2.3 Lab: Introduction to R

Basic Commands

To create a vector of numbers, we use the function `c()` (for *concatenate* or *combine*). Any numbers inside the parentheses are joined together. The following command saves the vector to the variable `x`.

```
x <- c(1, 3, 2, 5)
x
```

```
[1] 1 3 2 5
```

We can also save things using `=` rather than `<-`:

```
x = c(1, 6, 2)
x
```

```
[1] 1 6 2
```

```
y = c(1, 4, 3)
```

We can add the two vectors `x` and `y` together elementwise, but we must ensure that the two vectors are of the same length. We can check this using the `length()` function.

```
length(x)
```

```
[1] 3
```

```
length(y)
```

```
[1] 3
```

```
x + y
```

```
[1]  2 10  5
```

The `ls()` function lists all of the objects that we have saved so far. The `rm()` function can be used to delete any of them.

```
ls()
```

```
[1] "x" "y"
```

```
rm(x, y)
ls()
```

```
character(0)
```

It is also possible to remove all objects at once:

```
rm(list=ls())
```

The `matrix()` function can be used to create a matrix of numbers. Like any other function, we can enter a `?` before the function name to see more information.

```
?matrix
```

We create a simple matrix:

```
x = matrix(data=c(1, 2, 3, 4), nrow=2, ncol=2)
x
```

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

We can omit the argument names and instead just type

```
x = matrix(c(1, 2, 3, 4), 2, 2)
```

and this would have the same effect. We can see that by default `R` creates matrices by successively filling in columns. The parameter `byrow=TRUE` can be used to populate the matrix in order of the rows.

```
matrix(c(1, 2, 3, 4), 2, 2, byrow=TRUE)
```

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

The `sqrt()` function returns the square root of each element of a vector or matrix. The command `x^2` raises each element of `x` to the power 2.

```
sqrt(x)
```

```
         [,1]     [,2]
[1,] 1.000000 1.732051
[2,] 1.414214 2.000000
```

```
x^2
```

```
     [,1] [,2]
[1,]    1    9
[2,]    4   16
```

The `rnorm()` function generates standard normal random variables with mean 0 and standard deviation 1. These values can be altered using the `mean` and `sd` arguments. We use the `cor()` function to compute the correlation between them.

```
x = rnorm(50)
y = x + rnorm(50, mean=50, sd=0.1)
cor(x, y)
```

```
[1] 0.9917085
```

Sometimes we want our code to reproduce the exact same set of random numbers; we can use the `set.seed()` function to do this. This function takes an integer argument.

```
set.seed(1303)
rnorm(10)
```

```
 [1] -1.1439763145  1.3421293656  2.1853904757  0.5363925179  0.0631929665
 [6]  0.5022344825 -0.0004167247  0.5658198405 -0.5725226890 -1.1102250073
```

The `mean()` and `var()` functions can be used to compute the mean and variance of a vector of numbers. Applying `sqrt()` to the output of `var()` will give the standard deviation. Alternatively, we can use the `sd` function.

```
set.seed(3)
y = rnorm(100)
mean(y)
```

```
[1] 0.01103557
```

```
var(y)
```

```
[1] 0.7328675
```

```
sqrt(var(y))
```

```
[1] 0.8560768
```

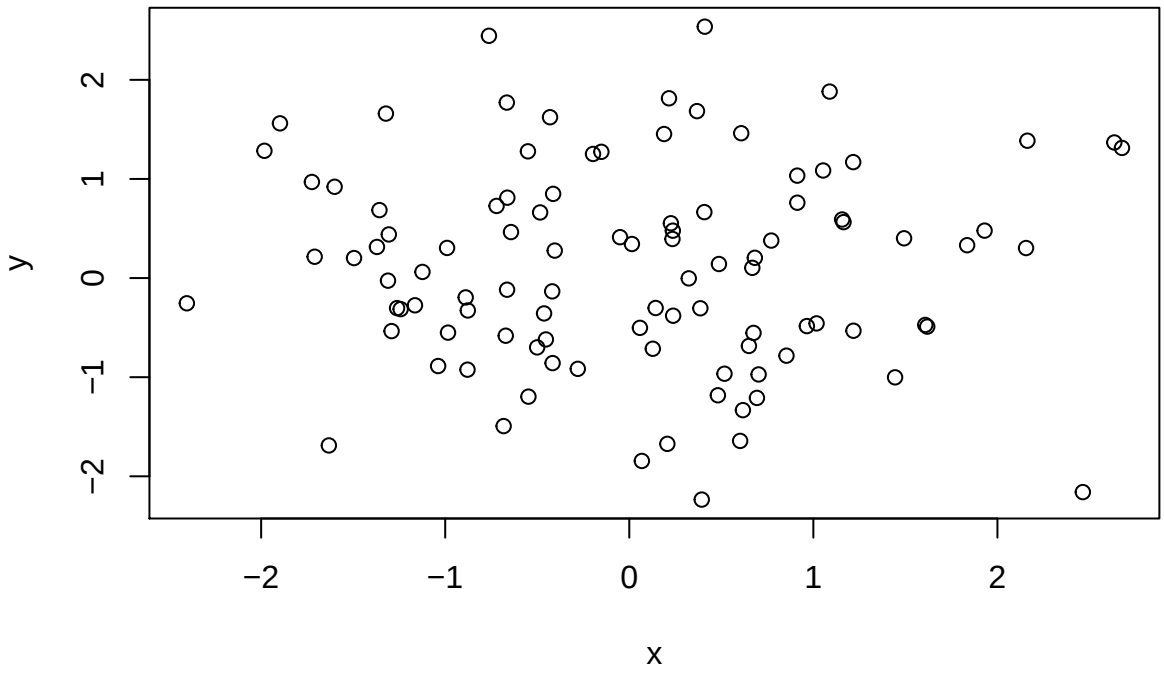
```
sd(y)
```

```
[1] 0.8560768
```

Graphics

The `plot()` function is the primary way to plot data in `R`. For instance, `plot(x, y)` produces a scatterplot of the numbers in `x` versus the numbers in `y`. There are many additional options that can be passed in to the `plot()` function. For example, passing in the argument `xlab` will result in a label on the x-axis.

```
x = rnorm(100)
y = rnorm(100)
plot(x, y)
```




```
plot(x, y, xlab="this is the x-axis", ylab="this is the y-axis", main="Plot of X vs Y", col="red")
```

We will often want to save the output of an `R` plot. The command that we use to do this will depend on the file type that we would like to create. For instance, to create a pdf, we use the `pdf()` function, and to create a jpeg, we use the `jpeg()` function.

```
pdf("Figure.pdf")
plot(x, y, col="green")
dev.off()
```

The function `dev.off()` indicates to `R` that we are done creating the plot.

The function `seq()` can be used to create a sequence of numbers. For instance, `seq(a, b)` makes a vector of integers between `a` and `b`. There are many other options: for instance, `seq(0, 1, length=10)` makes a sequence of `10` numbers that are equally spaced between `0` and `1`. Typing `3:11` is a shorthand for `seq(3, 11)` for integer arguments.

```
x = seq(1, 10)
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
x = 1:10
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

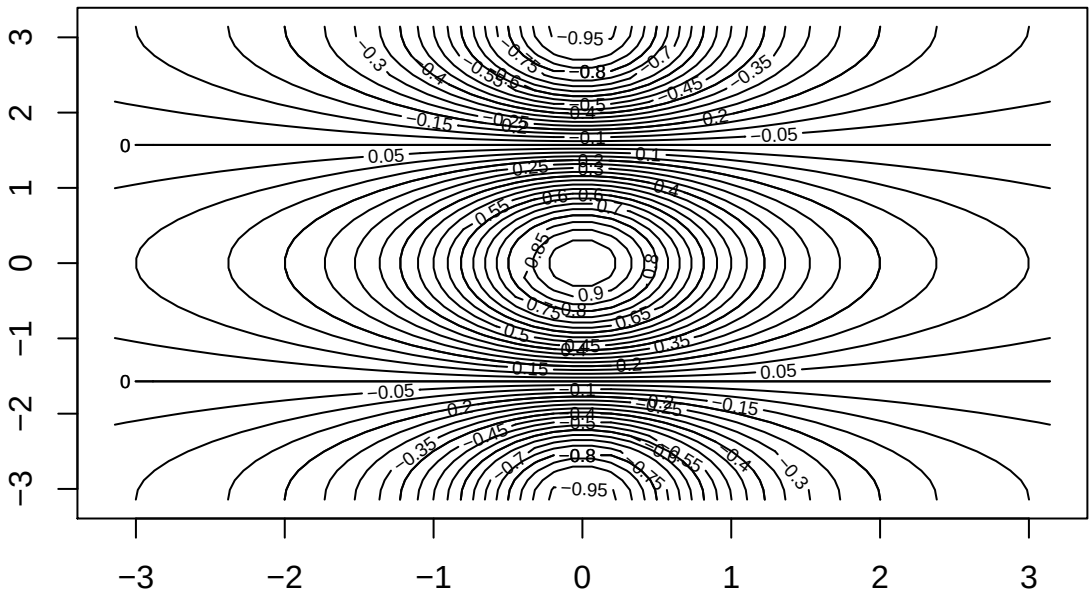
```
x = seq(-pi, pi, length=50)
```

We will now create some more sophisticated plots. The `contour()` function produces a *contour plot* in order to represent three-dimensional data; it is like a topographical map. It takes three arguments:

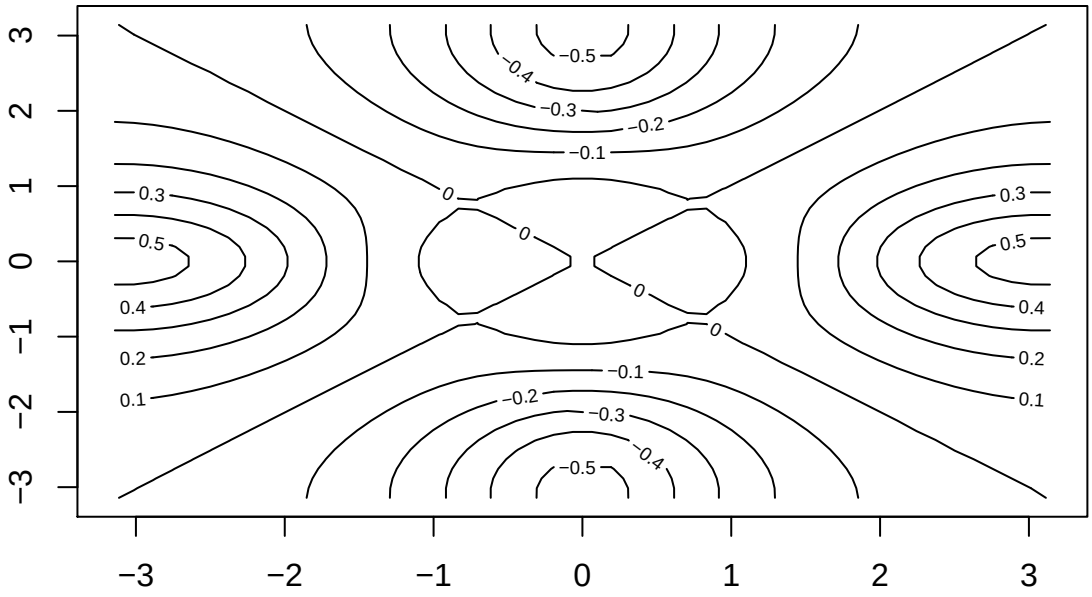
1. A vector of the `x` values (the first dimension),
2. A vector of the `y` values (the second dimension), and
3. A matrix whose elements correspond to the `z` value (the third dimension) for each pair of (`x`, `y`) coordinates.

As with the `plot()` function, there are many other inputs that can be used to fine-tune the output of the `contour()` function.

```
y = x
f = outer(x, y, function(x, y)cos(y)/(1 + x^2))
contour(x, y, f)
contour(x, y, f, nlevels=45, add=T)
```

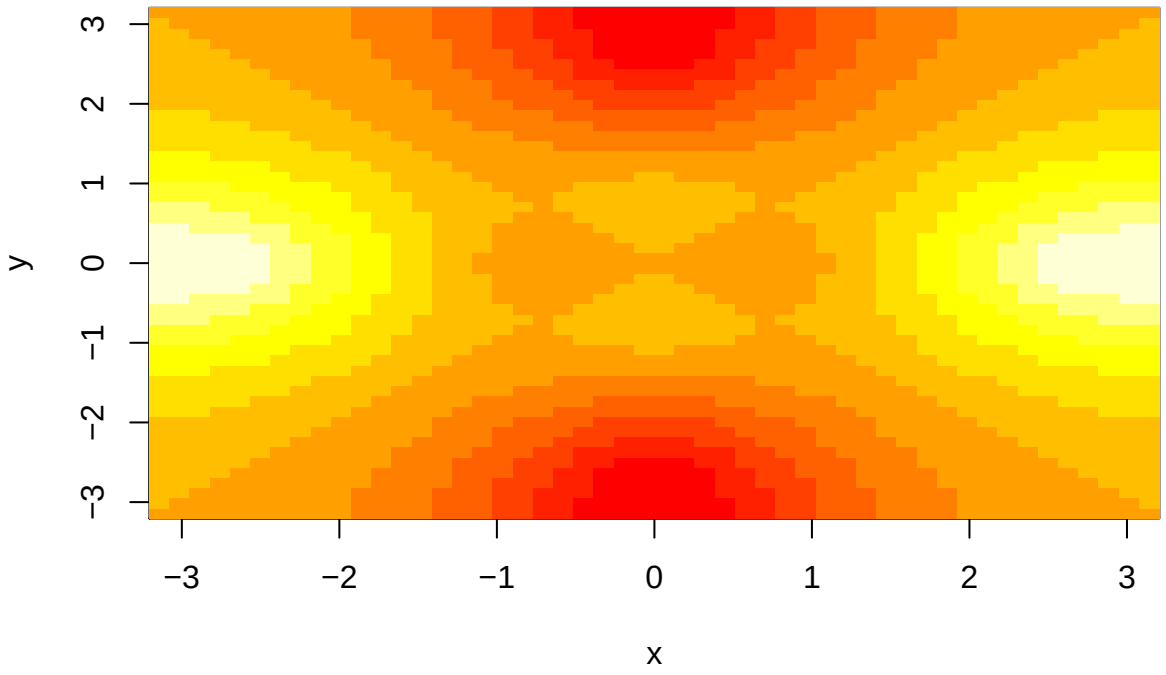


```
fa = (f - t(f))/2
contour(x, y, fa, nlevels=15)
```

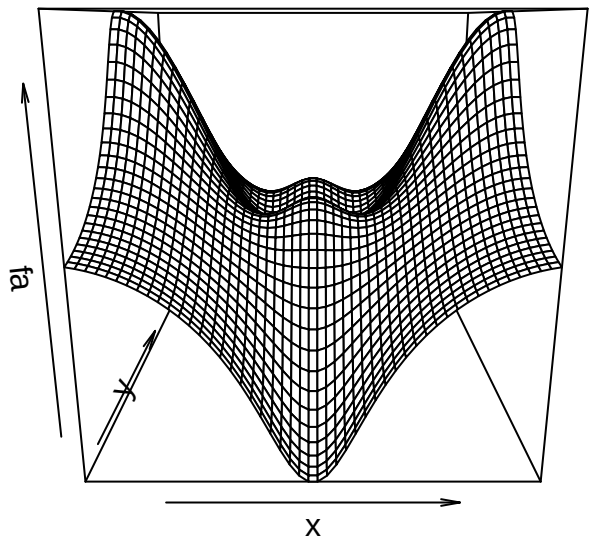


The `image()` function works the same way as `contour()`, except that it produces a color-coded plot whose colors depend on the `z` value. This is known as a *heatmap*, and is sometimes used to plot temperature in weather forecasts. Alternatively, `persp()` can be used to produce a three-dimensional plot. The arguments `theta` and `pi` control the angles at which the plot is viewed.

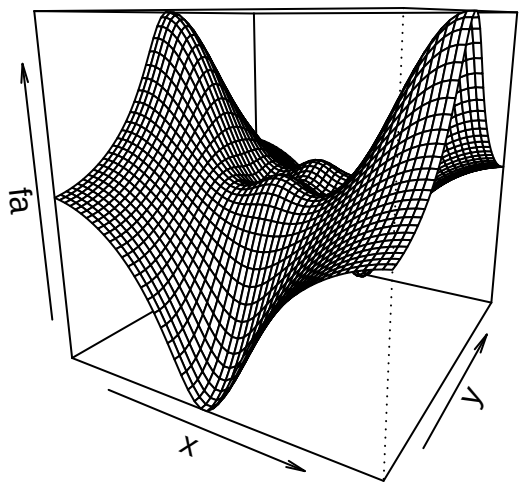
```
image(x, y, fa)
```



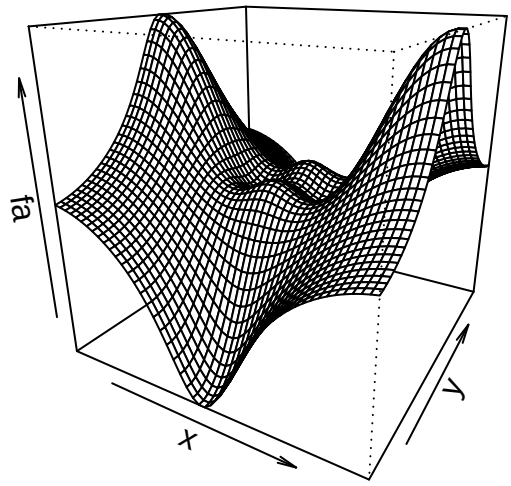

```
persp(x, y, fa)
```



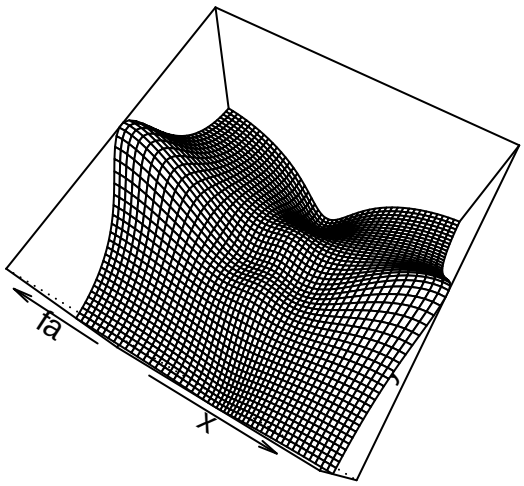

```
persp(x, y, fa, theta=30)
```



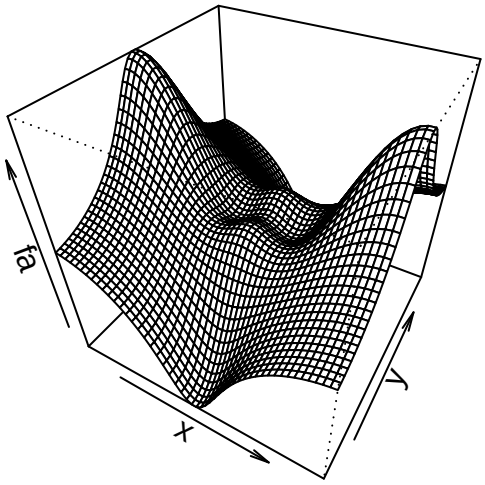

```
persp(x, y, fa, theta=30, phi=20)
```




```
persp(x, y, fa, theta=30, phi=70)
```



```
persp(x, y, fa, theta=30, phi=40)
```



Indexing Data

We often wish to examine part of a set of data. Suppose that our data is stored in the matrix `A`.

```
A = matrix(1:16, 4, 4)
A
```

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

Then, typing

```
A[2, 3]
```

```
[1] 10
```

will select the element corresponding to the second row and the third column. The first number after the `[` always refers to the row, and the second number always refers to the column. We can also select multiple rows and columns at a time, by providing vectors as the indices.

```
A[c(1, 3), c(2, 4)]
```

```
     [,1] [,2]
[1,]    5   13
[2,]    7   15
```

```
A[1:3, 2:4]
```

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

```
A[1:2,]
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
```

```
A[,1:2]
```

```
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

The last two examples include either no index for the columns or no index for the rows. These indicate that `R` should include all columns or all row, respectively. `R` treats a single row or column of a matrix as a vector.

```
A[1,]
```

```
[1]  1  5  9 13
```

The use of a negative sign `-` in the index tells `R` to keep all rows or columns except those indicated in the index.

```
A[-c(1, 3),]
```

```
     [,1] [,2] [,3] [,4]
[1,]    2    6   10   14
[2,]    4    8   12   16
```

The `dim()` function outputs the number of rows followed by the number of columns of a given matrix.

```
dim(A)
```

```
[1] 4 4
```

Loading Data

For most analyses, the first step involves importing a data set in to `R`. The `read.table()` function is one of the primary ways to do this. We can use the function `write.table()` to export data.

We begin by loading the `Auto` data set. This data is part of the `ISLR` library. The following command will load the `Auto.data` file into `R` and store it as an object called `Auto`, in a format referred to as a *data frame*. Once the data has been loaded, the `fix()` function can be used to view it in a spreadsheet like window.

```
Auto = read.table("Auto.data")
fix(Auto)
```

Note that Auto.data is simply a text file, which you can open with a text editor. It is often a good idea to view a data set using a text editor before loading it into R.

This particular data set has not been loaded correctly, because R had assumed that the variables names are part of the data and so had included them in the first row.

```
head(Auto)
```

```
    V1        V2           V3         V4     V5           V6   V7     V8
1  mpg cylinders displacement horsepower weight acceleration year origin
2 18.0         8        307.0      130.0  3504.         12.0   70      1
3 15.0         8        350.0      165.0  3693.         11.5   70      1
4 18.0         8        318.0      150.0  3436.         11.0   70      1
5 16.0         8        304.0      150.0  3433.         12.0   70      1
6 17.0         8        302.0      140.0  3449.         10.5   70      1
                        V9
1                     name
2 chevrolet chevelle malibu
3         buick skylark 320
4        plymouth satellite
5             amc rebel sst
6               ford torino
```

The data set also includes a number of missing observations, indicated by a question mark ?. Missing values are a common occurrence in real data sets. Using the option header=TRUE in the read.table() function tells R that the first line of the file contains the variable names, and using the option na.strings tells R that any time it sees a particular character or set of characters (such as a question mark), it should be treated as a missing element of the data matrix.

```
Auto = read.table("Auto.data", header=TRUE, na.strings="?")
fix(Auto)
```

Excel is a common-format data storage program. An easy way to load such data into R is to save it as a csv (comma separated values) file and then use the read.csv() function to load it in.

```
Auto = read.csv("Auto.csv", header=TRUE, na.strings="?")
dim(Auto)
```

```
[1] 397   9
```

```
Auto[1:4,]
```

```
  mpg cylinders displacement horsepower weight acceleration year origin
1  18         8          307        130   3504         12.0   70      1
2  15         8          350        165   3693         11.5   70      1
3  18         8          318        150   3436         11.0   70      1
4  16         8          304        150   3433         12.0   70      1
                     name
```

```
1 chevrolet chevelle malibu
2         buick skylark 320
3        plymouth satellite
4             amc rebel sst
```

The `dim()` function tells us that the data has 397 observations, or rows, and nine variables, or columns. There are various ways to deal with the missing data. In this case, only five of the rows contains missing observations, and so we choose the `na.omit()` function to simply remove these rows.

```
Auto = na.omit(Auto)
dim(Auto)
```

```
[1] 392   9
```

Once the data are loaded correctly, we can use `names()` to check the variable names.

```
names(Auto)
```

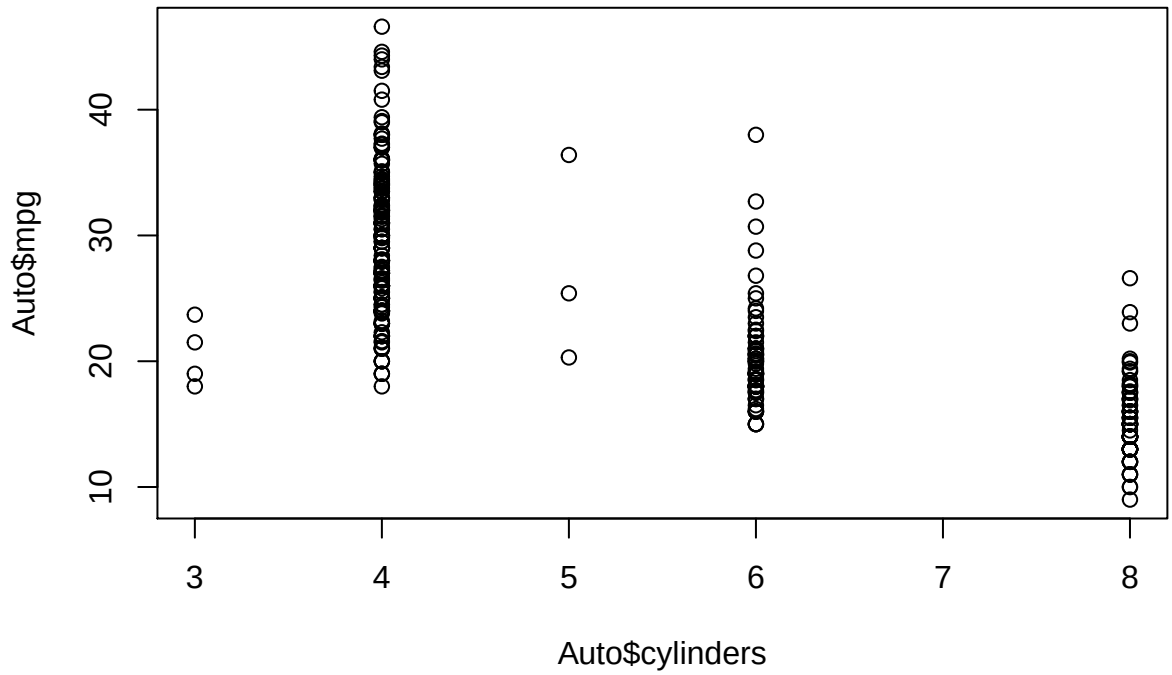
```
[1] "mpg"          "cylinders"    "displacement" "horsepower"
[5] "weight"       "acceleration" "year"         "origin"
[9] "name"
```

Additional Graphical and Numerical Summaries

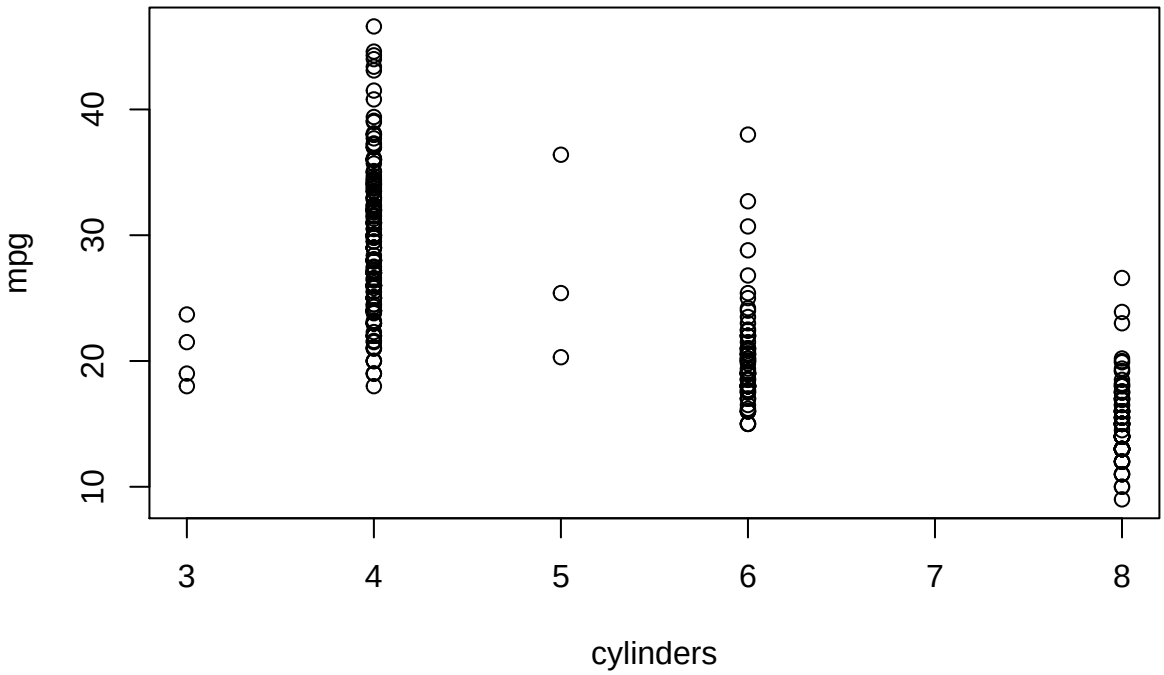
We can use the `plot()` function to produce *scatterplots* of the quantitative variables. However, simply typing the variable names will produce an error message, because `R` does not know how to look in the `Auto` data set for those variables.

To refer to a variable, we must type the data set and the variable name joined with a `$` symbol. Alternatively, we can use the `attach()` function in order to tell `R` to make the variables in this data frame available by name.

```
plot(Auto$cylinders, Auto$mpg)
```




```
attach(Auto)
plot(cylinders, mpg)
```

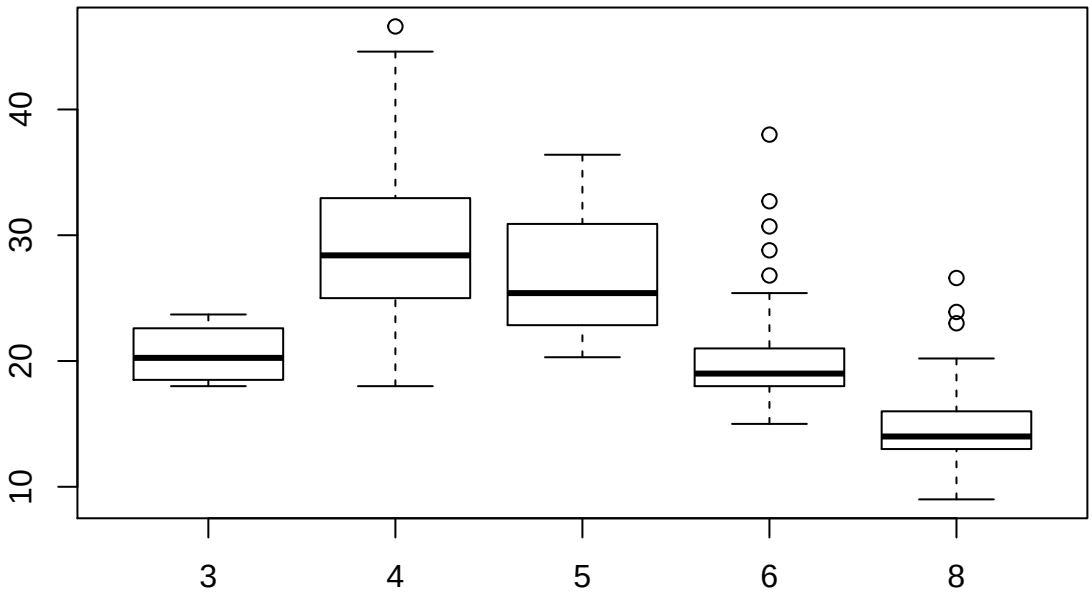


The `cylinders` variable is stored as a numeric vector, so `R` has treated it as quantitative. However, since there are only a small number of possible values for `cylinders`, one may prefer to treat it as a qualitative variable. The `as.factor()` function converts quantitative variables into qualitative variables.

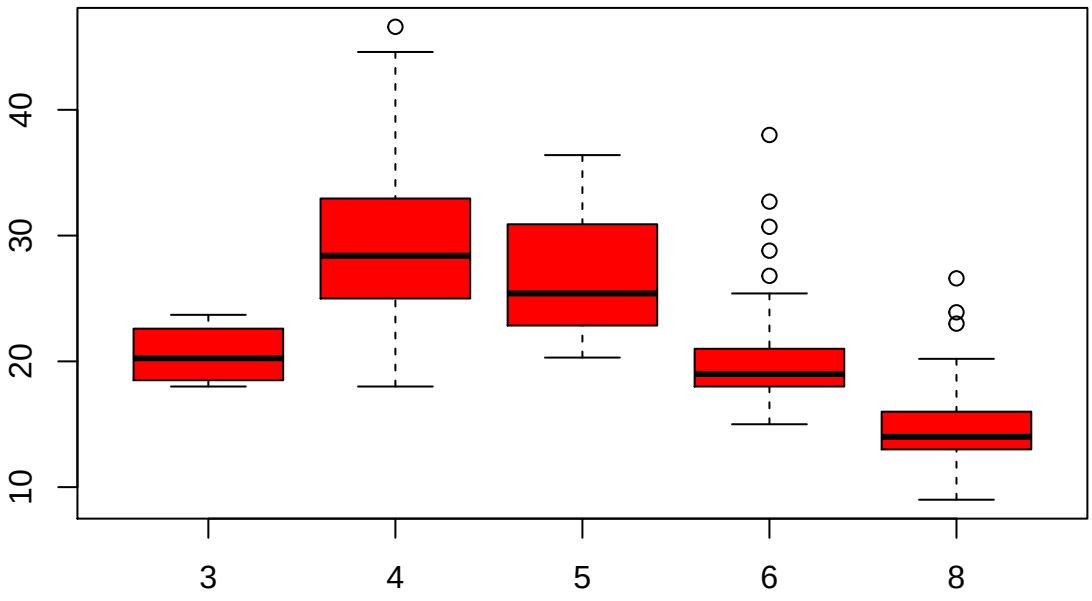
```
cylinders = as.factor(cylinders)
```

If the variable plotted on the x-axis is categorical, then *boxplots* will automatically be produced by the `plot()` function. As usual, a number of options can be specified in order to customize the plots.

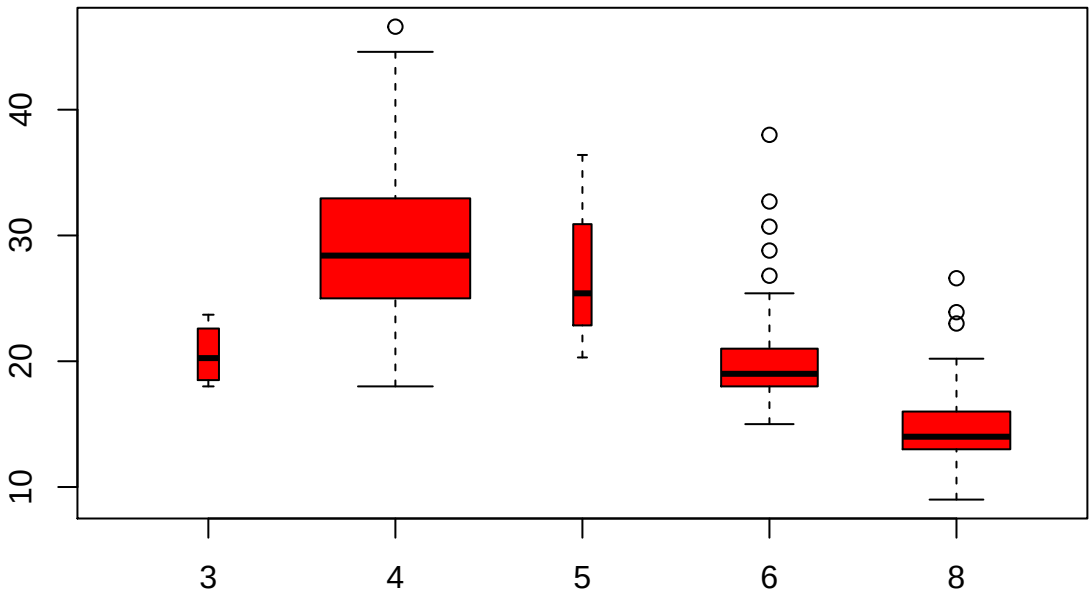
```
plot(cylinders, mpg)
```



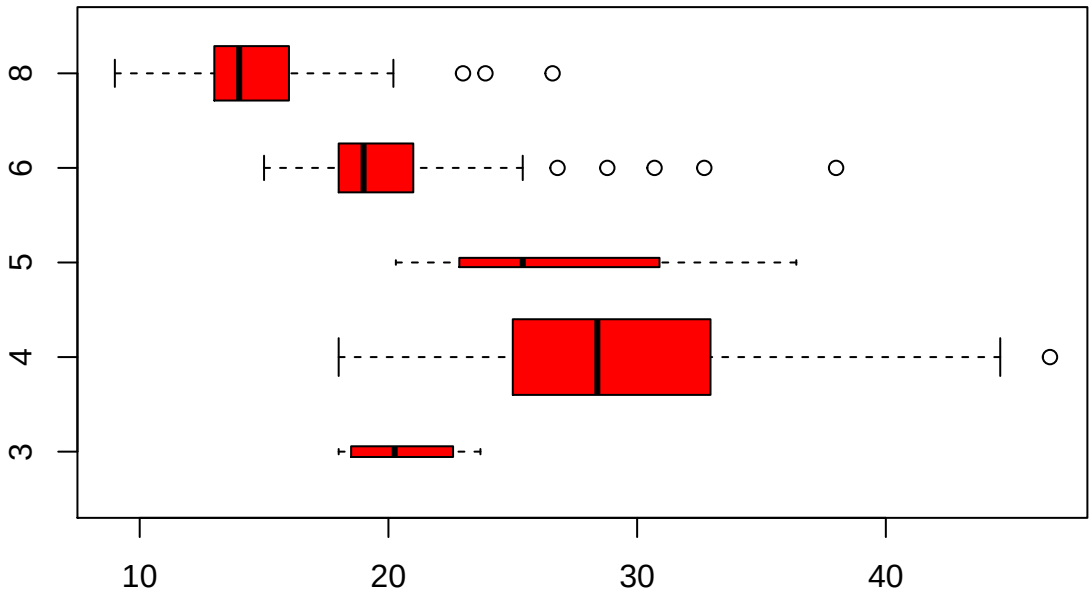

```
plot(cylinders, mpg, col="red")
```



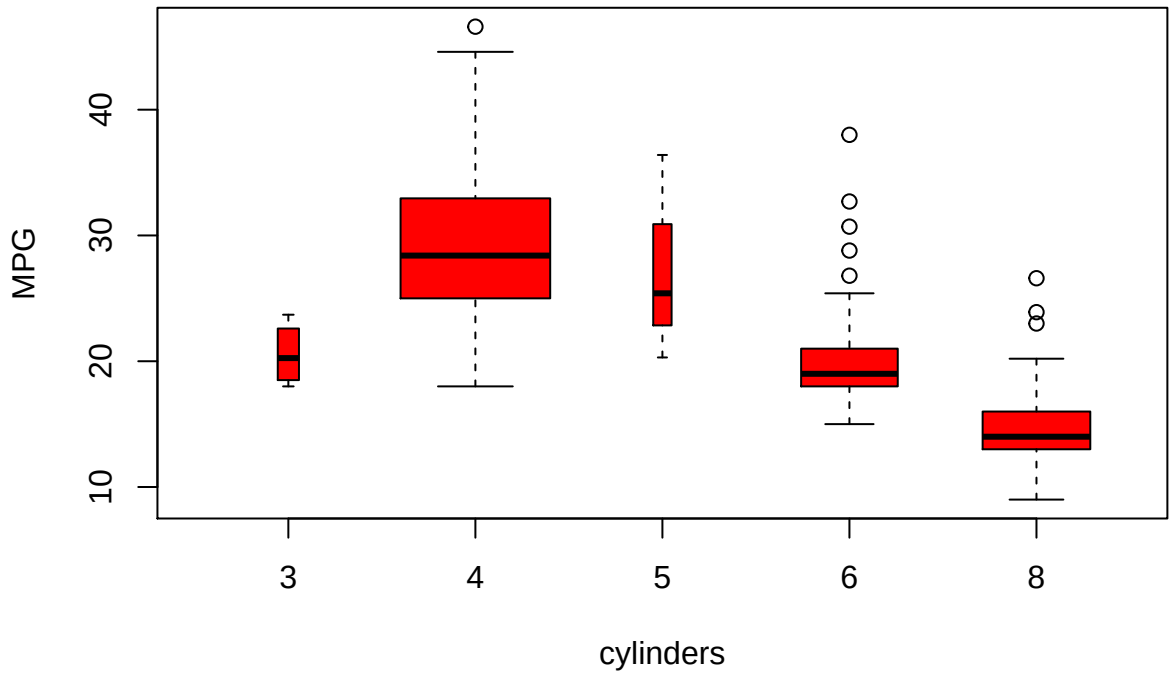

```
plot(cylinders, mpg, col="red", varwidth=T)
```




```
plot(cylinders, mpg, col="red", varwidth=T, horizontal=T)
```

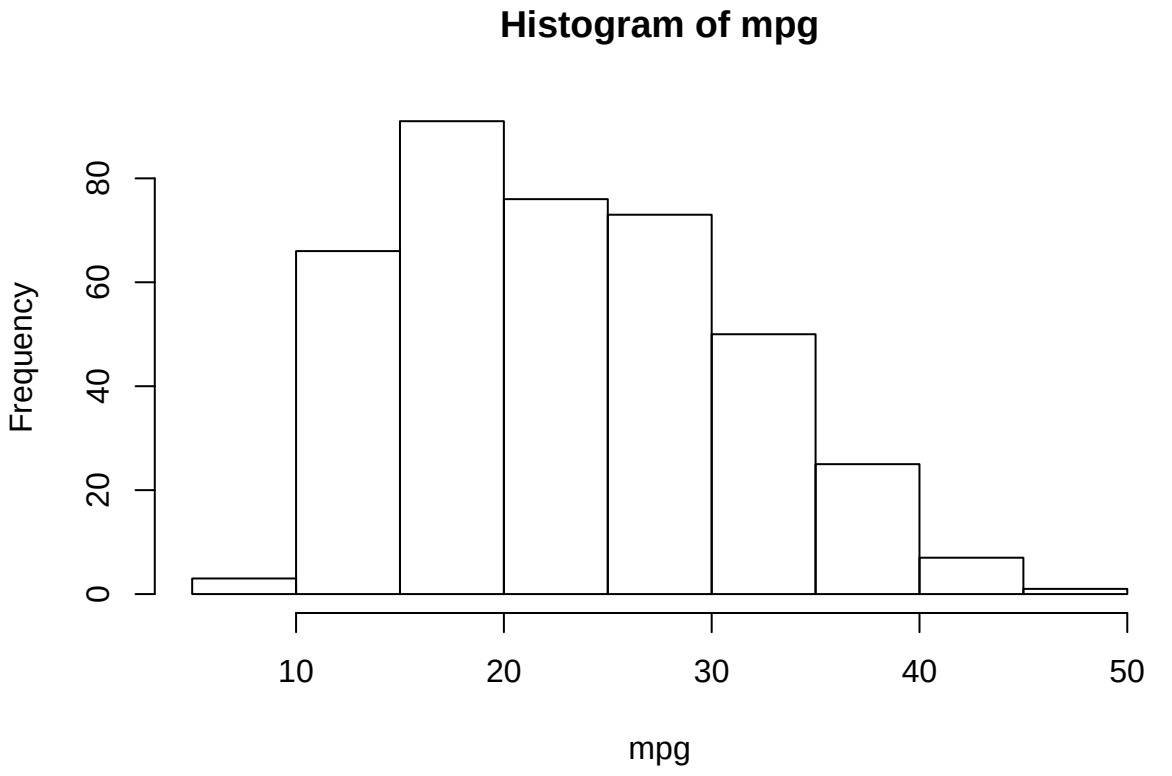



```
plot(cylinders, mpg, col="red", varwidth=T, xlab="cylinders", ylab="MPG")
```

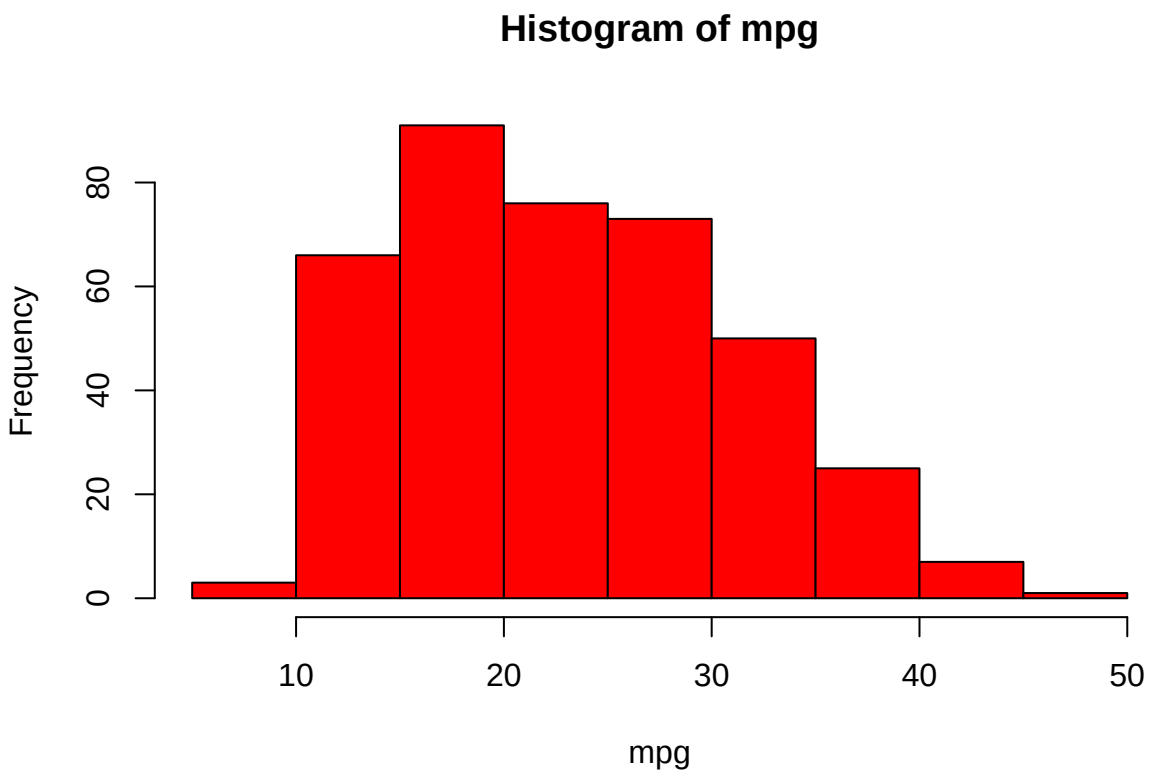


The `hist()` function can be used to plot a *histogram*. Note that `col=2` has the same effect as `col="red"`.

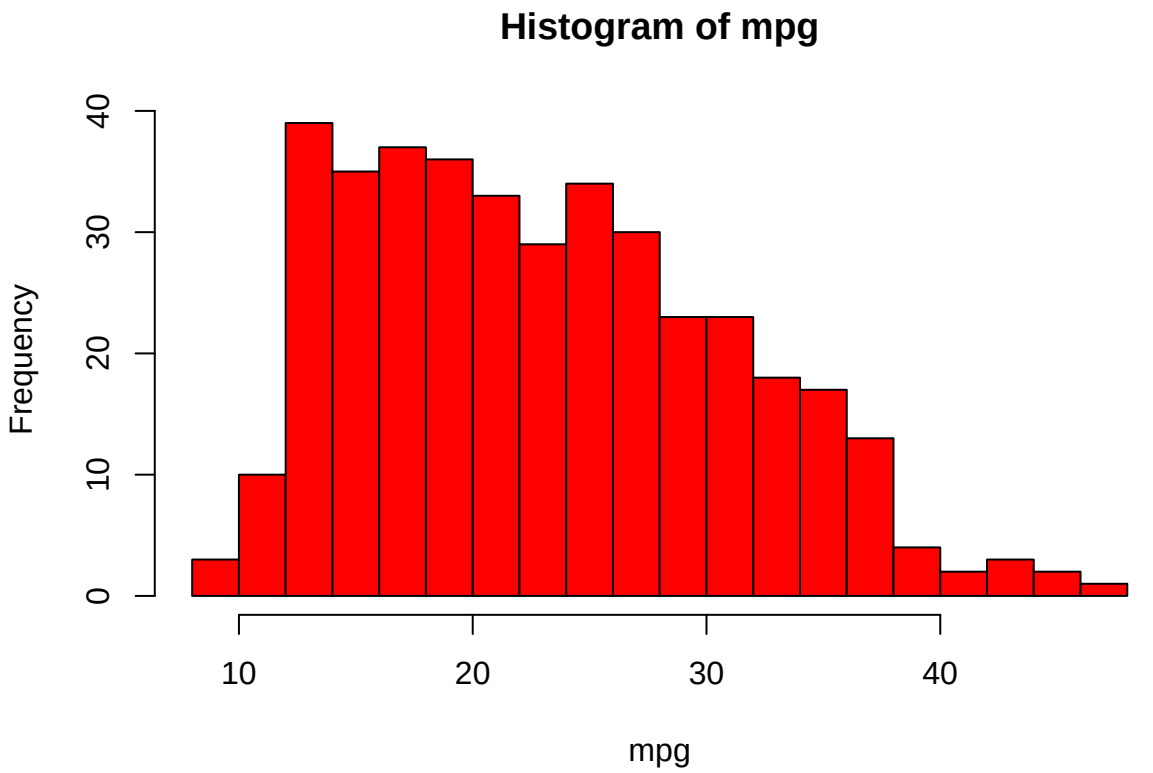
```
hist(mpg)
```




```
hist(mpg, col=2)
```

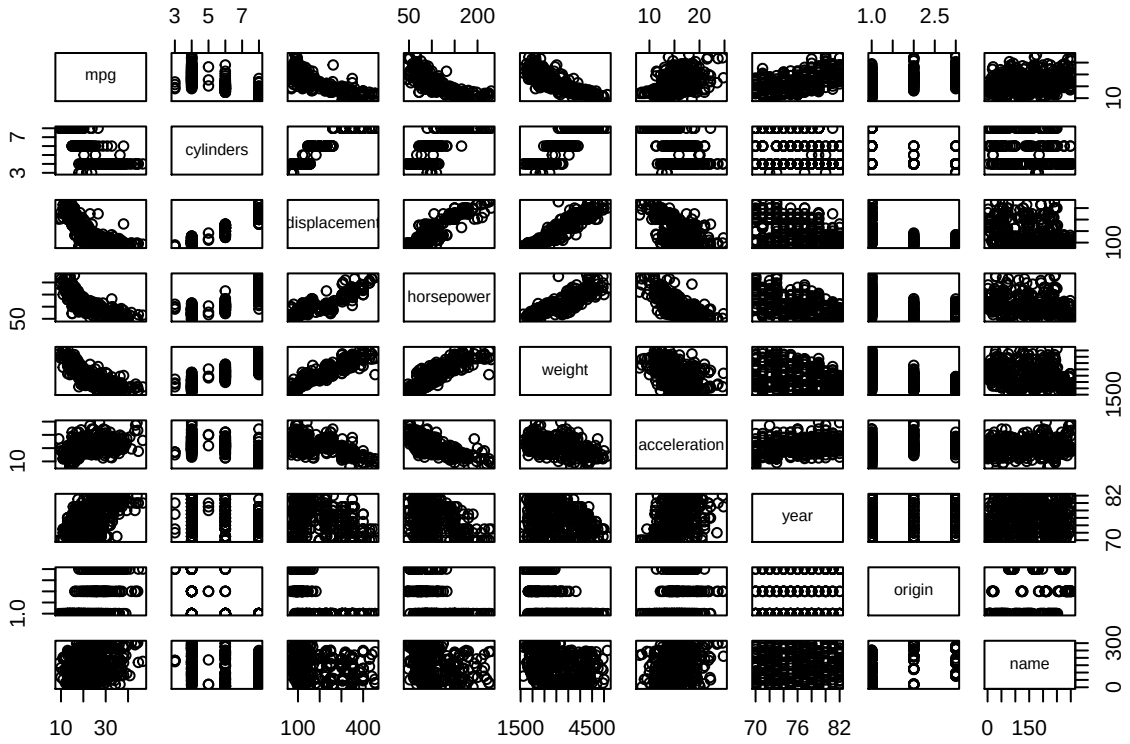



```
hist(mpg, col=2, breaks=15)
```

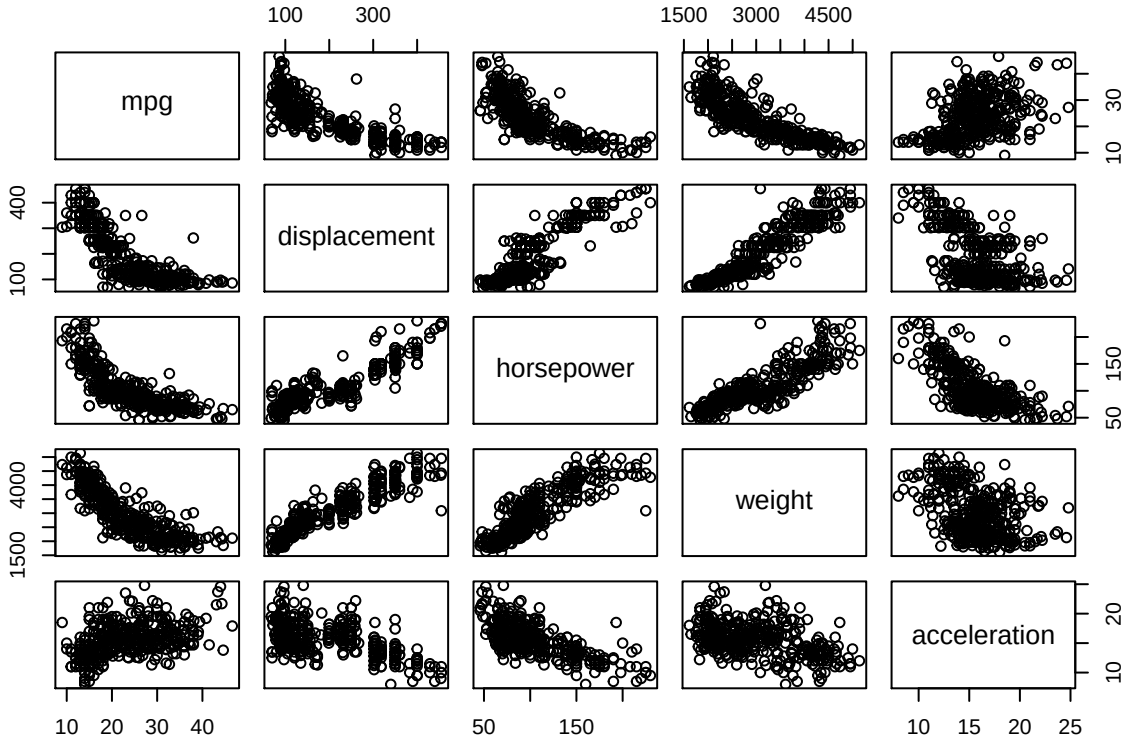


The `pairs()` function creates a *scatterplot matrix*; i.e. a scatterplot for every pair of variables for any given data set. We can also produce scatterplots for just a subset of the variables.

```
pairs(Auto)
```



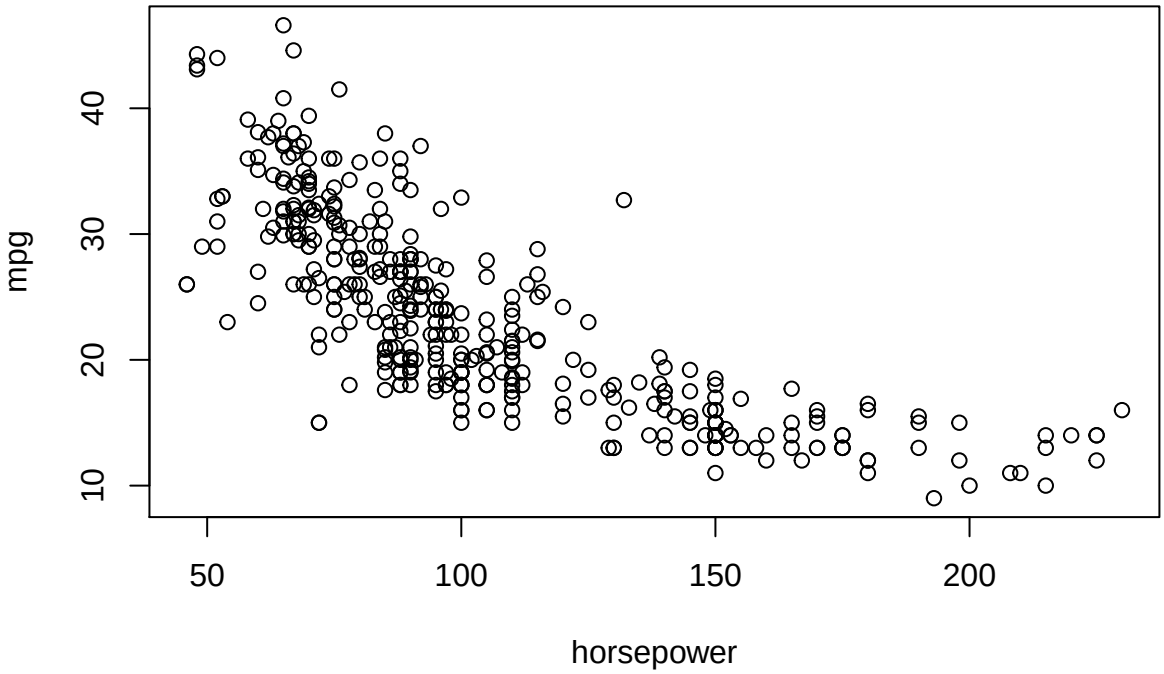

```
pairs(~mpg + displacement + horsepower + weight + acceleration, Auto)
```



In conjunction with the `plot()` function, `identify()` provides a useful interactive method for identifying the value for a particular variable for points on a plot. We pass in three arguments to `identify()`: the x-axis

variable, the y-axis variable, and the variable whose values we would like to see printed for each point. Then clicking on a given point in the plot will cause `R` to print the value of the variable of interest. Right-clicking on the plot will exit the `identify()` function. The numbers printed under the `identify()` function correspond to the rows for the selected points.

```
plot(horsepower, mpg)
```




```
identify(horsepower, mpg, name)
```

The `summary()` function produces a numerical summary of each variable in a particular data set.

```
summary(Auto)
```

```
      mpg          cylinders      displacement     horsepower
 Min.   : 9.00   Min.   :3.000   Min.   : 68.0   Min.   : 46.0
 1st Qu.:17.00   1st Qu.:4.000   1st Qu.:105.0   1st Qu.: 75.0
 Median :22.75   Median :4.000   Median :151.0   Median : 93.5
 Mean   :23.45   Mean   :5.472   Mean   :194.4   Mean   :104.5
 3rd Qu.:29.00   3rd Qu.:8.000   3rd Qu.:275.8   3rd Qu.:126.0
 Max.   :46.60   Max.   :8.000   Max.   :455.0   Max.   :230.0

     weight      acceleration        year           origin
 Min.   :1613   Min.   : 8.00   Min.   :70.00   Min.   :1.000
 1st Qu.:2225   1st Qu.:13.78   1st Qu.:73.00   1st Qu.:1.000
 Median :2804   Median :15.50   Median :76.00   Median :1.000
 Mean   :2978   Mean   :15.54   Mean   :75.98   Mean   :1.577
 3rd Qu.:3615   3rd Qu.:17.02   3rd Qu.:79.00   3rd Qu.:2.000
 Max.   :5140   Max.   :24.80   Max.   :82.00   Max.   :3.000

                 name
 amc matador       :  5
```

```
 ford pinto         :  5
 toyota corolla     :  5
 amc gremlin        :  4
 amc hornet         :  4
 chevrolet chevette:  4
 (Other)            :365
```

For qualitative variables such as `name`, `R` will list the number of observations that fall into each category. We can also produce a summary of just a single variable.

```
summary(mpg)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   9.00   17.00   22.75   23.45   29.00   46.60
```

Once we have finished using `R`, we type `q()` in order to shut it down, or quit. When exiting `R`, we have the option to save the current *workspace* so that all objects (such as data sets) that we have created in this `R` session will be available next time. Before exiting `R`, we may want to save a record of all commands that we typed in the most recent session; this can be accomplished using the `savehistory()` function. Next time we enter `R`, we can load that history using the `loadhistory()` function.