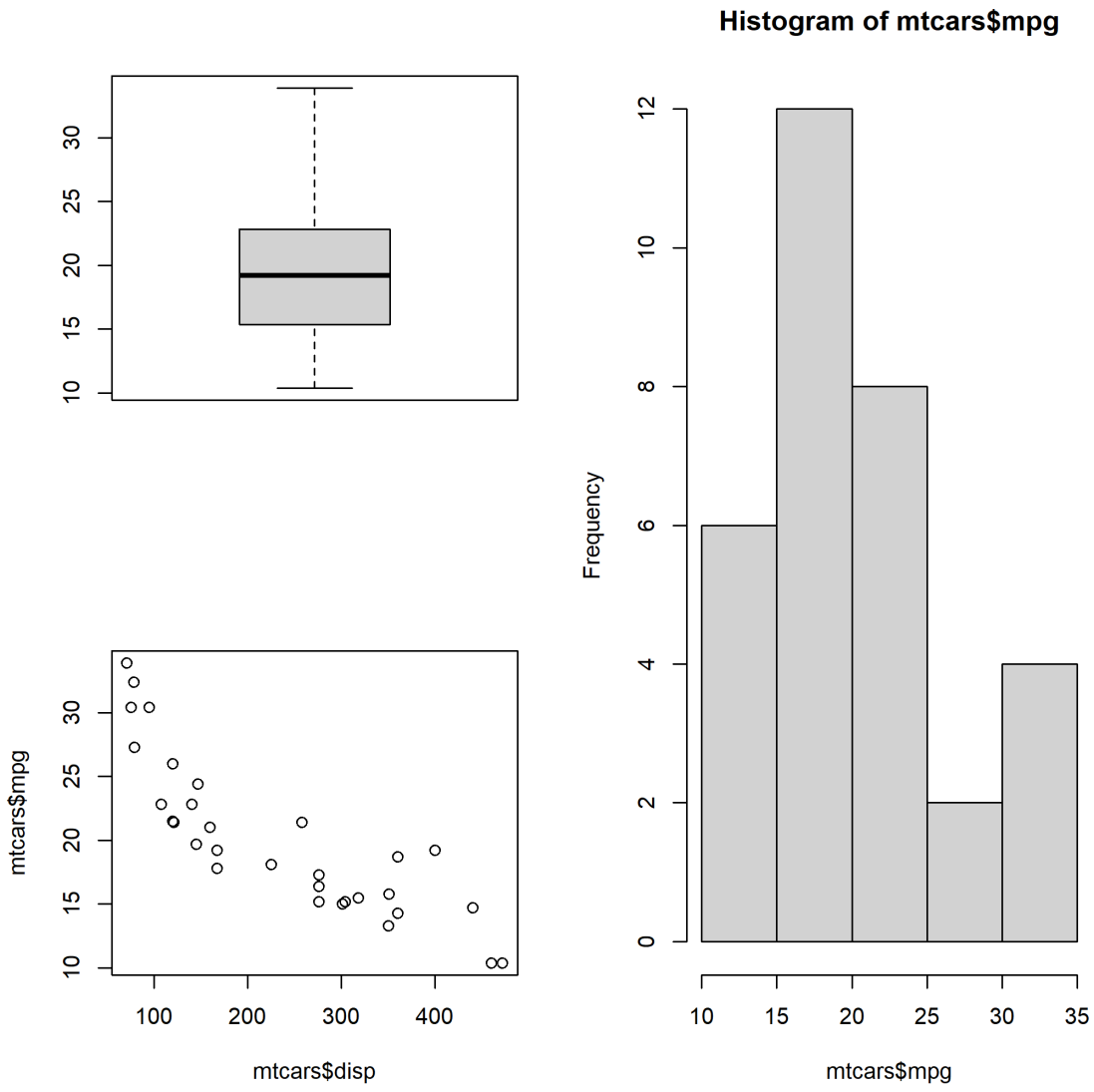

```r
layout(1)
```

Case Study 6

Combine 3 plots in 2 rows/2 columns filled by columns

The plots must be filled by columns and the first plot must occupy both the rows of the second column while the other two plots will be placed in the first column in two rows. The matrix would look like this:

```
# specify the matrix
matrix(c(1, 2, 3, 3), nrow = 2, byrow = FALSE)
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    3

# 3 plots to be combined in 2 row/ 2 columns and arranged by columns
layout(matrix(c(1, 2, 3, 3), nrow = 2, byrow = FALSE))

# specify the 3 plots
boxplot(mtcars$mpg)
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
```




```
layout(1)
```

In all the layouts created so far, we have kept the size of the rows and columns equal. What if you want to modify the width and height of the columns and rows? The widths and heights arguments in the layout() function address the above mentioned issue. Let us check them out one by one: The widths argument is used for specifying the width of the columns. Based on the number of columns in the layout, you can specify the width of each column. Let us look at some examples.

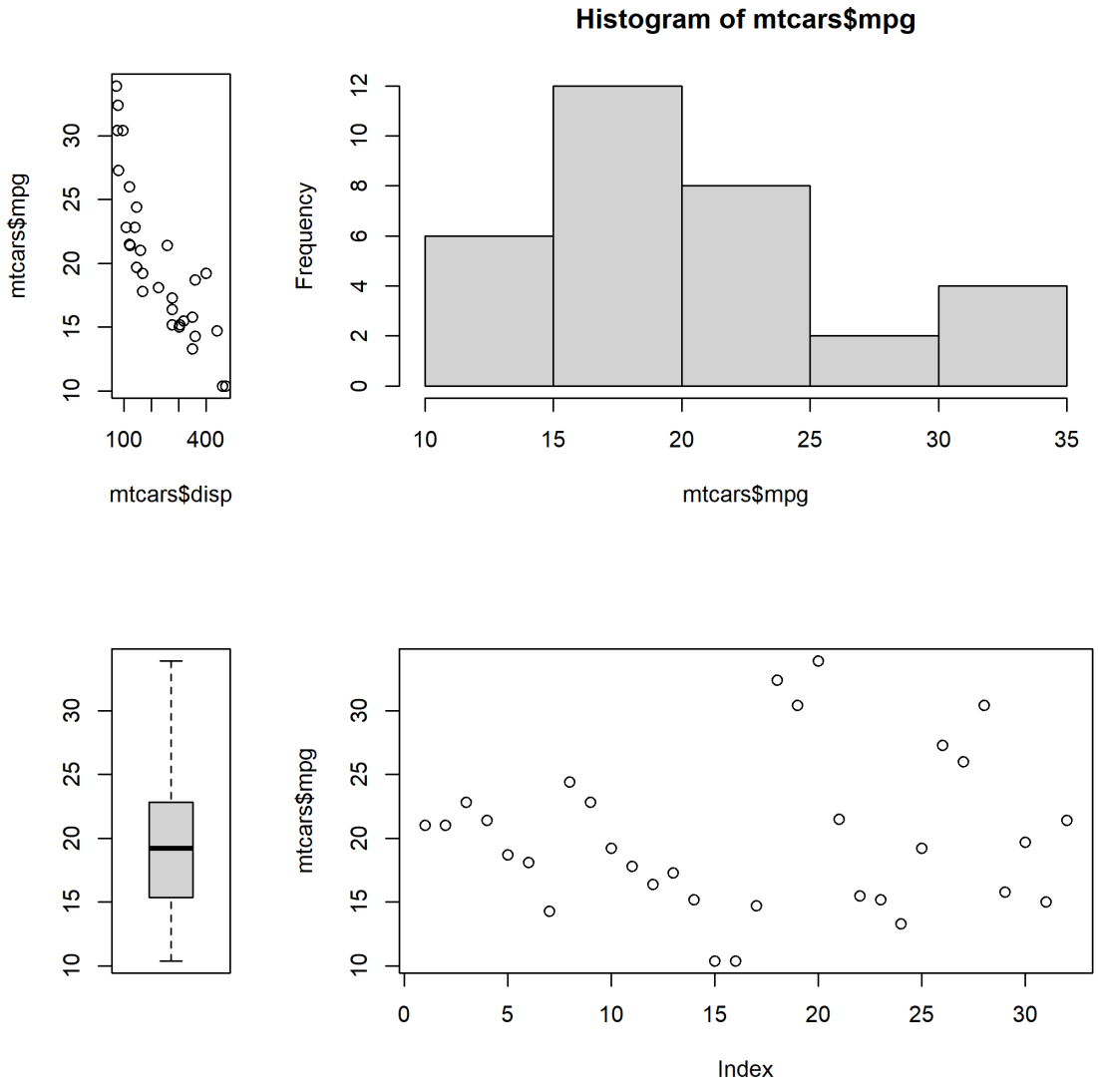
Case Study 7

Width of the 2nd column is twice the width of the 1st column

```
# specify the matrix
matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE)
```

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

```r
# 4 plots to be combined in 2 row/ 2 columns and arranged by columns
layout(matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE), widths = c(1, 3))

# specify the plots
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
plot(mtcars$mpg)
```

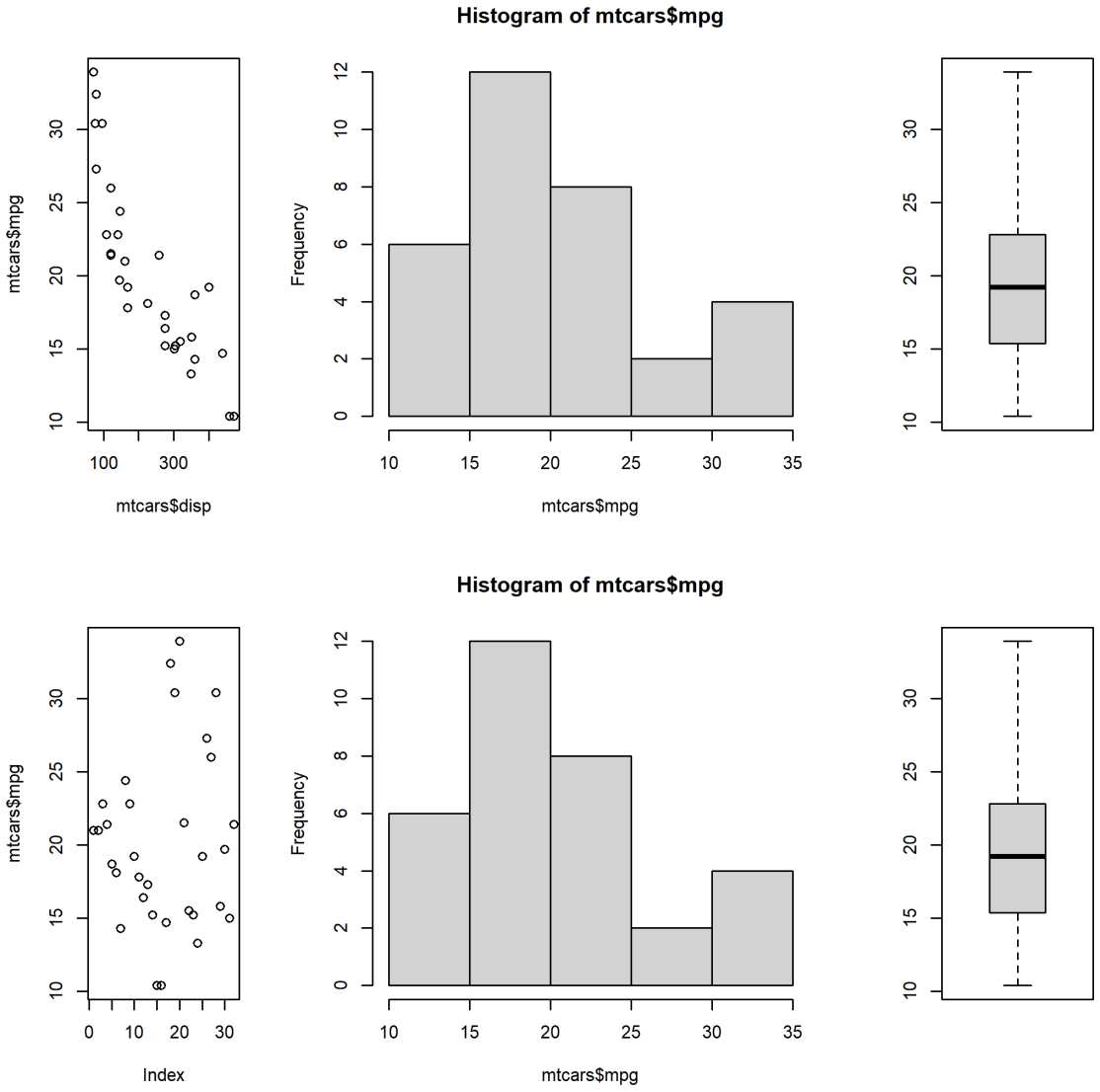



```r
layout(1)
```

Case Study 8

Width of the 2nd column is twice that of the first and last column

```
# specify the matrix
matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, byrow = TRUE)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6

# 6 plots to be combined in 2 row/ 3 columns and filled by rows
layout(matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, byrow = TRUE), widths = c(1, 2, 1))

# specify the plots
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
plot(mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```




```
layout(1)
```

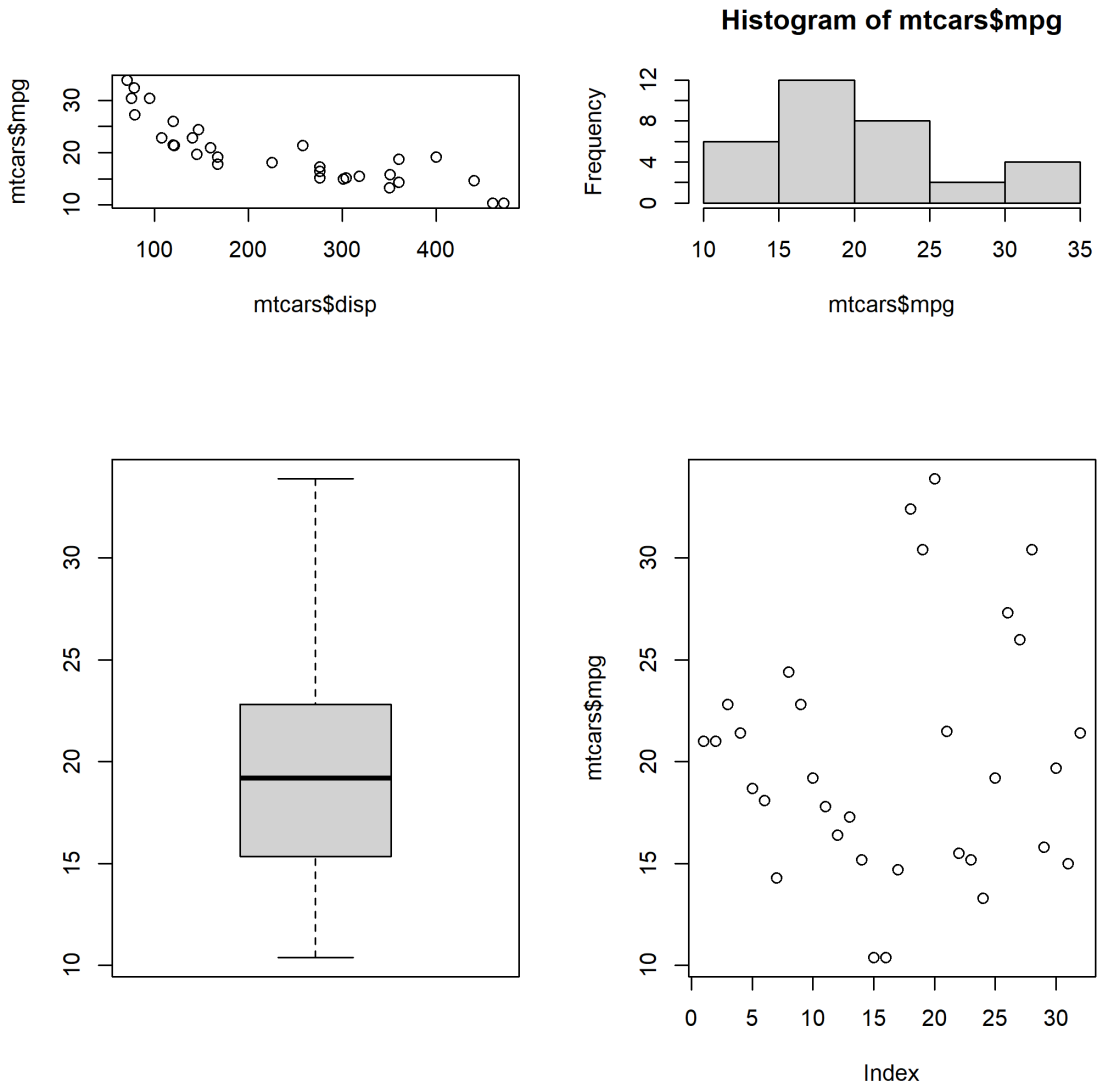
The heights arguments is used to modify the height of the rows and based on the number of rows specified in the layout, we can specify the height of each row.

Case Study 9

Height of the 2nd row is twice that of the first row

```
# 4 plots to be combined in 2 row/ 2 columns and filled by rows
layout(matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE), heights= c(1, 2))

# specify the 4 plots
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
```

```r
boxplot(mtcars$mpg)
plot(mtcars$mpg)
```



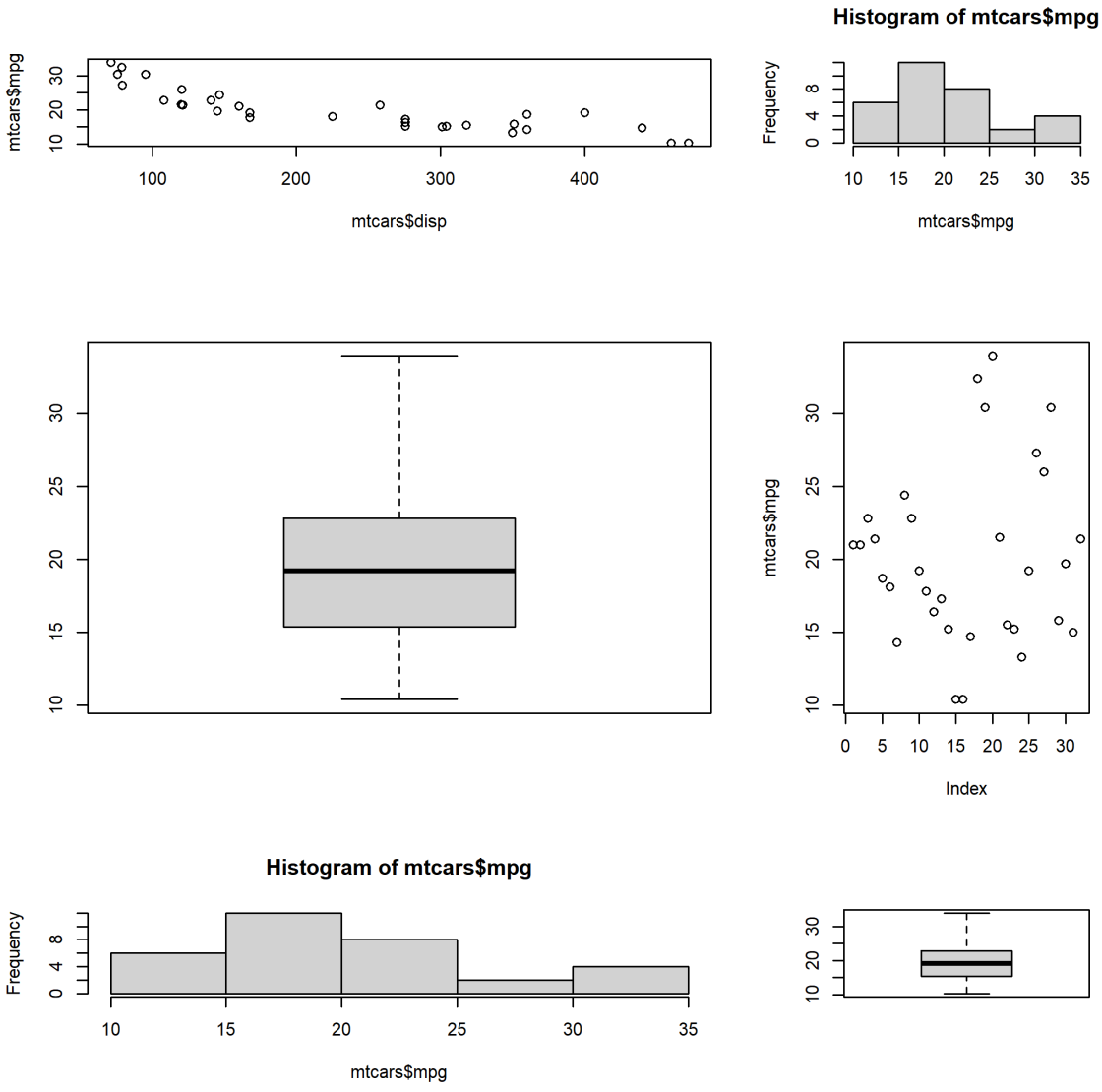

```r
layout(1)
```

Putting it all together...

Before we end this section, let us combine plots using both the widths and heights option

```r
# specify the matrix
matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, byrow = TRUE)
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

```
# 6 plots to be combined in 3 row/ 2 columns and arranged by rows
layout(matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, byrow = TRUE), heights= c(1, 2, 1),
widths = c(2, 1))

# specify the 6 plots
plot(mtcars$disp, mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
plot(mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
```




```
layout(1)
```

About the Author

Aravind Hebbali is the founder of Rsquared Academy. He earned his Masters in Economics from Madras School of Economics. As an active R user, he has authored several R packages such as

- olsrr
- rfm
- descriptr
- blorr
- xplorerr

In 2015, he founded Rsquared Academy, a free and open source education initiative with focus on data science and analytics. Apart from self paced online courses, Rsquared Academy offers customized learning modules for corporates and universities.

You can find him on GitHub.