Introduction to R

In this lab, we will introduce some simple `R` commands. The best way to learn a new language is to try out the commands. `R` can be downloaded from http://cran.r-project.org/.

Basic Commands

`R` uses *functions* to perform operations. To run a function called `funcname`, we type `funcname(input1, input2)`, where the inputs (or *arguments*) `input1` and `input2` tell `R` how to run the function. A function can have any number of inputs. For example, to create a vector of numbers, we use the function `c()` (for *concatenate*). Any numbers inside the parentheses are joined together. The following command instructs `R` to join together the numbers 1, 3, 2, and 5, and to save them as a *vector* named x. When we type `x`, it gives us back the vector.

Hide

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

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

We can also save things using `=` rather than `<-` though I don't recommend it:

Hide

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

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

Hide

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

Hitting the *up* arrow multiple times will display the previous commands, which can then be edited. This is useful since one often wishes to repeat a similar command. In addition, typing `?funcname` will always cause `R` to open a new help file window with additional information about the function `funcname`.

We can tell `R` to add two sets of numbers together. It will then add the first number from `x` to the first number from `y`, and so on. However, `x` and `y` should be the same length. We can check their length using the `length()` function.

Hide

```
length(x)
```

```
[1] 3
```

Hide

```
length(y)
```

```
[1] 3
```

Hide

```
x+y
```

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

The `ls()` function allows us to look at a list of all of the objects, such as data and functions, that we have saved so far. The `rm()` function can be used to delete any that we don't want.

Hide

```
ls()
```

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

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

Hide

```
character(0)
```

It's also possible to remove all objects at once:

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

Hide

The `matrix()` function can be used to create a matrix of numbers. Before we use the `matrix()` function, we can learn more about it:

```
?matrix
```

Hide

The help file reveals that the `matrix()` function takes a number of inputs, but for now we focus on the first three: the data (the entries in the matrix), the number of rows, and the number of columns. First, we create a simple matrix.

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

Hide

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

Note that we could just as well omit typing `data=`, `nrow=`, and `ncol=` in the `matrix()` command above: that is, we could just type

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

Hide

and this would have the same effect. However, it can sometimes be useful to specify the names of the arguments passed in, since otherwise `R` will assume that the function arguments are passed into the function in the same order that is given in the function's help file. As this example illustrates, by default `R` creates matrices by successively filling in columns. Alternatively, the `byrow=TRUE` option can be used to populate the matrix in order of the rows.

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

Hide

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

Notice that in the above command we did not assign the matrix to a value such as `x`. In this case the matrix is printed to the screen but is not saved for future calculations. 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`; any powers are possible, including fractional or negative powers.

```
sqrt(x)
```

Hide

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

```
x^2
```

Hide

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

The `rnorm()` function generates a vector of random normal variables, with first argument `n` the sample size. Each time we call this function, we will get a different answer. Here we create two correlated sets of numbers, `x` and `y`, and use the `cor()` function to compute the correlation between them.

```
x=rnorm(50)
y=x+rnorm(50,mean=50,sd=.1)
cor(x,y)
```

Hide

```
[1] 0.9960598
```

By default, `rnorm()` creates standard normal random variables with a mean of 0 and a standard deviation of 1. However, the mean and standard deviation can be altered using the `mean` and `sd` arguments, as illustrated above. Sometimes we want our code to reproduce the exact same set of random numbers; we can use the `set.seed()` function to do this. The `set.seed()` function takes an (arbitrary) integer argument.

```
set.seed(1303)
rnorm(50)
```

Hide

```
 [1] -1.1439763145  1.3421293656  2.1853904757  0.5363925179  0.0631929665  0.5022344825 -0.0004
167247
 [8]  0.5658198405 -0.5725226890 -1.1102250073 -0.0486871234 -0.6956562176  0.8289174803  0.2066
528551
[15] -0.2356745091 -0.5563104914 -0.3647543571  0.8623550343 -0.6307715354  0.3136021252 -0.9314
953177
[22]  0.8238676185  0.5233707021  0.7069214120  0.4202043256 -0.2690521547 -1.5103172999 -0.6902
124766
[29] -0.1434719524 -1.0135274099  1.5732737361  0.0127465055  0.8726470499  0.4220661905 -0.0188
157917
[36]  2.6157489689 -0.6931401748 -0.2663217810 -0.7206364412  1.3677342065  0.2640073322  0.6321
868074
[43] -1.3306509858  0.0268888182  1.0406363208  1.3120237985 -0.0300020767 -0.2500257125  0.0234
144857
[50]  1.6598706557
```

We use `set.seed()` throughout the labs whenever we perform calculations involving random quantities. In general this should allow the user to reproduce our results. However, it should be noted that as new versions of `R` become available it is possible that some small discrepancies may form between the book and the output from `R`.

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. Or we can simply use the `sd()` function.

Hide

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

```
[1] 0.01103557
```

```
var(y)
```

Hide

```
[1] 0.7328675
```

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

Hide

```
[1] 0.8560768
```

```
sd(y)
```

Hide

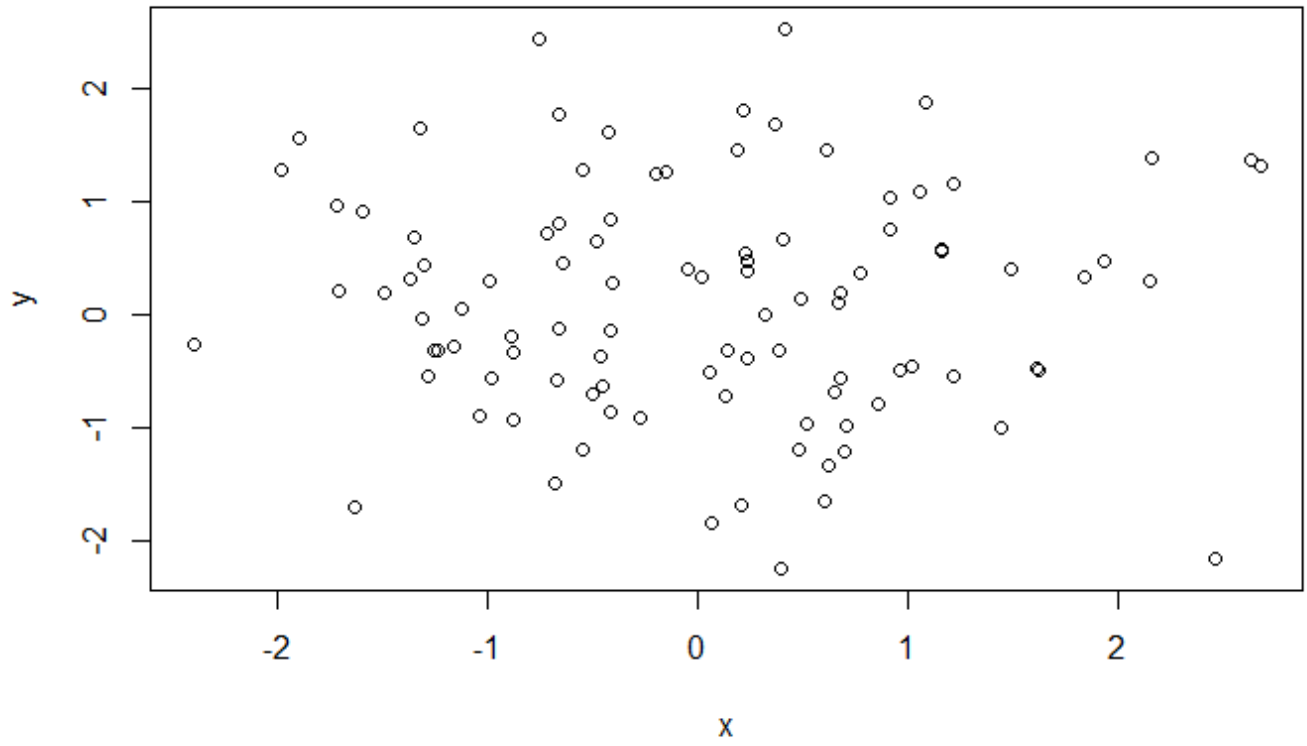
```
[1] 0.8560768
```

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. To find out more information about the `plot()` function, type `?plot`.

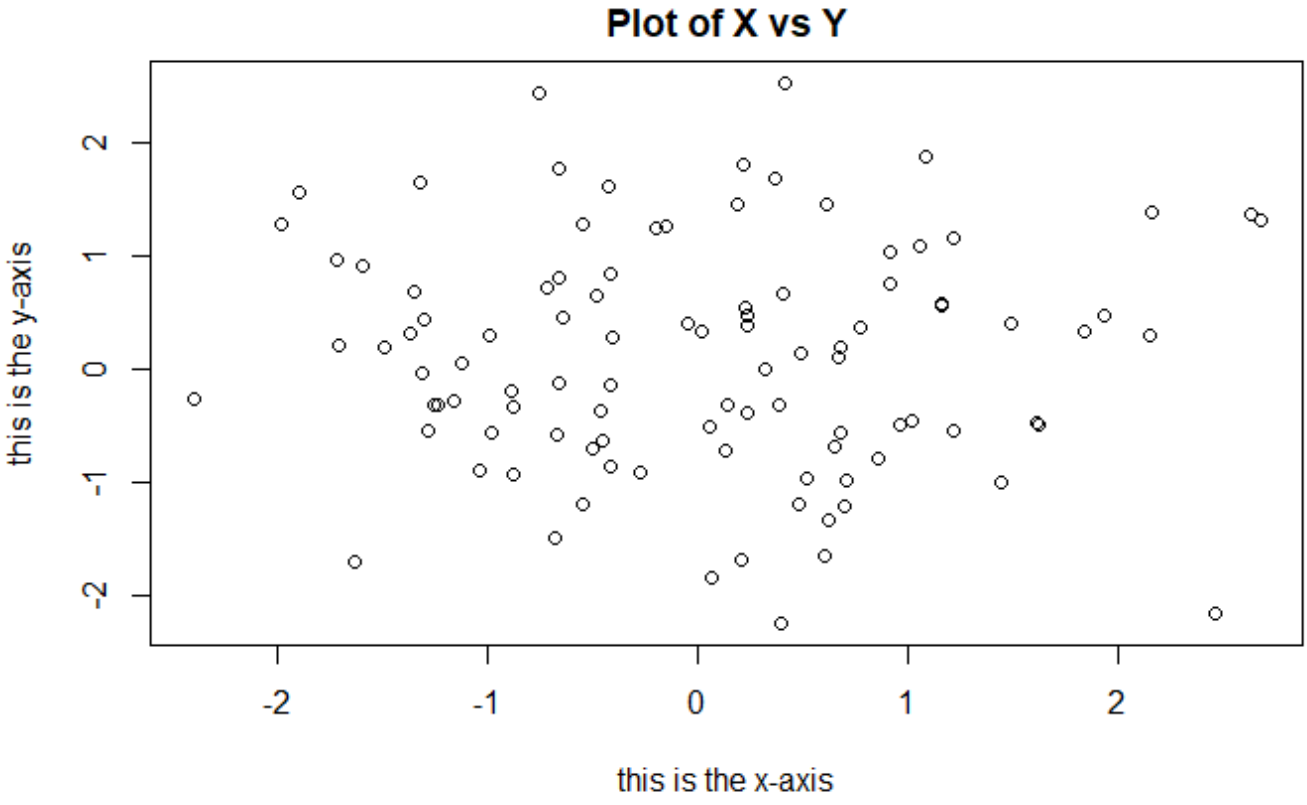
```
x=rnorm(100)
y=rnorm(100)
plot(x,y)
```

Hide




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

Hide



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.

Hide

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

The function `dev.off()` indicates to `R` that we are done creating the plot. Alternatively, we can simply copy the plot window and paste it into an appropriate file type, such as a Word document.

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.

Hide

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

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

Hide

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

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

Hide

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

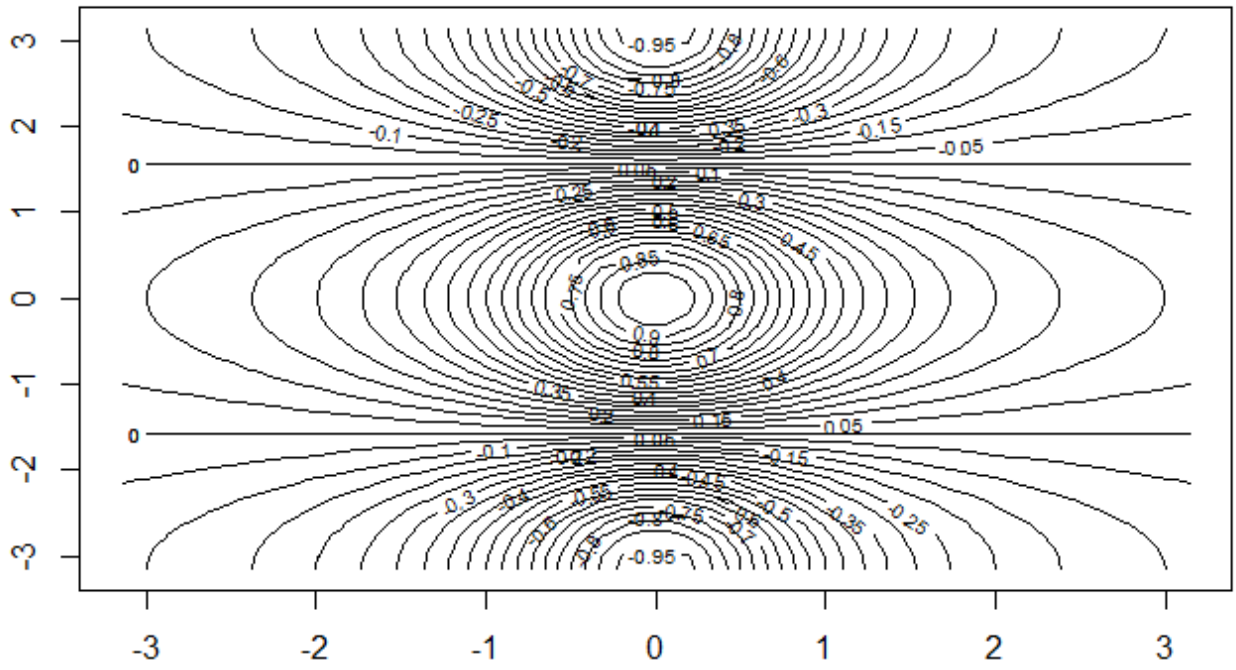
We will now create some more sophisticated plots. The `contour()` function produces a contour plot in order to represent three-dimensional data; *contour plot* it is like a topographical map. It takes three arguments:

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. To learn more about these, take a look at the help file by typing `?contour` .

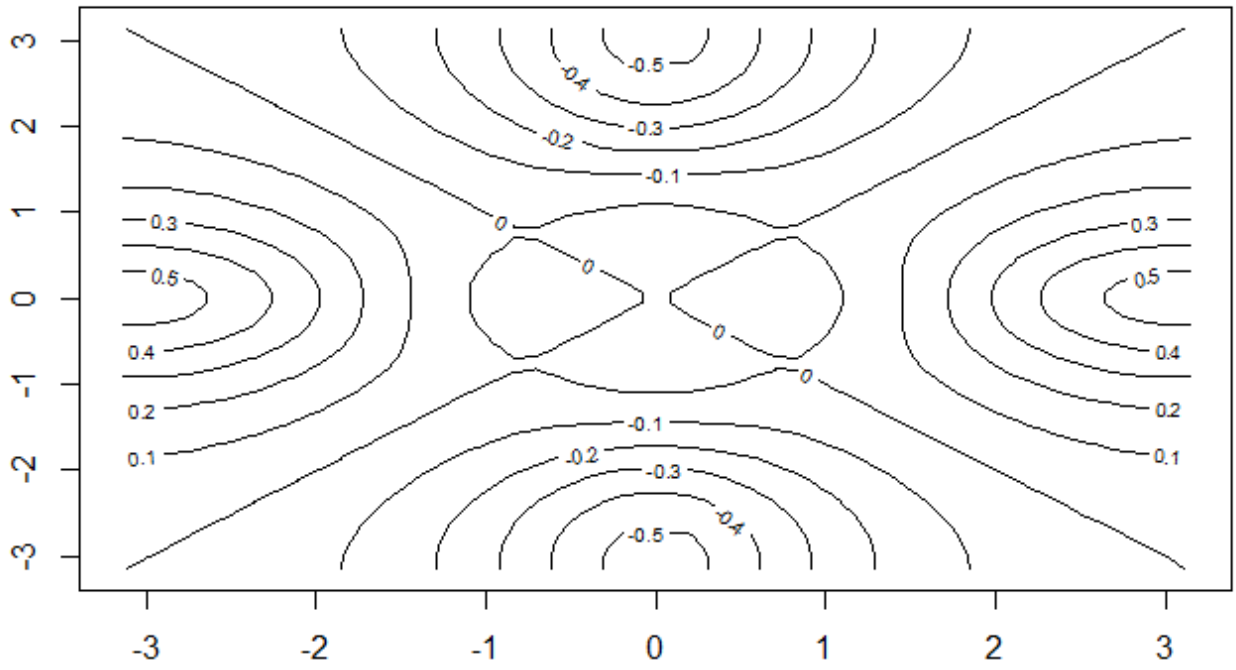
Hide

```
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)
```



Hide

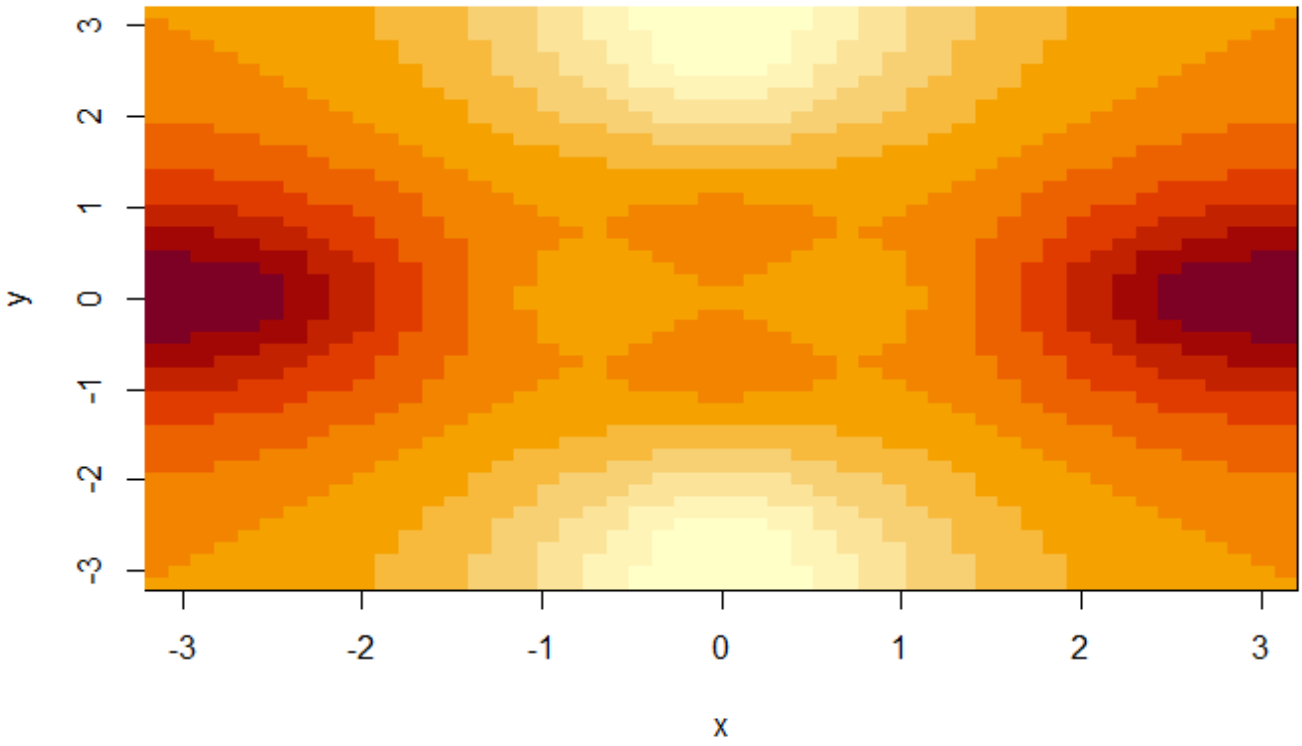
```
fa=(f-t(f))/2
contour(x,y,fa,nlevels=15)
```



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 `phi` control the angles at which the plot is viewed.

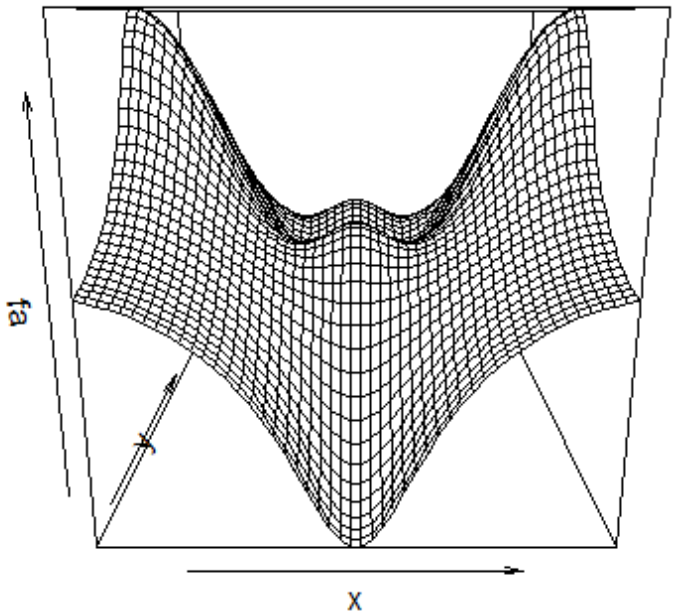
Hide

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



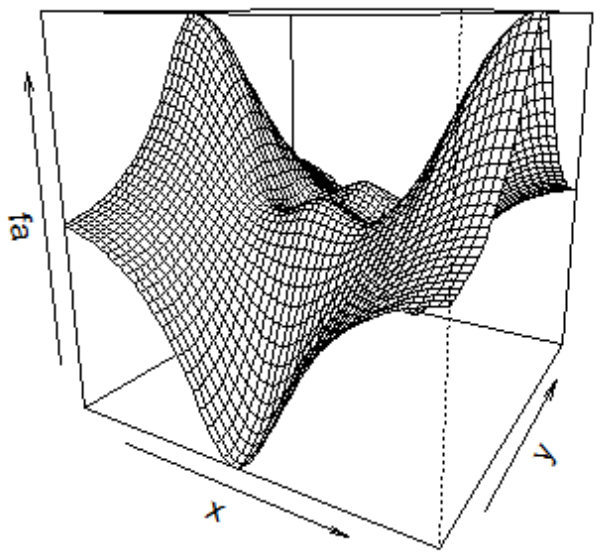

```
persp(x,y,fa)
```

Hide



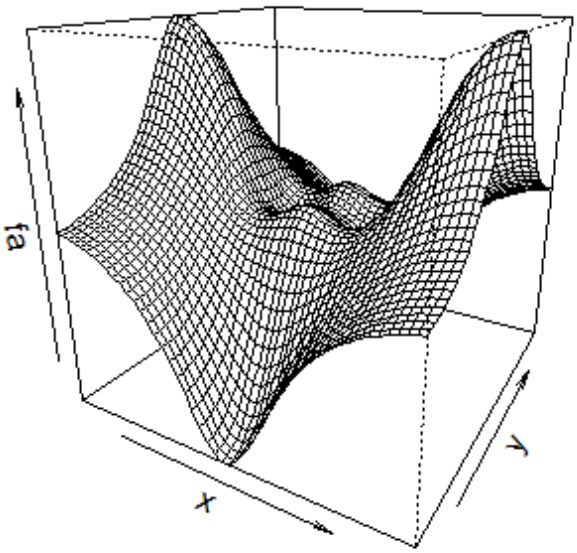

```
persp(x,y,fa,theta =30)
```

Hide



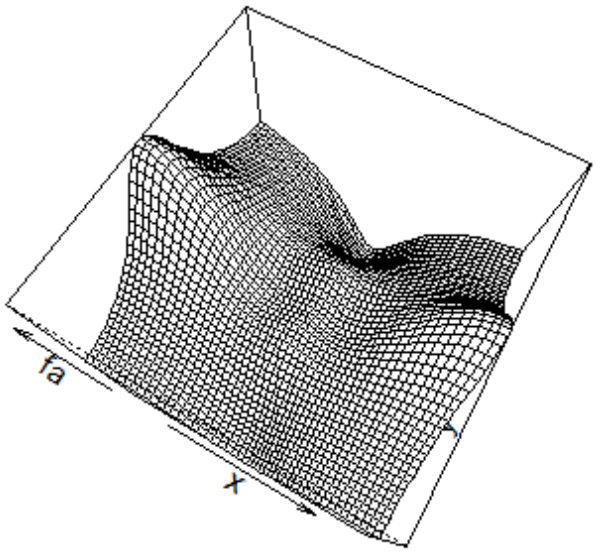

```
persp(x,y,fa,theta=30,phi=20)
```

Hide



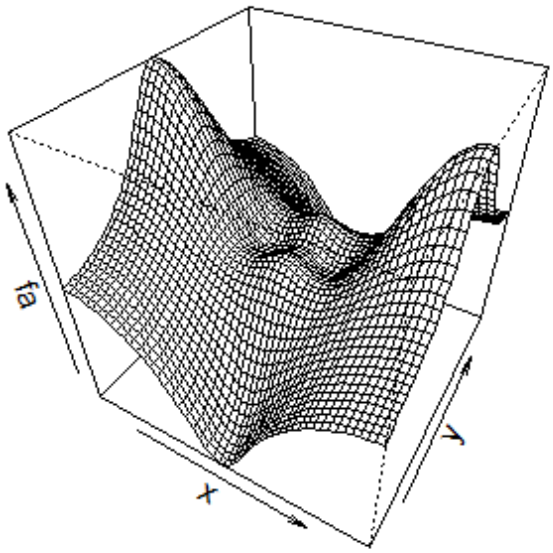

```
persp(x,y,fa,theta=30,phi=70)
```

Hide




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

Hide



Indexing Data

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

Hide

```
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

Hide

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

```
[1] 10
```

will select the element corresponding to the second row and the third column. The first number after the open-bracket symbol `[` 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.

Hide

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

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

Hide

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

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

Hide

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

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

Hide

```
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 rows, respectively. `R` treats a single row or column of a matrix as a vector.

```
A[1,]
```

Hide

```
[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),]
```

Hide

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

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

Hide

```
[1] 6 8
```

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

```
dim(A)
```

Hide

```
[1] 4 4
```

Loading Data

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

Before attempting to load a data set, we must make sure that `R` knows to search for the data in the proper directory. For example on a Windows system one could select the directory using the `Change dir...` option under the `File` menu. However, the details of how to do this depend on the operating system (e.g. Windows, Mac, Unix) that is being used, and so we do not give further details here. We begin by loading in the `Auto` data set. This data is part of the `ISLR` library (we discuss libraries in Chapter 3) but to illustrate the `read.table()` function we load it now from a text file. 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*. (The text file can be obtained from this book's website.) Once the data has been loaded, the `str()` function can be used to view the metadata.

```
Auto=read.table("http://faculty.marshall.usc.edu/gareth-james/ISL/Auto.data")
str(Auto)
```

Hide

```
'data.frame':   398 obs. of  9 variables:
 $ V1: Factor w/ 130 levels "10.0","11.0",..: 130 17 7 17 9 13 7 5 5 5 ...
 $ V2: Factor w/ 6 levels "3","4","5","6",..: 6 5 5 5 5 5 5 5 5 5 ...
 $ V3: Factor w/ 83 levels "100.0","101.0",..: 83 50 53 51 48 47 59 61 60 62 ...
 $ V4: Factor w/ 95 levels "?","100.0","102.0",..: 95 17 35 29 29 24 42 47 46 48 ...
 $ V5: Factor w/ 351 levels "1613.","1649.",..: 351 247 265 241 240 244 318 319 315 327 ...
 $ V6: Factor w/ 97 levels "10.0","10.5",..: 97 10 8 3 10 2 1 94 93 1 ...
 $ V7: Factor w/ 14 levels "70","71","72",..: 14 1 1 1 1 1 1 1 1 1 ...
 $ V8: Factor w/ 4 levels "1","2","3","origin": 4 1 1 1 1 1 1 1 1 1 ...
 $ V9: Factor w/ 305 levels "amc ambassador brougham",..: 198 49 36 232 14 161 141 54 224 242
...
```

Note that Auto.data is simply a text file, which you could alternatively open on your computer using a standard text editor. It is often a good idea to view a data set using a text editor or other software such as Excel before loading it into `R`.

This particular data set has not been loaded correctly, because `R` has assumed that the variable names are part of the data and so has included them in the first row. 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=T` (or `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.

Hide

```
Auto=read.table("http://faculty.marshall.usc.edu/gareth-james/ISL/Auto.data",header=T,na.s
trings ="?")
str(Auto)
```

```
'data.frame':   397 obs. of  9 variables:
 $ mpg         : num  18 15 18 16 17 15 14 14 14 15 ...
 $ cylinders   : int  8 8 8 8 8 8 8 8 8 8 ...
 $ displacement: num  307 350 318 304 302 429 454 440 455 390 ...
 $ horsepower  : num  130 165 150 150 140 198 220 215 225 190 ...
 $ weight      : num  3504 3693 3436 3433 3449 ...
 $ acceleration: num  12 11.5 11 12 10.5 10 9 8.5 10 8.5 ...
 $ year        : int  70 70 70 70 70 70 70 70 70 70 ...
 $ origin      : int  1 1 1 1 1 1 1 1 1 1 ...
 $ name        : Factor w/ 304 levels "amc ambassador brougham",..: 49 36 231 14 161 141 54 223
241 2 ...
```

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 value) file and then use the `read.csv()` function to load it in.

Hide

```
Auto=read.csv("http://faculty.marshall.usc.edu/gareth-james/ISL/Auto.csv",header=T,na.stri
ngs ="?")
str(Auto)
```

```
'data.frame':   397 obs. of  9 variables:
 $ mpg         : num  18 15 18 16 17 15 14 14 14 15 ...
 $ cylinders   : int  8 8 8 8 8 8 8 8 8 8 ...
 $ displacement: num  307 350 318 304 302 429 454 440 455 390 ...
 $ horsepower  : int  130 165 150 150 140 198 220 215 225 190 ...
 $ weight      : int  3504 3693 3436 3433 3449 4341 4354 4312 4425 3850 ...
 $ acceleration: num  12 11.5 11 12 10.5 10 9 8.5 10 8.5 ...
 $ year        : int  70 70 70 70 70 70 70 70 70 70 ...
 $ origin      : int  1 1 1 1 1 1 1 1 1 1 ...
 $ name        : Factor w/ 304 levels "amc ambassador brougham",..: 49 36 231 14 161 141 54 223
241 2 ...
```

Hide

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

m.	cylinders	displacem…	horsepo…	wei…	acceleration	y…	origin
<dbl>	<int>	<dbl>	<int>	<int>	<dbl>	<int>	<int>
118	8	307	130	3504	12.0	70	1
215	8	350	165	3693	11.5	70	1

m. <dbl>	cylinders <int>	displacem… <dbl>	horsepo… <int>	wei… <int>	acceleration <dbl>	y… <int>	origin <int>
318	8	318	150	3436	11.0	70	1
416	8	304	150	3433	12.0	70	1

4 rows | 1-9 of 9 columns

The `str()` 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 contain missing observations, and so we choose to use the `na.omit()` function to simply remove these rows.

Hide

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

```
[1] 392   9
```

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

Hide

```
names(Auto)
```

```
[1] "mpg"          "cylinders"    "displacement" "horsepower"   "weight"       "acceleration"
[7] "year"         "origin"       "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 to look in the Auto data set for those variables.

Hide

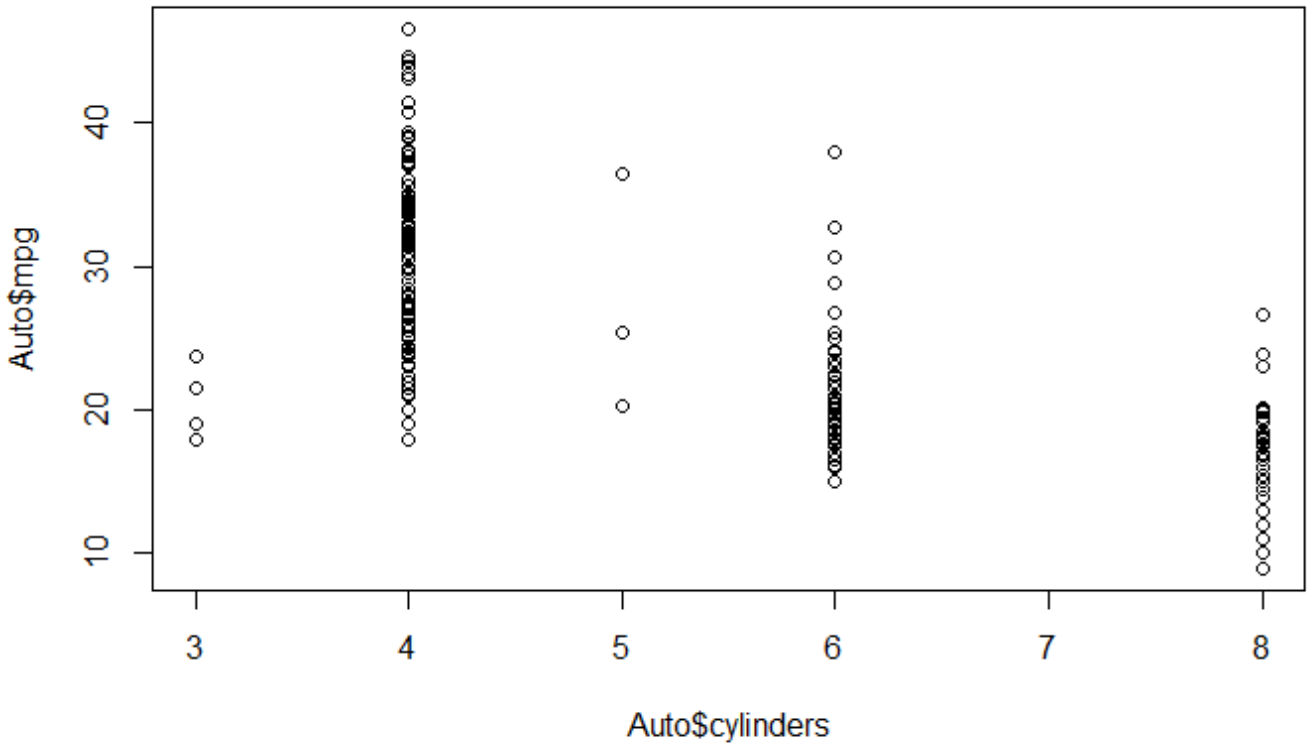
```
plot(cylinders , mpg)
```

```
Error in plot(cylinders, mpg) : object 'cylinders' not found
```

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.

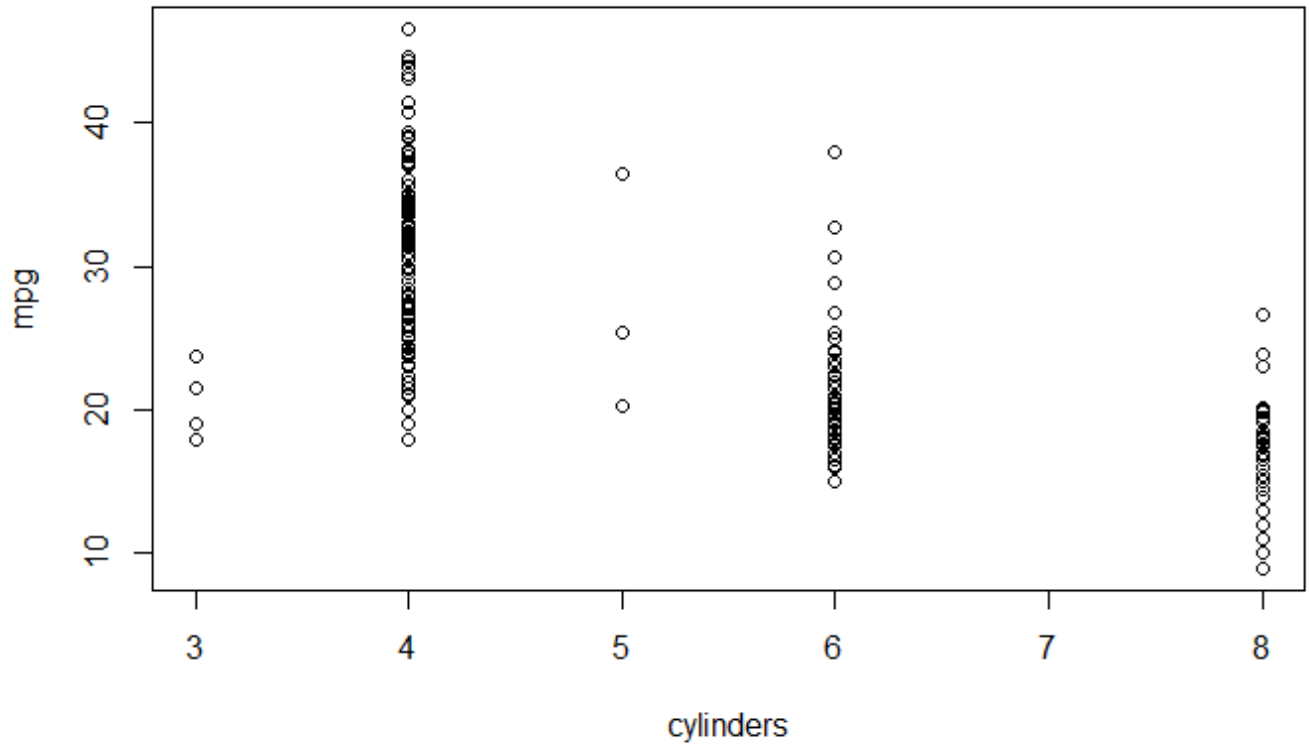
Hide

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




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

Hide



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.

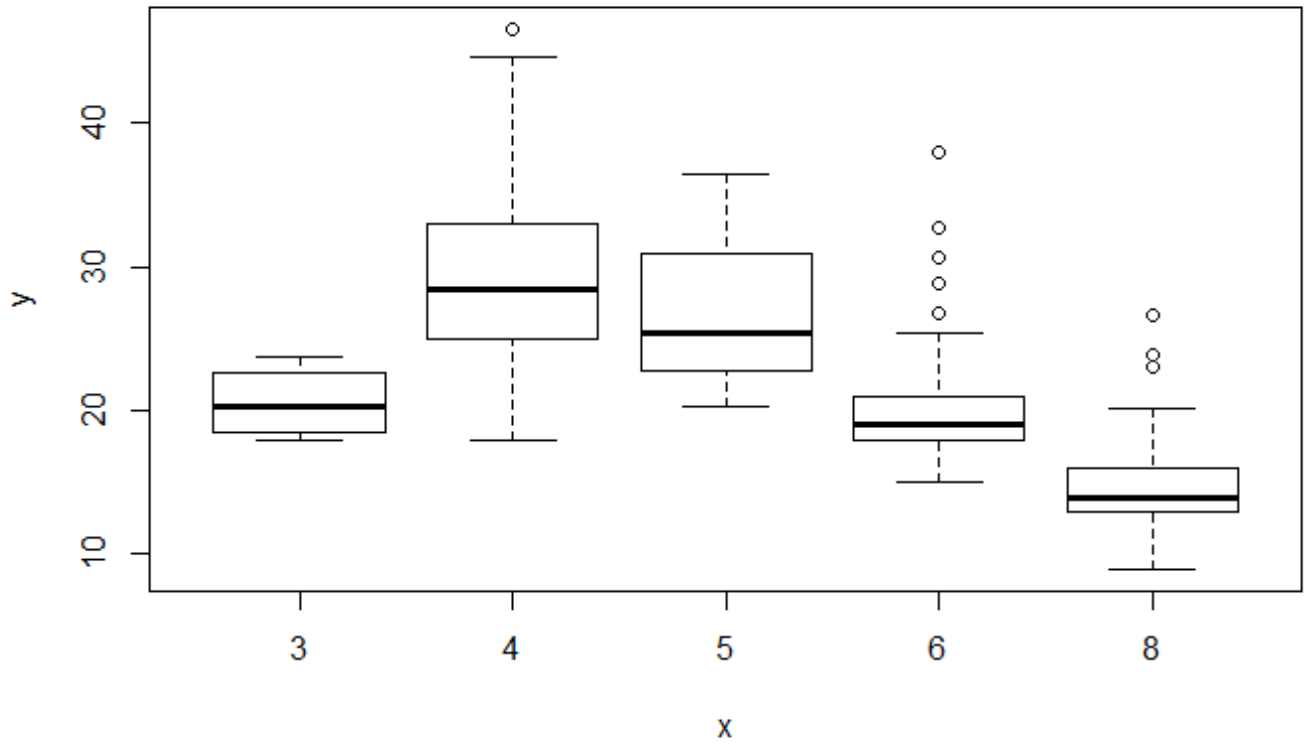
```
cylinders=as.factor(cylinders)
```

Hide

If the variable plotted on the x-axis is categorial, 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.

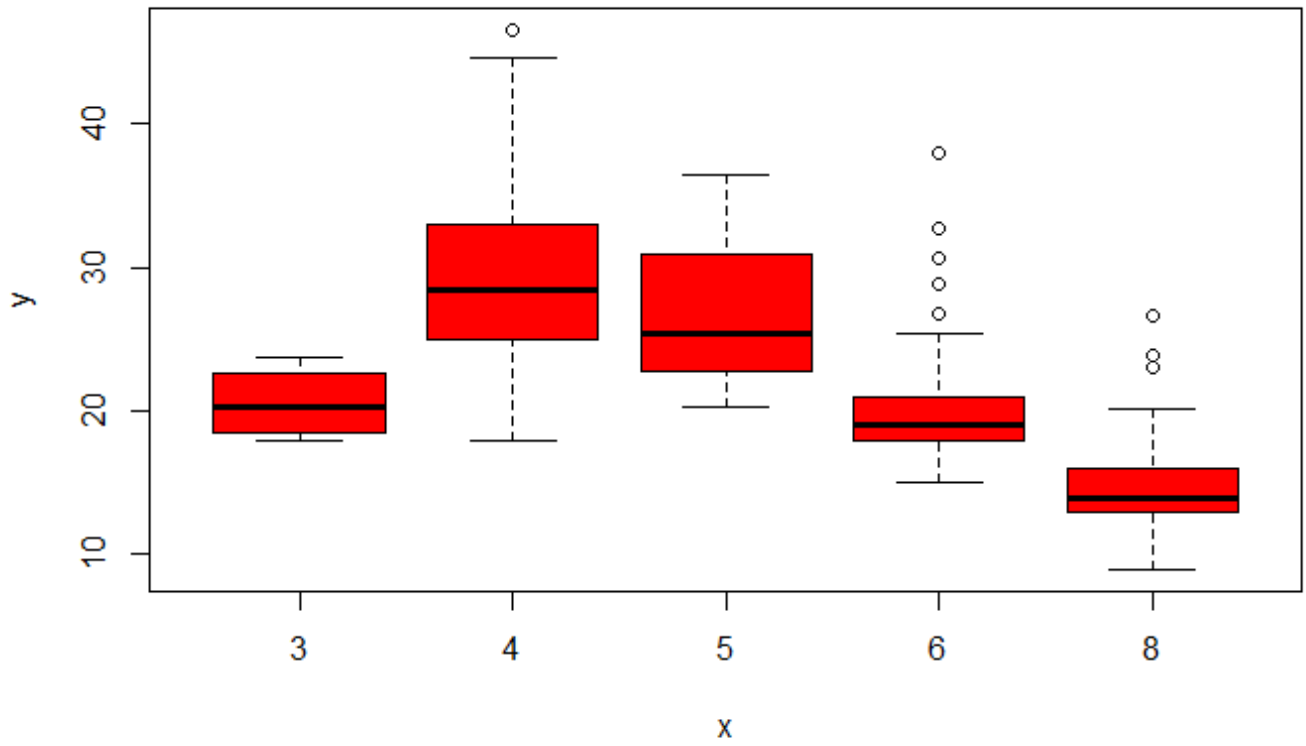
Hide

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



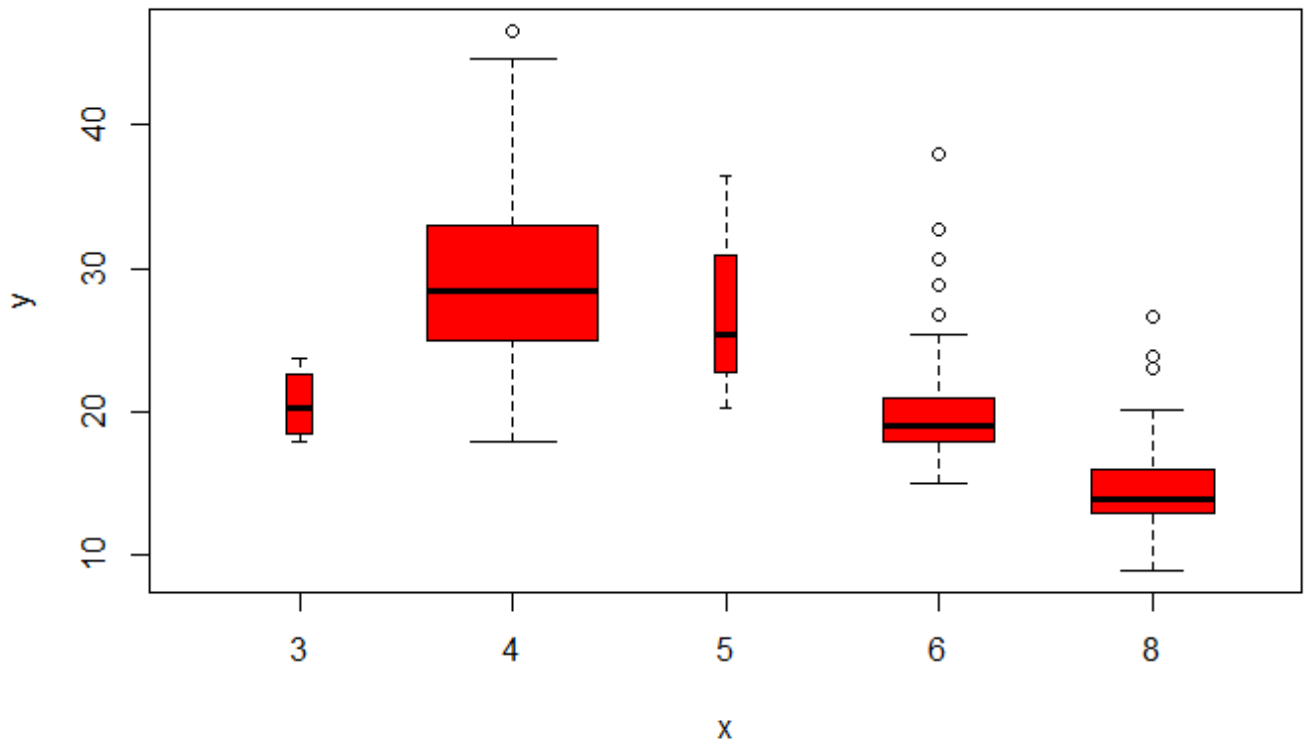
Hide

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



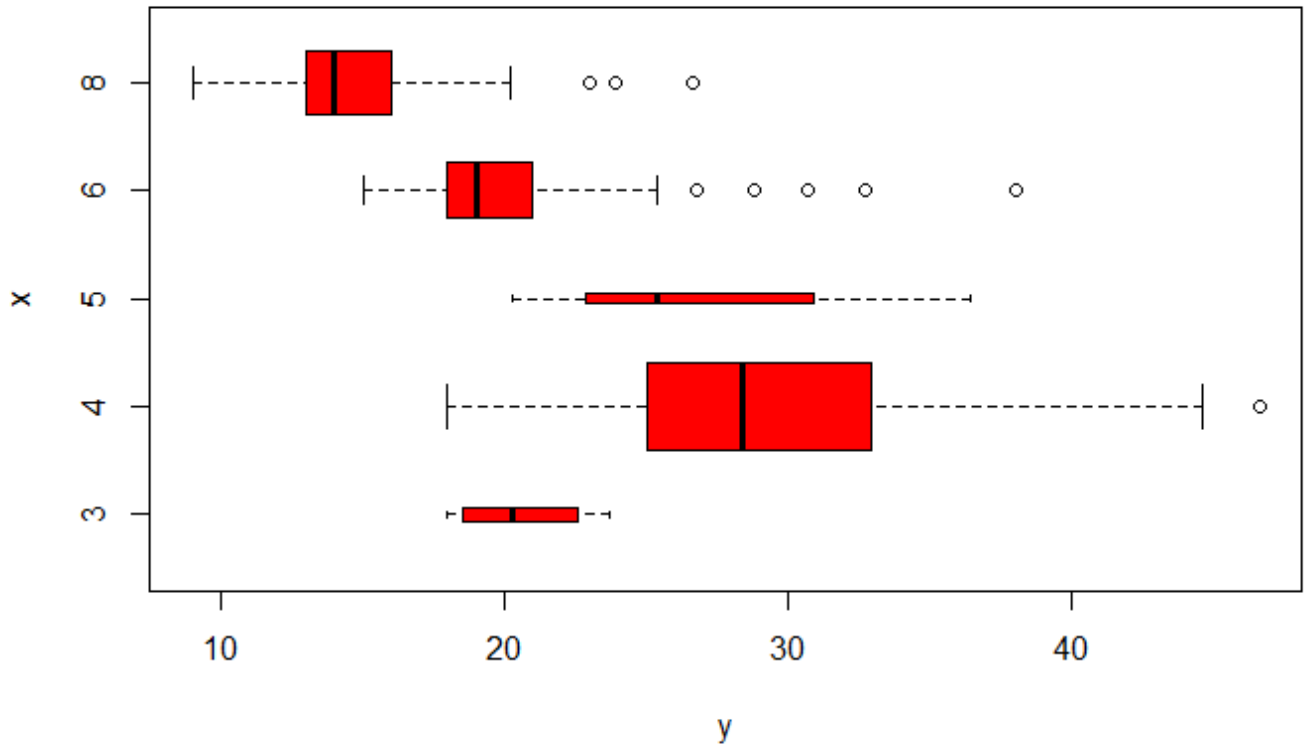
Hide

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



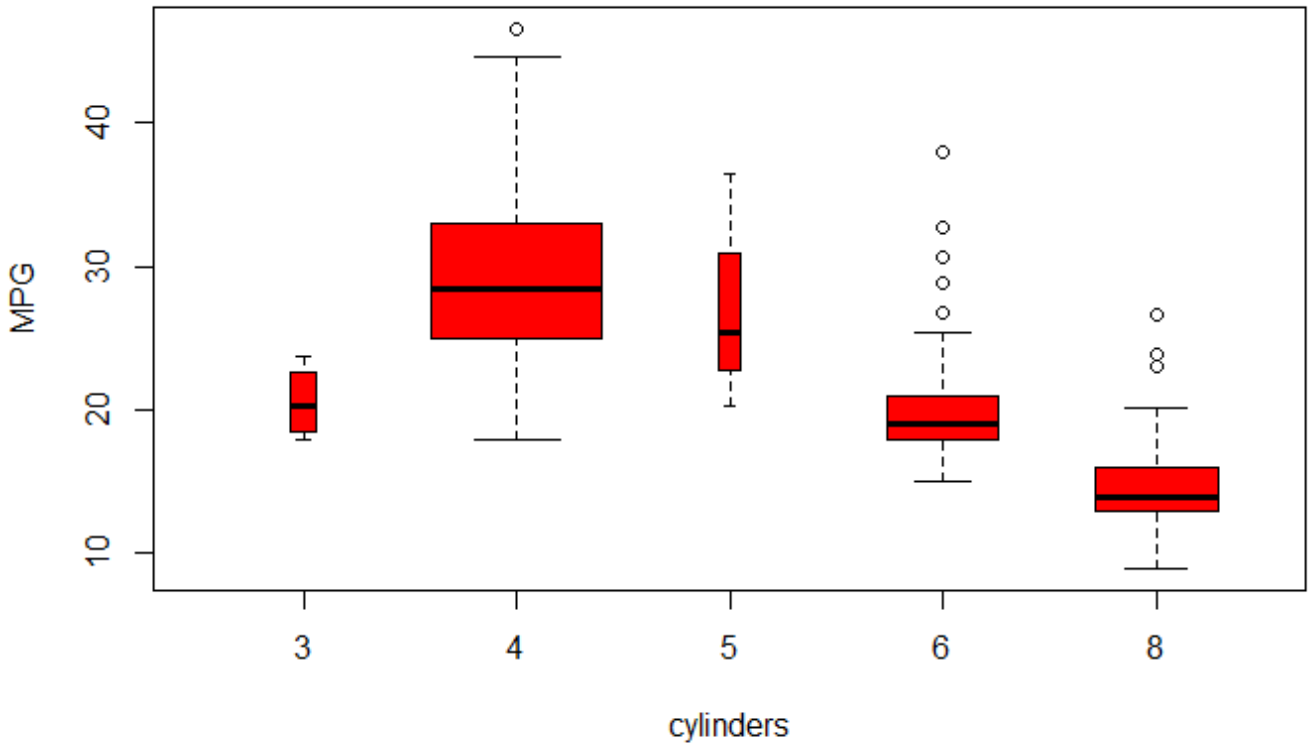

```
plot(cylinders,mpg,col="red",varwidth=T,horizontal =T)
```

Hide




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

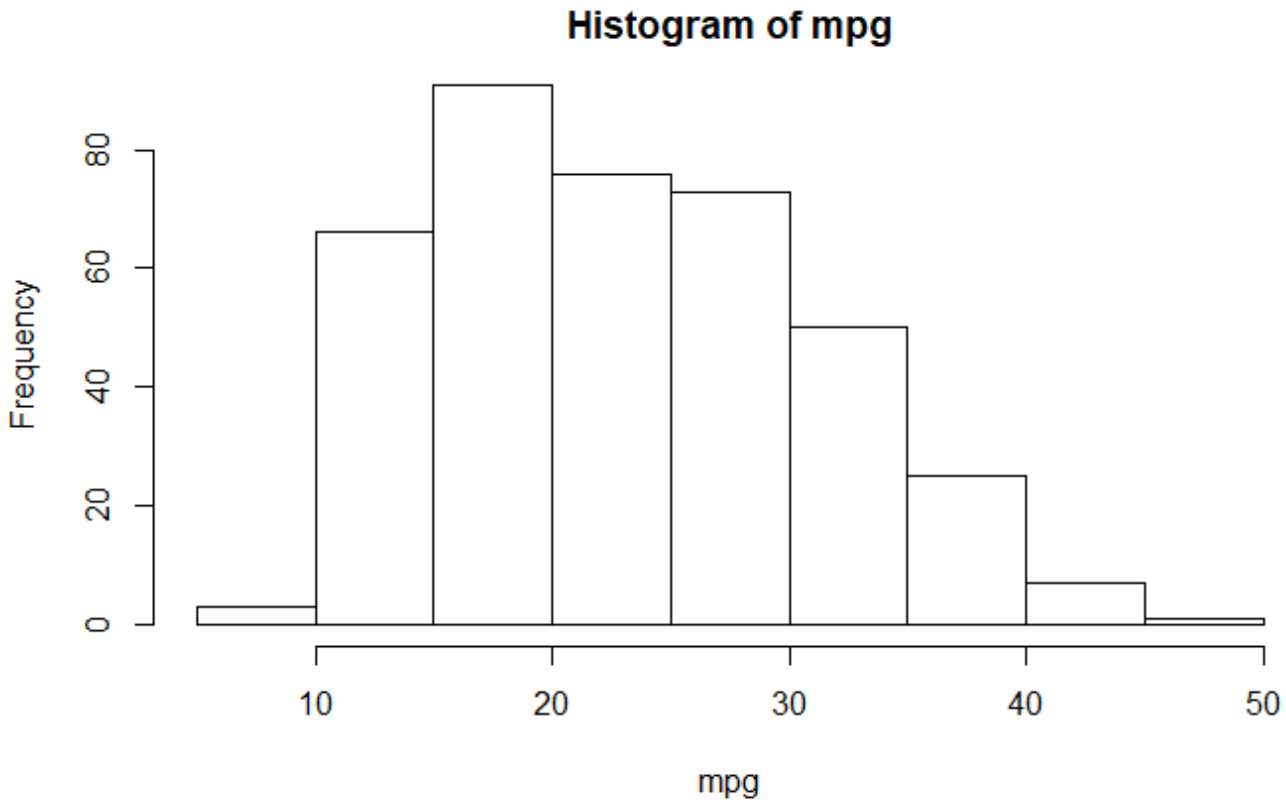
Hide



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

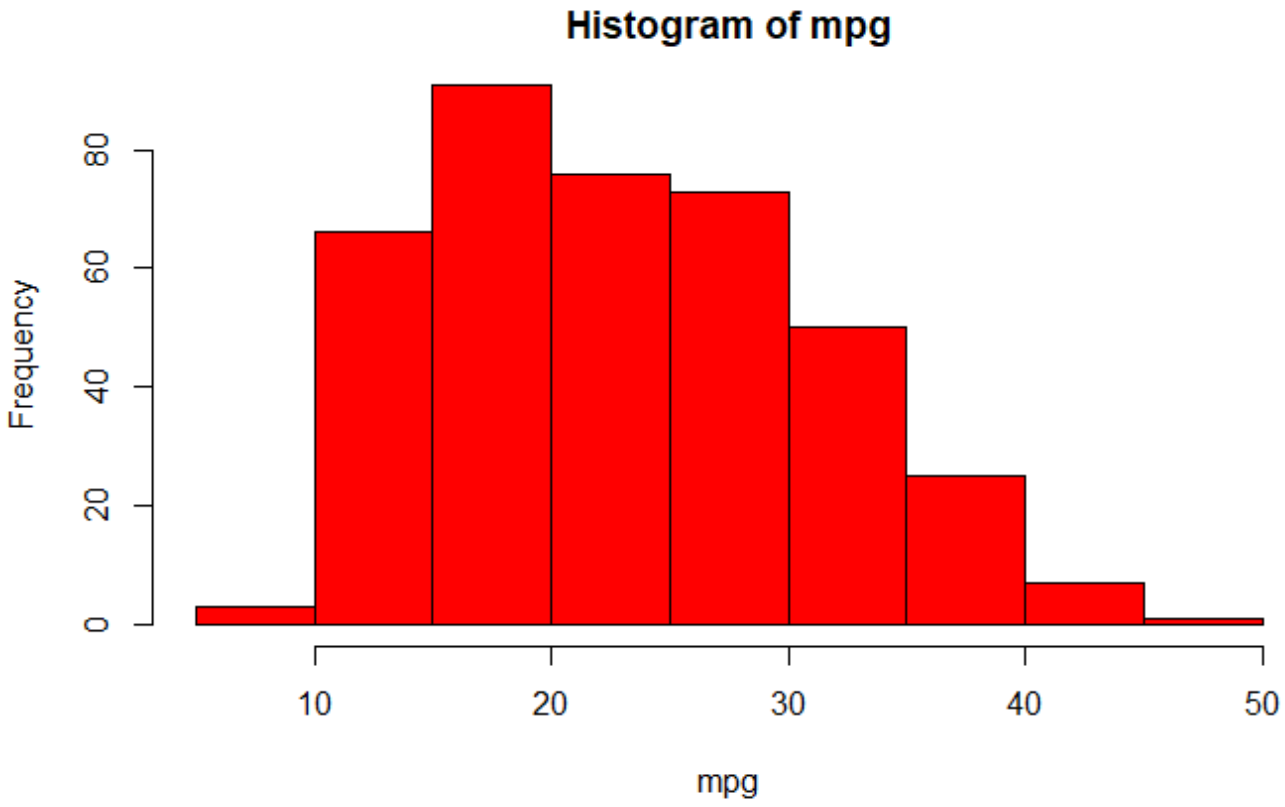
```
hist(mpg)
```

Hide



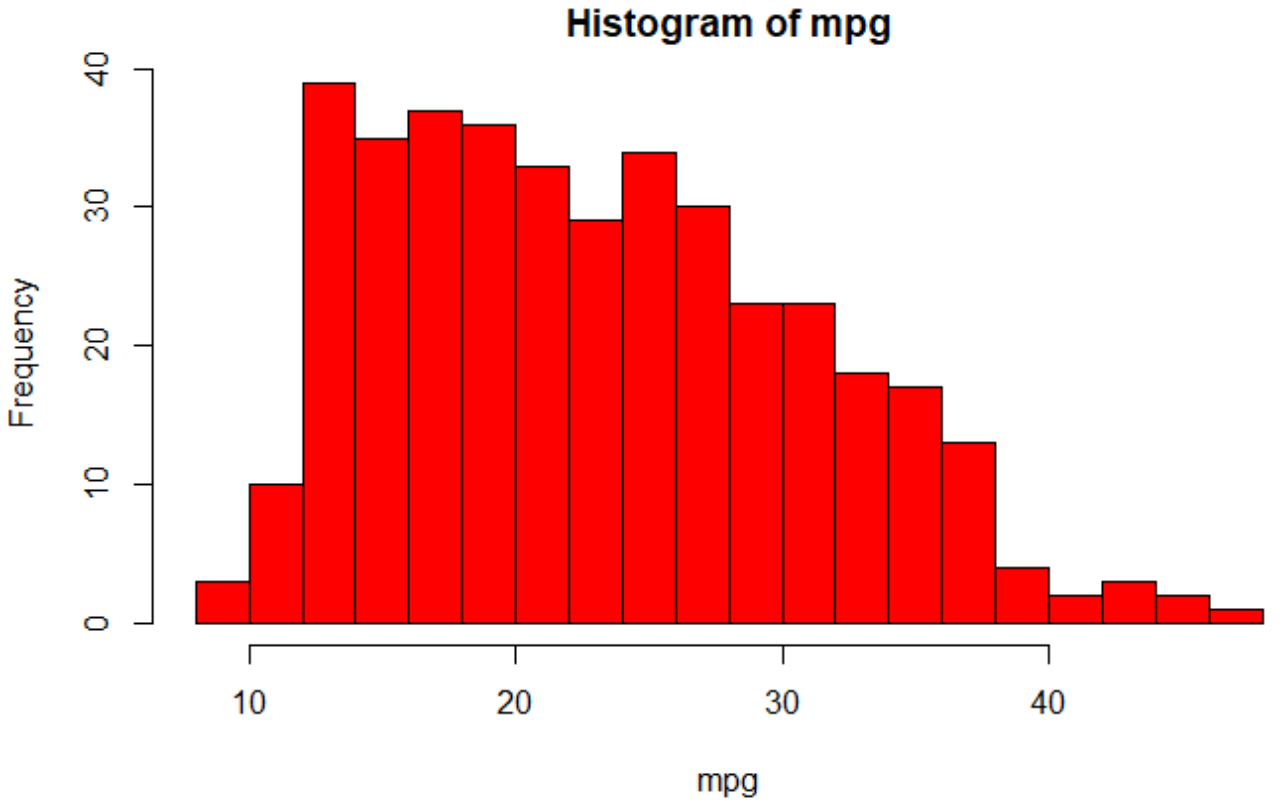

```
hist(mpg,col=2)
```

Hide




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

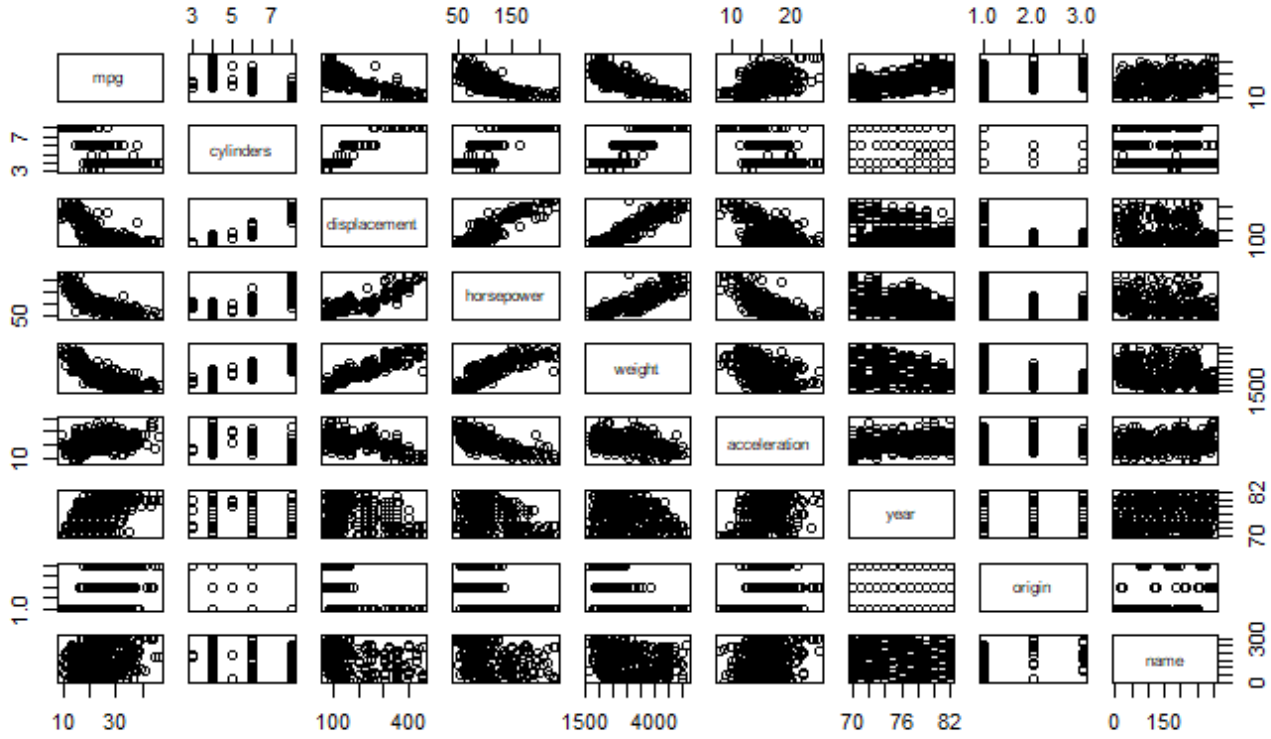
Hide



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.

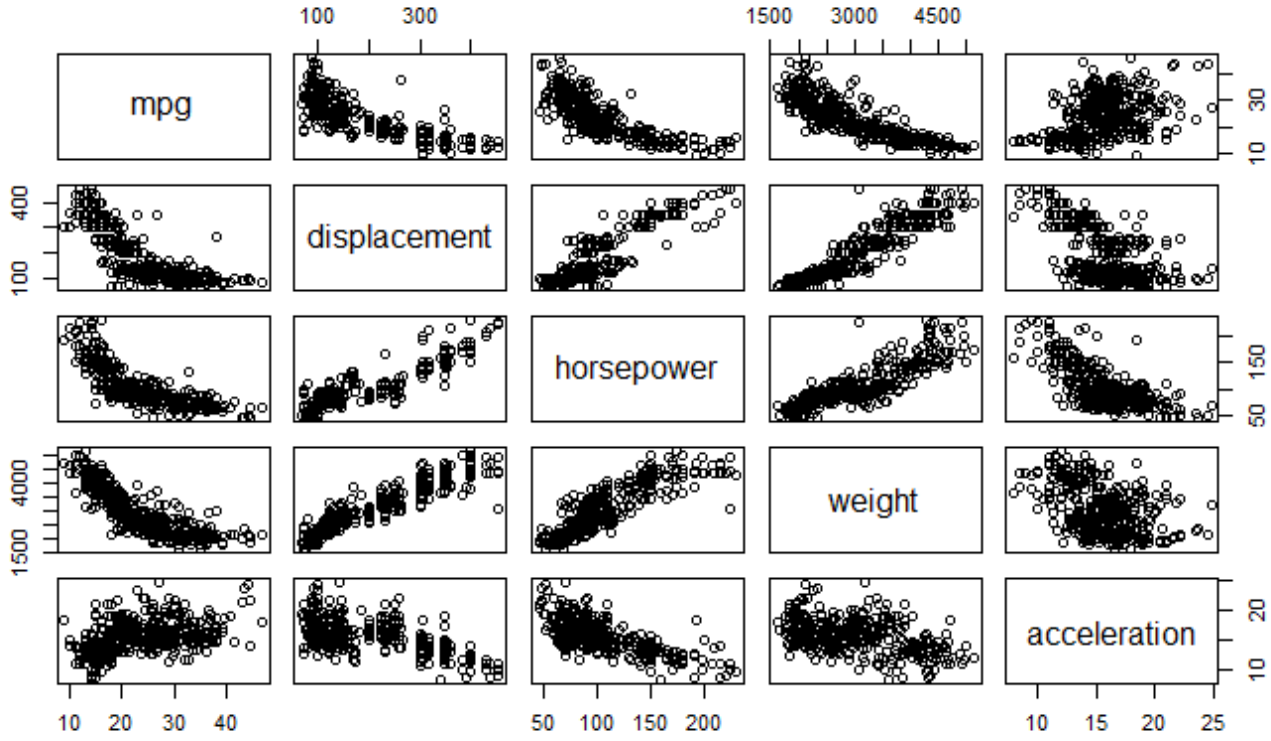
```
pairs(Auto)
```

Hide




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

Hide

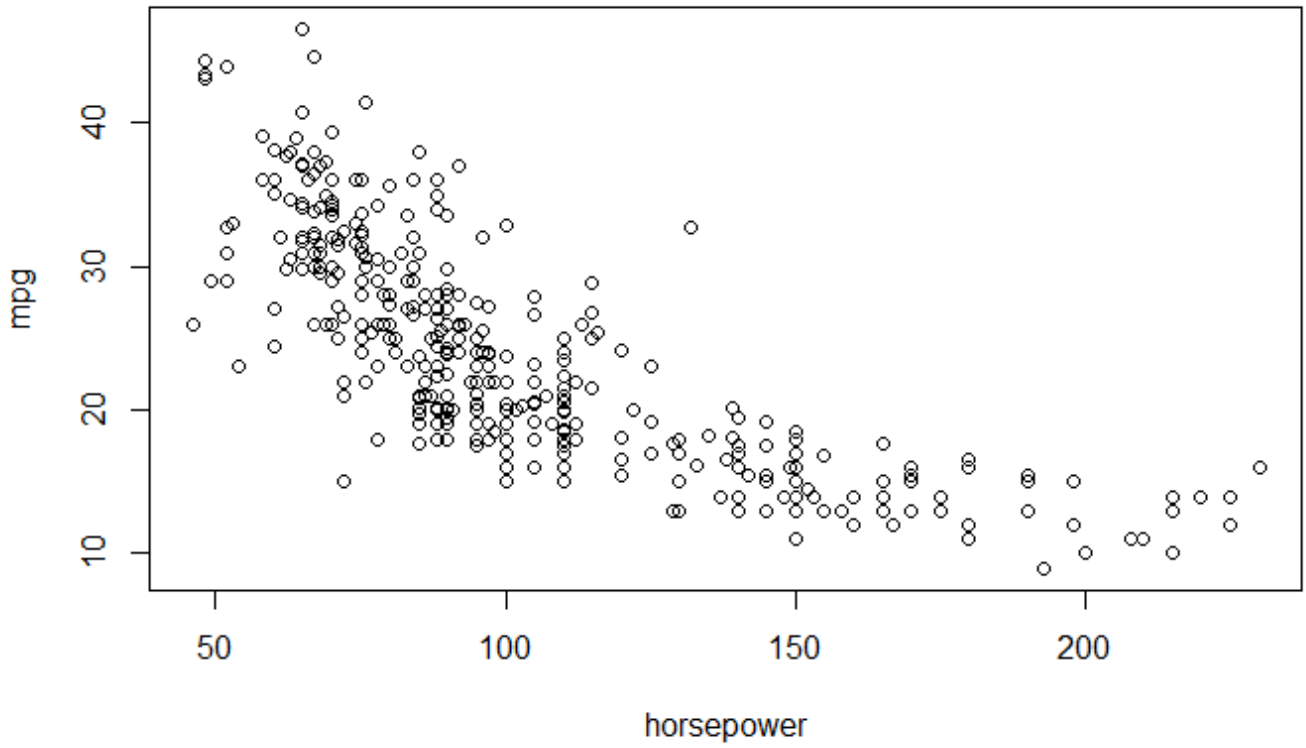


In conjunction with the `plot()` function, `identify()` provides a useful interactive method for identifying the value for a particular variable for points on a plot. We pass in three arguments to `identify()` : the *x*-axis variable, the *y*-axis variable, and the variable whose values we would like 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 (control-click on a Mac). The numbers printed under the identify() function correspond to the rows for the selected points.

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

Hide

```
integer(0)
```



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

```
summary(Auto)
```

Hide

```
      mpg          cylinders      displacement     horsepower        weight      acceleration
 Min.   : 9.00   Min.   :3.000   Min.   : 68.0   Min.   : 46.0   Min.   :1613   Min.   : 8.00
 1st Qu.:17.00   1st Qu.:4.000   1st Qu.:105.0   1st Qu.: 75.0   1st Qu.:2225   1st Qu.:13.78
 Median :22.75   Median :4.000   Median :151.0   Median : 93.5   Median :2804   Median :15.50
 Mean   :23.45   Mean   :5.472   Mean   :194.4   Mean   :104.5   Mean   :2978   Mean   :15.54
 3rd Qu.:29.00   3rd Qu.:8.000   3rd Qu.:275.8   3rd Qu.:126.0   3rd Qu.:3615   3rd Qu.:17.02
 Max.   :46.60   Max.   :8.000   Max.   :455.0   Max.   :230.0   Max.   :5140   Max.   :24.80

      year           origin                      name
 Min.   :70.00   Min.   :1.000   amc matador       :  5
 1st Qu.:73.00   1st Qu.:1.000   ford pinto        :  5
 Median :76.00   Median :1.000   toyota corolla    :  5
 Mean   :75.98   Mean   :1.577   amc gremlin       :  4
 3rd Qu.:79.00   3rd Qu.:2.000   amc hornet        :  4
 Max.   :82.00   Max.   :3.000   chevrolet chevette:  4
                                 (Other)           :365
```

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

```
summary(mpg)
```

Hide

```
   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 of the commands that we typed in the most recent session; this can be accomplished using the `savehistory()` function. Next time we enter `R`, `savehistory()` we can load that history using the `loadhistory()` function.