Introduction to R and R Markdown

Wendong Li

Last compiled on 09/26/2022

Contents

R Markdown

R Markdown provides an authoring framework for data science. You can use a single R Markdown file to both

- save and execute code
- generate high quality reports that can be shared with an audience

R Markdown was designed for easier reproducibility, since both the computing code and narratives are in the same document, and results are automatically generated from the source code. R Markdown supports dozens of static and dynamic/interactive output formats.

A video introduction to R Markdown is on the website https://rmarkdown.rstudio.com. You can watch the videos in the "Get Started" section, which cover the basics of R Markdown. An R Markdown cheatsheet is on the website https://www.rstudio.com/resources/cheatsheets/.

An R Markdown file usually consists of three elements: **metadata**, **inline texts** and **codes**.

Metadata

The metadata is written between the pair of three dashes. The syntax for the metadata is YAML (YAML Ain't Markup Language), so sometimes it is also called the YAML metadata.

One simple example:

```yaml
---
title: "Hello R Markdown"
author: "Awesome Me"
date: "2018-02-14"
output: html_document
---
```

Each output format is often accompanied with several format options. All these options are documented on the R package help pages. For example, you can type `?rmarkdown::html_document` in R to open the help page of the `html_document` format. I must warn you in advance that indentation matters in YAML, so do not forget to indent the sub-fields of a top field properly.

To illustrate, the metadata of this file is as follows:

```yaml
---
title: "Introduction to R and R Markdown"
author: "Wendong Li"
date: "Last compiled on 09/26/2022"
output:
  html_document:
    toc: TRUE
    toc_float:
      collapsed: TRUE
      smooth_scroll: TRUE
    number_sections: TRUE
---
```

- `toc: TRUE` includes a table of contents in the output
- `toc_float` floats the table of contents to the left of the main document content. Rather than TRUE, you may also pass a list of options that control the behavior of the floating table of contents
  - `collapsed` controls whether the table of contents appears with only the top-level headers
  - `smooth_scroll` controls whether page scrolls are animated when table of contents items are navigated to via mouse clicks
- `number_sections: TRUE` numbers section headings
- ……

Inline Text

The text in an R Markdown document is written with the Markdown syntax.

Inline text will be *italic* if surrounded by underscores or asterisks, e.g., `_text_` or `*text*`. **Bold** text is produced using a pair of double asterisks (`**text**`). A pair of tildes (`~`) turn text to a subscript (e.g., `H~3~PO~4~` renders H_3PO_4). A pair of carets (`^`) produce a superscript (e.g., `Cu^2+^` renders Cu^{2+}).

To mark text as `inline code`, use a pair of backticks, e.g., `` `code` ``. To include n literal backticks, use at least $n + 1$ backticks outside, e.g., you can use four backticks to preserve three backtick inside: ````` ```` ```code``` ```` `````, which is rendered as ```` ```code``` ````.

Hyperlinks are created using the syntax `<link>` or `[text](link)`. For example, `<https://www.rstudio.com>` and `[RStudio](https://www.rstudio.com)` render https://www.rstudio.com and RStudio respectively.

Code

Code Chunks

You can insert an R code chunk either using the RStudio toolbar (the `Insert` button) or the keyboard shortcut `Ctrl + Alt + I` (`Cmd + Option + I` on macOS). When you render your .Rmd file, R Markdown will run each code chunk and embed the results beneath the code chunk in your final report. There are a lot of things you can do in a code chunk: you can produce text output, tables, or graphics. For example:

```
text <- "hello world!"
text
```

```
## [1] "hello world!"
```

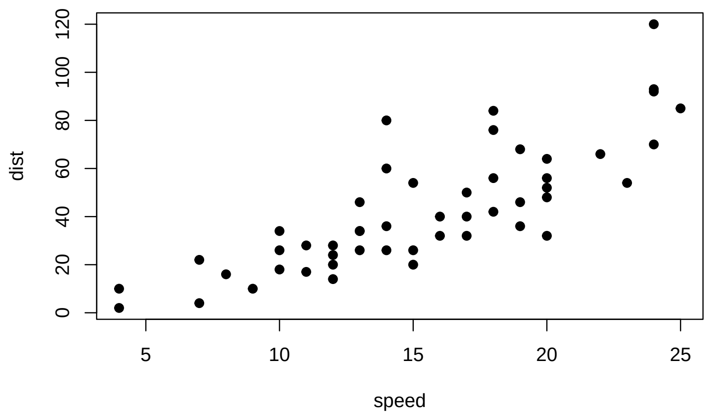
```
x <- 1
x
```

```
## [1] 1
```

```
knitr::kable(iris[1:5, ])
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa

```
plot(cars, pch = 19)
```



Chunk output can be customized with chunk options, arguments set in the {} of a chunk header. For example:

- `eval`: Whether to evaluate a code chunk.
- `echo`: Whether to echo the source code in the output document (someone may not prefer reading your smart source code but only results).
- `warning`, `message`, and `error`: Whether to show warnings, messages, and errors in the output document.
- `include`: Whether to include anything from a code chunk in the output document. If `include=FALSE`, R Markdown still runs the code in the chunk, and the results can be used by other chunks.
- ……

To set global options that apply to every chunk in your file, call `knitr::opts_chunk$set` in a code chunk. Knitr will treat each option that you pass to `knitr::opts_chunk$set` as a global default that can be overwritten in individual chunk headers.

Inline Code

Code results can be inserted directly into the text of a .Rmd file by enclosing the code with `` `r ` ``.

R Markdown will always

- display the results of inline code, but not the code
- apply relevant text formatting to the results

For example 1+1 equals 2.

Exercise

- Knit your first R Markdown file!
- Make a CV using R Markdown that involves your education, interests, future plan and what you expect to learn from this class. Feel free to add more sections and use more R markdown skills even they are not introduced. English and are both acceptable, but make sure to only use one of them :)

R

Functions

R uses functions to perform operations. With functions, you can almost do anything you want.

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.

Some functions are in preloaded packages and thus can be directly used. Some functions can only be used after you install and load the corresponding packages with `install.packages()` and `library()`.

You can also write your own functions in order to make repetitive operations. For example:

```r
add1 <- function(a=1,b=2){
  c <- a+b
  return(c)
}
add1()
## [1] 3
add1(a=5,b=8)
## [1] 13
```

Note: We could just as well omit typing `a=` and `b=` in the `add1()` command above: that is, we could just type `add1(5,8)`, 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.

```
add1(5,8)
## [1] 13
add1(b=8,a=12)
## [1] 20
add1(a=1,b=2,g=3)
## Error in add1(a = 1, b = 2, g = 3):    (g = 3)
```

Data Types and Structures

To make the best of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on them.

The three most commonly used data types:

- character: `"RStudio"`, `"R is so good!"`
- numeric: `24`, `24.8`
- logical: `TRUE`, `FALSE`
- factor: to represent categorical data. We will introduce this later.

```
1==1
## [1] TRUE
1!=1
## [1] FALSE
1>=4
## [1] FALSE
4<8
## [1] TRUE
"Rstudio"=="R"
## [1] FALSE
1-(2==2)   # In numerical calculation, TRUE is equivalent to 1.
## [1] 0
1*(1!=1)
## [1] 0
```

Elements of these data types may be combined to form data structures, including: `vector`, `matrix`, `data frame`, `list`.

Vector

A vector is the most common and basic data structure in R. You can create vectors with `logical()`, `character()` and `numeric()`.

```
logical(5)
## [1] FALSE FALSE FALSE FALSE FALSE
character(5)
```

```
## [1] "" "" "" "" ""
numeric(5)
## [1] 0 0 0 0 0
```

More commonly, you can also create vectors by directly specifying their content. R will then guess the appropriate mode of storage for the vector. The function is `c()` (for *combine*). Any numbers inside the parentheses are joined together.

```
x <- c(1, 2, 3)
x
## [1] 1 2 3
y <- c(TRUE, TRUE, FALSE, FALSE)
y
## [1]  TRUE  TRUE FALSE FALSE
z <- c("  ", "  ")
z
## [1] "  " "  "
```

`c()` can also be used to add elements to a vector.

```
z <- c(z, "  ")
z
## [1] "  " "  " "  "
z <- c("  ", z)
z
## [1] "  " "  " "  " "  "
```

You can create vectors as a sequence of numbers with `seq()` and `rep()`.

```
x <- 1:10
x
##  [1]  1  2  3  4  5  6  7  8  9 10
y <- seq(1,10)
y
##  [1]  1  2  3  4  5  6  7  8  9 10
z <- seq(from = 1, to = 2, by = 0.1)
z
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
z1 <- rep(0, 10)
z1
##  [1] 0 0 0 0 0 0 0 0 0 0
```

R supports missing data in vectors. They are represented as NA (Not Available) and can be used for all the vector types. The function `is.na()` indicates the elements of the vectors that represent missing data, and the function `anyNA()` returns TRUE if the vector contains any missing values:

```
x <- c("a", NA, "c", "d", NA)
is.na(x)
## [1] FALSE  TRUE FALSE FALSE  TRUE
anyNA(x)
## [1] TRUE
```

Elements from a vector, matrix, or data frame can be extracted using numeric indexing. There are many ways to do so:

```
x <- c(1,4,4,NA,2,2,3)
x[1]
## [1] 1
x[1:3]
## [1] 1 4 4
x[c(5,7)]
## [1] 2 3
x[-7]
## [1]  1  4  4 NA  2  2
x1 <- x[!is.na(x)]
x1
## [1] 1 4 4 2 2 3
x1[x1==4]
## [1] 4 4
```

Some useful vector functions:

```
which(x1==4)
## [1] 2 3
sort(x1)
## [1] 1 2 2 3 4 4
order(x1)
## [1] 1 4 5 6 2 3
rev(x1)
## [1] 3 2 2 4 4 1
unique(x1)
## [1] 1 4 2 3
table(x1)
## x1
## 1 2 3 4
## 1 2 1 2
```

Factors: Factors are used to represent categorical data. Factors can be ordered or unordered and are important for statistical analysis and for plotting.

While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

Once created, factors can only contain a pre-defined set values, known as levels. By default, R always sorts levels in alphabetical order.

The factor() command is used to create and modify factors in R:

```
sex <- factor(c("male", "female", "female", "male"))
sex
```

```
## [1] male   female female male
## Levels: female male
```

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., “low”, “medium”, “high”) or it is required by particular type of analysis. Additionally, specifying the order of the levels allows us to compare levels:

```
food <- factor(c("low", "high", "medium", "high", "low", "medium", "high"))
levels(food)
## [1] "high"   "low"    "medium"
food <- factor(food, levels = c("low", "medium", "high"))
levels(food)
## [1] "low"    "medium" "high"
```

```
min(food)
```

```
## Error in Summary.factor(structure(c(1L, 3L, 2L, 3L, 1L, 2L, 3L), levels = c("low", : 'min' not meani
```

```
food <- factor(food, levels = c("low", "medium", "high"), ordered = TRUE)
min(food)
```

```
## [1] low
## Levels: low < medium < high
```

Generate random numbers: The `rnorm()` function generates a vector of random normal variables, with first argument `n` the sample size.

```
x <- rnorm(5)
x
```

```
## [1] 1.4706251 1.9687998 0.5732563 0.7462188 0.3664677
```

Each time we call this function, we will get a different answer. Sometimes we want our code to reproduce the exact same set of random numbers. We can use the `set.seed()` function to do this.

```
set.seed(2022)
x <- rnorm(5)
set.seed(NULL)
x
```

```
## [1]  0.9001420 -1.1733458 -0.8974854 -1.4445014 -0.3310136
```

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.

```
x <- rnorm(5, mean = 10, sd=2)
x
```

```
## [1]  8.246581 13.773308  9.384343 10.238739  5.244495
```

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.

```
y <- rnorm(100)
mean(y)
## [1] -0.1749701
var(y)
## [1] 0.9021592
sqrt(var(y))
## [1] 0.9498206
sd(y)
## [1] 0.9498206
```

Similarly, you can generate random numbers from other distributions with `rt()`, `runif()`, etc.

Matrix

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

```
A <- matrix(data = 1:16, nrow = 4, ncol = 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
```

Note: As discussed earlier, we could just omit typing `data=`, `nrow=`, and `ncol=` and type `matrix(1:16, 4, 4)`, which would have the same effect.

As this example illustrates, by default R creates matrices by filling in columns. Alternatively, the `byrow = TRUE` option can be used to populate the matrix in order of the rows.

```
A <- matrix(1:16, 4, 4, byrow = TRUE)
A
```

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

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

```
dim(A)
```

```
## [1] 4 4
```

To index the elements in A, typing `A[2,3]` 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.

```
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
A[2,3]
## [1] 10
A[1,]
## [1]  1  5  9 13
A[,1]
## [1] 1 2 3 4
```

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
A[-c(1, 3), ]
##      [,1] [,2] [,3] [,4]
## [1,]    2    6   10   14
## [2,]    4    8   12   16
```

Functions that can be applied to scalar are generally can be applied to vector/matrix. For example, 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.

```
x <- matrix(c(1, 2, 3, 4), 2, 2, byrow = TRUE)
sqrt(x)
##          [,1]     [,2]
## [1,] 1.000000 1.414214
## [2,] 1.732051 2.000000
x^2
##      [,1] [,2]
## [1,]    1    4
## [2,]    9   16
```

```
x+2
##      [,1] [,2]
## [1,]    3    4
## [2,]    5    6
t(x)
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

List

In R lists act as containers. The contents of a list are not restricted to a single mode and can encompass any mixture of data types.

Create lists using `list()`:

```
x <- list(1, "a", TRUE)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
```

Vectors can also be transformed to lists with `as.list()`:

```
x <- 5:7
x <- as.list(x)
x
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 7
```

Elements of a list can be named

```
x <- list(name = "Wendong Li", data=c(28,177))
x
```

```
## $name
## [1] "Wendong Li"
##
## $data
## [1]  28 177
```

Note: Lists can be extremely useful inside functions. Because the functions in R are able to return only a single object, you can "staple" together lots of different kinds of results into a single object that a function can return.

Elements are indexed by double brackets. If the elements of a list are named, they can be referenced by the $ notation (i.e. `x$data`).

```
x[[1]]
## [1] "Wendong Li"
x$data
## [1]  28 177
x$data[1]
## [1] 28
```

Data Frame

A data frame is a very important data type in R. It's pretty much the data structure for most tabular data and what we use for statistics.

A data frame is a special type of list where every element of the list has same length (i.e. data frame is a "rectangular" list).

Some additional information on data frames:

- Usually created by `read.csv()` and `read.table()`, i.e. when importing the data into R.
- Assuming all columns in a data frame are of same type, data frame can be converted to a matrix with `as.matrix()`.
- Can also create a new data frame with `data.frame()` function.
- Rownames are often automatically generated and look like `1, 2,..., n`.

```
dat <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)
dat
```

```
##    id  x  y
## 1   a  1 11
## 2   b  2 12
## 3   c  3 13
## 4   d  4 14
## 5   e  5 15
## 6   f  6 16
## 7   g  7 17
## 8   h  8 18
## 9   i  9 19
## 10  j 10 20
```

Useful data frame functions:

- `head()` - shows first 6 rows
- `tail()` - shows last 6 rows
- `dim()` - returns the dimensions of data frame (i.e. number of rows and number of columns)
- `nrow()` - number of rows
- `ncol()` - number of columns

```
head(dat)
##   id x  y
## 1  a 1 11
## 2  b 2 12
## 3  c 3 13
## 4  d 4 14
## 5  e 5 15
## 6  f 6 16
dim(dat)
## [1] 10  3
nrow(dat)
## [1] 10
ncol(dat)
## [1] 3
```

As data frames are also lists, it is possible to refer to columns (which are elements of such list) using the list notation, i.e. either double square brackets or a $.

```
dat[[1]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
dat$id
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
dat[,1]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
dat[1,]
##   id x  y
## 1  a 1 11
```

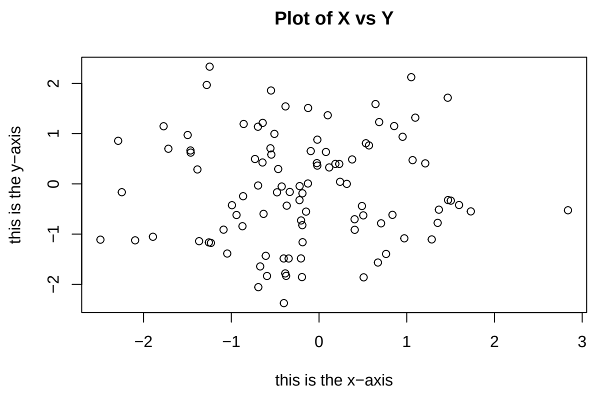
Exercise

- create the vector `1,1,1,1,1,2,2,2,2,2` with only `rep()` and name it `x1`.
- create the vector `1,2,1,2,1,2,1,2,1,2` with only `rep()` and name it `x2`.
- combine `x1` and `x2` into a matrix `x.col` by columns, i.e., `x1` and `x2` are the two columns of `x`. Hint: use `cbind()`.
- combine `x1` and `x2` into a matrix `x.row` by rows, i.e., `x1` and `x2` are the two rows of `x`. Hint: use `rbind()`.
- find two ways to calculate the sum of each column of `x.row`. Hint: use `apply()`.

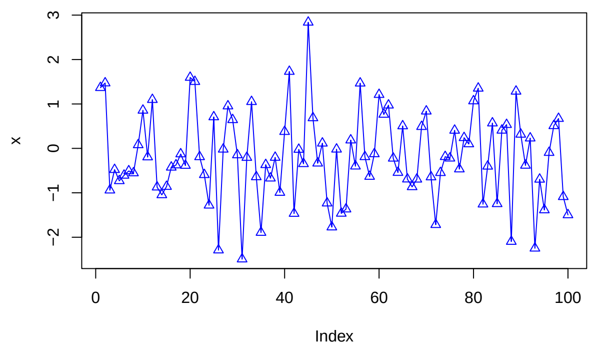
Graphics

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. For example, passing in the argument `xlab` will result in a label on the x-axis. To find out more information, type `?plot`.

```
x <- rnorm(100)
y <- rnorm(100)
plot(x, y, xlab = "this is the x-axis", ylab = "this is the y-axis", main = "Plot of X vs Y")
```

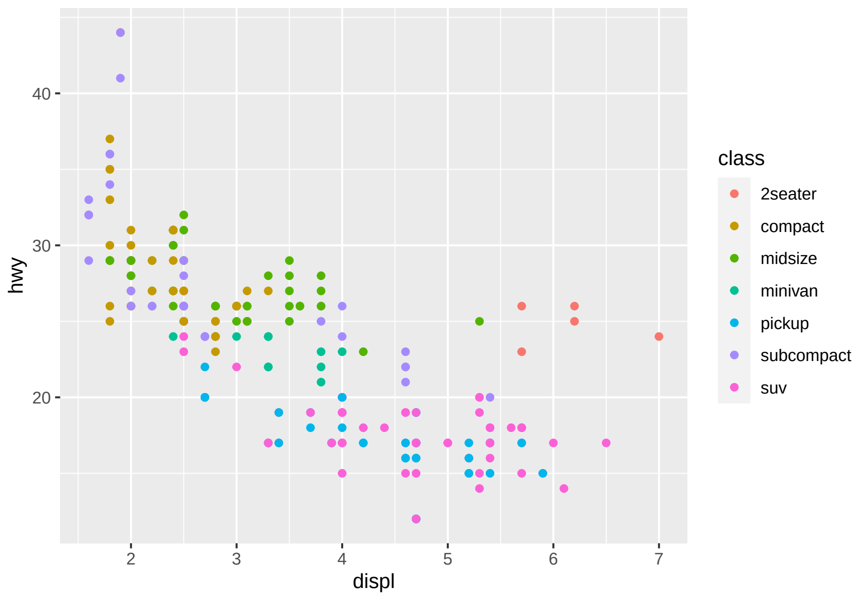


```
plot(x,type='o',pch=2,lty=1,col='blue')
```



A powerful package to create elegant plots is `ggplot2`. It understands and builds a plot by breaking down a plot into several layers. For example:

```
library(ggplot2)
ggplot(mpg, aes(displ, hwy, colour = class)) + geom_point()
```



We will not go into further details about ggplot2. If you are interested in ggplot2, I strongly recommend that you visit https://ggplot2.tidyverse.org/ and pay attention to the course “Data Visualization” taught by Prof. Xin Liu.

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 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. 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.

To illustrate the `read.table()` function, we load it now from a text file, `Auto.data`. The following command will load the file into R and store it as a **data frame** called *Auto*. The `head()` function can also be used to view the first few rows of the data.

```
Auto <- read.table("Auto.data")
Auto[c(1:5,34),]
```

```
##      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
## 34 25.0         4        98.00          ?  2046.         19.0   71      1
##                         V9
## 1                     name
## 2  chevrolet chevelle malibu
## 3          buick skylark 320
## 4         plymouth satellite
## 5               amc rebel sst
## 34                 ford pinto
```

Something is not right about the data frame Auto. Have you noticed?

R has assumed that the variable names are part of the data and so has included them in the first row. Using the option `header = TRUE` in the `read.table()` function tells R that the first line of the file contains the variable names.

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 `na.strings` (NA means not available) 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 = "?", stringsAsFactors = TRUE)
Auto[c(1:4,33),]
```

```
##    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
## 33  25         4           98         NA   2046         19.0   71      1
##                          name
## 1   chevrolet chevelle malibu
## 2           buick skylark 320
## 3          plymouth satellite
## 4               amc rebel sst
## 33                 ford pinto
```

`stringsAsFactors = TRUE` tells R that any variable containing character strings should be interpreted as a qualitative variable, and that each distinct character string represents a distinct level for that qualitative variable.

An easy way to load data from Excel into R is to save it as a csv (comma-separated values) file, and then use the `read.csv()` function.

```
Auto <- read.csv("Auto.csv", header=TRUE, na.strings = "?", stringsAsFactors = TRUE)
head(Auto)
```

```
##   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
## 5  17         8          302        140   3449         10.5   70      1
## 6  15         8          429        198   4341         10.0   70      1
##                        name
## 1 chevrolet chevelle malibu
## 2         buick skylark 320
## 3        plymouth satellite
## 4             amc rebel sst
## 5               ford torino
## 6          ford galaxie 500
```

```
dim(Auto)
```

```
## [1] 397   9
```

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 contain missing observations, and so we choose to use 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"   "weight"
## [6] "acceleration" "year"         "origin"       "name"
```

Additional Graphical and Numerical Summaries

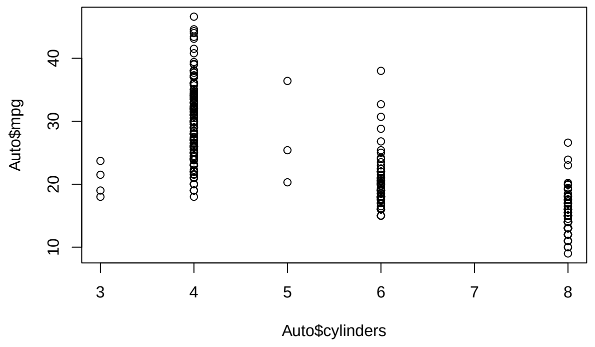
We can use the plot() function to produce scatterplots of the quantitative variables. However, simply typing the variable names will produce an error message, because R does not know to look in the Auto data set for those variables.

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

```
## Error in plot(cylinders, mpg):    'cylinders'
```

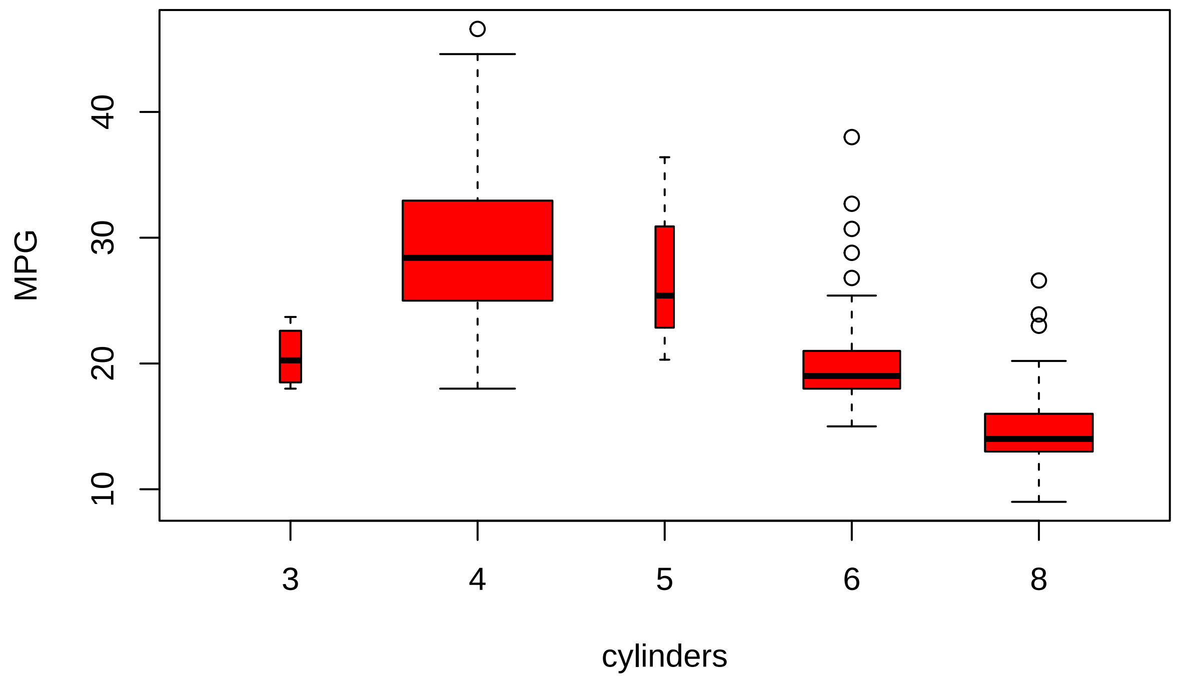
To refer to a variable, we must type the data set and the variable name joined with a $ symbol.

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

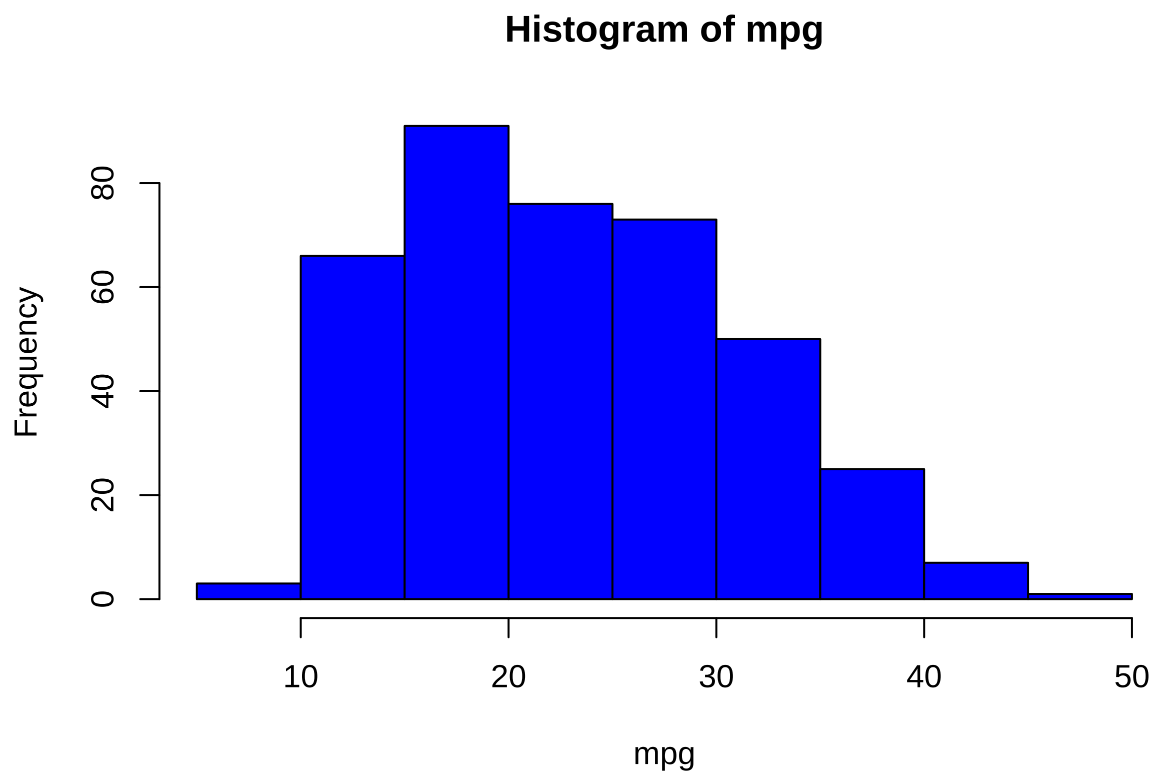


The cylinders variable is stored as a numeric vector, so R has treated it as quantitative. However, since there are only a small number of possible values for cylinders, one may prefer to treat it as a qualitative variable. The `as.factor()` function converts quantitative variables into qualitative variables.

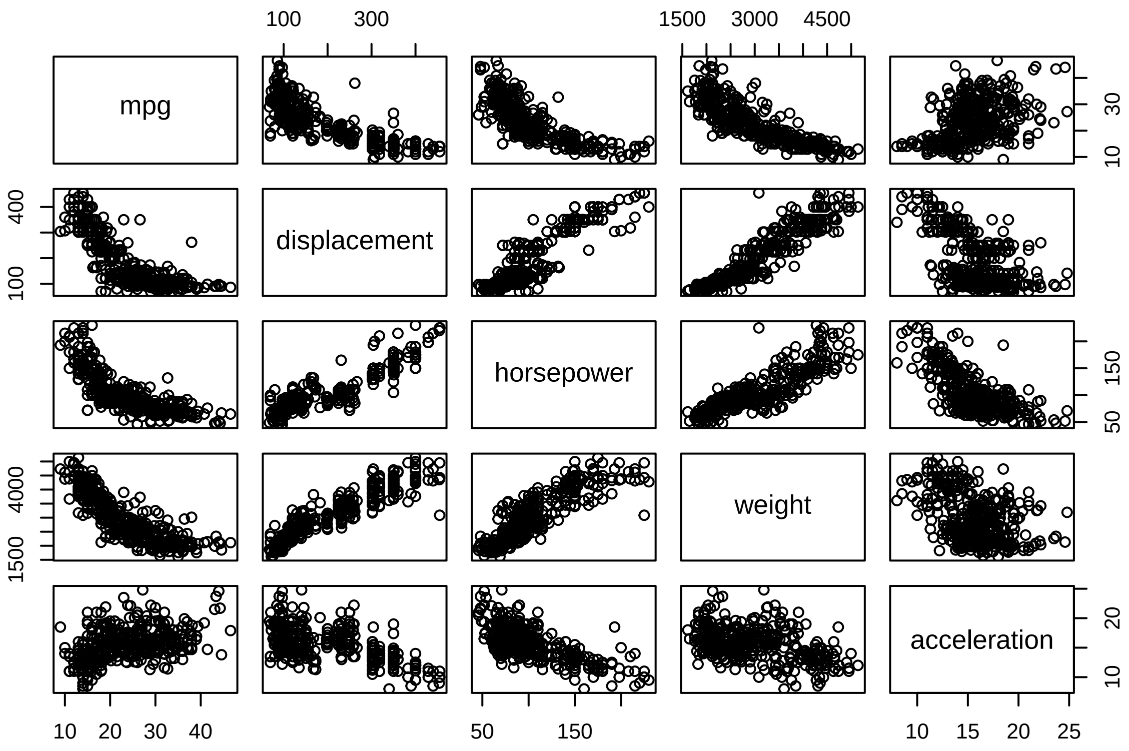
```
attach(Auto)
cylinders <- as.factor(cylinders)
plot(cylinders, mpg, col = "red", varwidth = TRUE, xlab = "cylinders", ylab = "MPG")
```



```
hist(mpg, col = "blue")
```



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



```
summary(Auto)
```

```
##       mpg          cylinders      displacement     horsepower        weight    
##  Min.   : 9.00   Min.   :3.000   Min.   : 68.0   Min.   : 46.0   Min.   :1613  
##  1st Qu.:17.00   1st Qu.:4.000   1st Qu.:105.0   1st Qu.: 75.0   1st Qu.:2225  
##  Median :22.75   Median :4.000   Median :151.0   Median : 93.5   Median :2804  
##  Mean   :23.45   Mean   :5.472   Mean   :194.4   Mean   :104.5   Mean   :2978  
##  3rd Qu.:29.00   3rd Qu.:8.000   3rd Qu.:275.8   3rd Qu.:126.0   3rd Qu.:3615  
##  Max.   :46.60   Max.   :8.000   Max.   :455.0   Max.   :230.0   Max.   :5140  
##                                                                                
##   acceleration        year           origin                      name    
##  Min.   : 8.00   Min.   :70.00   Min.   :1.000   amc matador       :  5  
##  1st Qu.:13.78   1st Qu.:73.00   1st Qu.:1.000   ford pinto        :  5  
##  Median :15.50   Median :76.00   Median :1.000   toyota corolla    :  5  
##  Mean   :15.54   Mean   :75.98   Mean   :1.577   amc gremlin       :  4  
##  3rd Qu.:17.02   3rd Qu.:79.00   3rd Qu.:2.000   amc hornet        :  4  
##  Max.   :24.80   Max.   :82.00   Max.   :3.000   chevrolet chevette:  4  
##                                                  (Other)           :365
```

Exercise

3. This exercise involves the Boston housing data set. To begin, load in the Boston data set. The Boston data set is part of the `ISLR2` library. Of course, you should install the `ISLR2` package before using it. After `library(ISLR2)`, the data set is contained in the object `Boston`. Read about the data set with `?Boston`.

- How many rows are in this data set? How many columns? What do the rows and columns represent?
- Which of the predictors are quantitative, and which are qualitative?
- What is the range of each quantitative predictor? You can answer this using the `range()` function.
- Make some pairwise scatterplots of the predictors (columns) in this data set. Describe your findings.
- Are any of the predictors associated with per capita crime rate? If so, explain the relationship.
- What is the mean and standard deviation of each quantitative predictor?
- How many of the census tracts in this data set bound the Charles river?
- What is the median pupil-teacher ratio among the towns in this data set?
- Which census tract of Boston has lowest median value of owner-occupied homes? What are the values of the other predictors for that census tract, and how do those values compare to the overall ranges for those predictors? Comment on your findings.
- In this data set, how many of the census tracts average more than seven rooms per dwelling? More than eight rooms per dwelling? Comment on the census tracts that average more than eight rooms per dwelling.