# MSDS 401 Quick Start Guide for R

## Table of Contents

# MSDS 401 Quick Start Guide for R

## 1 Why R?

**R** provides a wide range of statistical methods and the capability to develop programs tailored to your needs. It is an open source package designed for the manipulation, analysis, description, and visual presentation of data. It is command-based, not menu-driven. There is a large user community and thousands of available packages with more being added all the time. **R** is free of cost

New users who have limited experience with other programming languages face a steep learning curve. You will need to use available resources such as textbooks, other students, the internet, and instructors. No single comprehensive user's guide to R exists. Trial and error will be necessary. **R** will provide guidance in the form of error messages. These messages may not be complete, and may require research. This document provides an introduction to the **R** language. It is not comprehensive by any means. It addresses basics of data manipulation, computation and data visualization. Use this document as an introductory workbook. Examples and solved problems are provided.

## 2 Installing R

R may be downloaded freely at R Project. Everything required by this introduction can be accomplished in the **R** environment with a plain text editor. However, many users choose to work in RStudio, an integrated development environment (IDE) for R. RStudio is used for the examples in this document. It is convenient and easy to use. **You are encouraged to install the most current version of R and RStudio.**

**Step 1**: Download and install **R** (**R** GUI) from R Project.. Documentation is readily available on CRAN and on many **R** community and resource sites.

**Step 2**: Download and install RStudio.

There are many facilities in **R** and thousands of software packages that can be downloaded. Typing > *library()* will give a list and short description of the libraries available. Typing > *library(package)* where "package" is the name of the required library, will give you access to the functions in that library. Packages may be easily downloaded using RStudio.

---

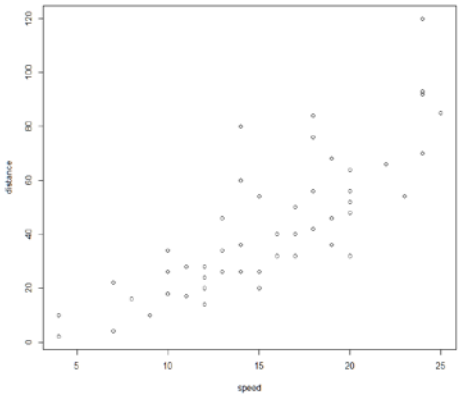
In this document, when the prompt symbol > appears in an example, that example was copied from output in the **R** console. The > sign is not typed in **R** script or the console. The symbol <- is used in **R** script to assign values to an object as the following examples will show. Most examples in this document are copied directly from the **R** console following execution of the indicated code so that the results are displayed. Executed code appears using a blue font.

## 3 Getting Started with R

When you start a session in **R** or **RStudio** a workspace, the current **R** working environment, is initialized. Objects defined during the session are stored in this workspace. Commands may be entered interactively in the console at the prompt, ">." It is also possible to write and execute the code from **R** script (.R). **R** is case-sensitive. Commands written in **R** are stored as history for at least the duration of the session. You can scroll back to previous commands typed by using the up arrow key (and down to scroll back again). You can also copy and paste using standard windows editor techniques (for example, using the copy and paste dialog buttons). As an alternative, you might copy and paste commands manually into a text editor or something similar. At the end of an **R** session, you can save an image of the current workspace that is automatically reloaded the next time **R** is started. However, it is far preferable to maintain your work in R script. R code, order, and comments are preserved; scripts are transparent, editable and detailed in a way that workspace images are not.

Here is a sample program that was executed. The results appear along with the code. Phrases that appear after the # sign are not executed. These phrases appear to explain each program step.

```
> data(cars) # data() loads the specified dataset.
> mydata <- cars # assigns the "cars" dataset to the name "mydata"
> str(mydata) # returns the "structure" of the "mydata" object
'data.frame':          50 obs. of 2 variables:
 $ speed: num 4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num 2 10 4 22 16 10 18 26 34 17 ...
> speed <- mydata$speed # assigns the values of the variable "speed" to a vector "speed"
> distance <- mydata$dist # assigns the values of the variable "dist" to a vector "distance"
> mean(speed) # returns the arithmetic mean of the values of "speed"
[1] 15.4
> mean(distance) # returns the arithmetic mean of the values of "distance"
[1] 42.98
> plot(speed, distance) # plots distance (y) as a function of speed (x)
```



There are datasets stored with **R** that can be used. The dataset "cars" is one of these. This program called this dataset, and assigned it to an object called "mydata" forming a data frame. Most of the work in MSDS 401 will involve data frames. The *str()* function revealed mydata contained fifty observations of two variables: speed and dist. These data were assigned to separate vectors: speed and distance. Their mean values were calculated and a plot of distance versus speed was produced. This example illustrates that **R** stores information and operates on ***objects***. A data frame is an object, as are functions, vectors, lists, and matrices. Other types of objects can also be defined.

Learning **R** requires practice. Use this document as a workbook. Start a new **R** or **RStudio** session, and work through this document at the computer. Enter all of the commands in the following examples, making sure you understand how they operate. Then try the exercises at the end of each section. Solutions are provided separately.

## 4 Managing Your Workspace and Files

It is important to be aware of where your work is kept. It is possible to keep different projects in different physical directories. For the purposes of this guide, and subsequent course work, it is advisable to establish one working directory. Let's assume you name your working directory MSDS401. The following commands are examples that allow you to set this directory, verify the working directory, and list the contents. These and other examples can be found at StatMethods thanks to Rob Kabacoff, author of *R in Action.*

| | |
|---|---|
| setwd(MSDS401) | Change working directory to MSDS401. |
| setwd("c:/docs/MSDS401") | Specify location. Note / instead of \ as in Windows. |
| getwd() | Print the current working directory or cwd. |
| ls() | List the objects in the current workspace. |
| rm(list=ls()) | Removes all objects in the current workspace. |

When you work in **R,** objects created are stored in the current workspace. Each object created remains in the image unless you explicitly delete it. The workspace will be lost at the session's end unless you save it. Here are commands to help you manage your workspace. The abbreviation "cwd" stands for "current working directory".

```
save.image()        # Saves the workspace to the file .RData in the cwd.

save(object, file="mywork.RData")    # Saves "object" to the "mywork.RData" file.

load("mywork.RData")    # Loads a workspace into the current session from the cwd.

q() # Quit R. You will be prompted to save the workspace.
```

There are a variety of ways to import data into R depending on whether the file is from Excel, SAS, SPSS, etc. **In this course we will be dealing with comma-separated value files, denoted by .csv.** This is illustrated below which used the *read.table()* function to read-in a .csv into **R**. A special case of the *read.table()* command is the *read.csv()* command also shown below. Either can be used in this course. These are functions with arguments that need specification. The arguments are listed on the R documentations page which can be found by entering and executing *?read.table()* or *help(read.table)*. For example, R is informed the first row of a file contains variable names with "header = TRUE". Note the statements below:

```
mydata <- read.table("c:/mydata.csv", header = TRUE, sep = ",", row.names="id")

mydata <- read.csv(file.path("c:/R401/", "mydata.csv"), sep= ",")
```

At some point, you will need to save a data file. You can save it to your working directory using the first command shown below, or to some other file location by specifying the file.path using *file.path()*. These statements are intended for comma delimited files.

```
write.table(mydata, "mydata.csv")      # Saves to current directory

write.table(mydata, file.path("c:/R401/", "mydata.csv"))

write.csv(mydata, file.path("c:/R401/", "mydata.csv"))
```

## 5 Working with Vectors

What follows is a program executed in **R** that illustrates scalar arithmetic. This illustrates the values 5 and 6 are be added to return 11, and that these values can be assigned to the objects x and y with the result assigned to z. It is possible to list the objects so created using *ls()*. R will return those objects, along with others so created. To clear the workspace use the command *rm(list=list())*. Enter *ls()* to verify character(0) is displayed, which indicates the workspace has been cleared.

```
> 5 + 6
[1] 11
> x <- 5
> y <- 6
> z <- x + y
> z
[1] 11
> ls()
[1] "x" "y" "z"
> rm(list = ls())
> ls()
character(0)
```

One of the advantages of **R** is how the language operates on vectors. The output of a program using vectors is shown below. A vector "z" is generated in two different ways with its length checked and the arithmetic sum of its elements determined. Note the use of the function *c()* to ***combine*** individual elements. The length, or number of elements in z, can be determined using the *length()* function. Since the elements of z are numeric, their arithmetic total can be obtained using *sum()*. Note that the statement *print(z)* can also be used.

```
> z <- c(1, 2, 3, 1.4, 1.7, 3.1)
> z
[1] 1.0 2.0 3.0 1.4 1.7 3.1
> x <- c(1, 2, 3)
> y <- c(1.4, 1.7, 3.1)
> z <- c(x, y)
> z
[1] 1.0 2.0 3.0 1.4 1.7 3.1
> length(z)
[1] 6
> sum(z)
[1] 12.2
> print(z)
[1] 1.0 2.0 3.0 1.4 1.7 3.1
```

Sequences can be generated in different ways. Here are examples one of which shows how the *seq()* command might be used. More general sequences can be generated using the *seq()* command.

```
> x <- 1:11
> x
[1] 1 2 3 4 5 6 7 8 9 10 11
> seq(1, 11, by = 2)
[1] 1 3 5 7 9 11
> seq(1, 11, length = 6)
[1] 1 3 5 7 9 11
```

Another useful function for building vectors is the *rep()* command for repeating things. The following output illustrates some of the possibilities.

```
> rep(0,11)
 [1] 0 0 0 0 0 0 0 0 0 0 0
> rep(1:3,6)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
> rep(1:3,each = 6)
 [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
> rep(1:3,rep(6,3))
```

```
 [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
> rep(1:3, c(6, 6, 6))
 [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
```

**R** performs element-wise arithmetic. Care should be taken to make sure that **R** is doing what you wish. Note, *c()* is a generic function which combines its arguments, it does not perform arithmetic addition. The examples below show **R** uses element-wise arithmetic on vectors.

```
> x <- c(1, 2, 3)
> y <- c(1.4, 1.7, 3.1)
> z <- c(x, y)
> z
[1] 1.0 2.0 3.0 1.4 1.7 3.1
> x + y
[1] 2.4 3.7 6.1
> x <- c(1, 2, 3)
> x + 2
[1] 3 4 5
```

The *append()* command can be used to merge vectors. Suppose we wish to combine "x", "y" and "z" so that the resulting vector has the elements in ascending order. It is necessary to specify the positions. We wish to put "y" after the first element in "x" and assign the resulting new vector to "x". Then, "z" is positioned after the last element in "x", and we assign the resulting vector to "x". This shows how an object, in this case "x", can be redefined.

```
> x <- c(1, 2, 3)
> y <- c(1.4, 1.7)
> z <- 3.1
> x <- append(x, y, after = 1)
> x
[1] 1.0 1.4 1.7 2.0 3.0
> x <- append(x, z, after = 5)
> x
[1] 1.0 1.4 1.7 2.0 3.0 3.1
```

Non-numeric elements may also be merged using the *append()* command. Put "A" and "B" between "1.7" and "2.0". The position of 3 in the sequence must be designated (i.e. after = 3). Note that when we append the character elements, the elements of "x" are coerced to characters; they are no longer numeric.

```
> append(x, c("A", "B"), after = 3 )

[1] "1" "1.4" "1.7" "A" "B" "2" "3"  "3.1"
```

Functions will be considered later in this guide. Keep in mind, functions will operate element-wise on an object. For example, *sqrt()* calculates the square root of the vector z.

```
> z <- c(16, 4)
> sqrt(z)
[1] 4 2
```

The preceding has introduced various functions. If you don't know how to use a function, or don't know what the arguments (i.e. options) or their default values are, type *help(functionname)* where "functionname" is the name of the function you are interested in. You can also type *?functionname()* into the console and obtain information. This information may not be sufficient, at which point further research will be necessary.

## Exercises

1. Define the vectors, "x" and "y". Determine the answers to (a) through (g) and check with **R.**

```
x <- seq(1, 6)
y <- rep(1:3, 2)
```

(a) print(x) and print(y)
(b) combine the elements of y with x
(c) find the length of c(x, y)
(d) sum the elements in c(x,y)
(e) calculate x + y, x*y, x -1, x^2

2. Decide what the following expressions are and use **R** to check your answers.

(a) seq(2, 9)
(b) seq(2, 9, length = 8)
(c) seq(9, 2, -1)
(d) rep(c(1, 2), 4)
(e) rep(c(1, 2), c(4, 4))
(f) rep(1:4, rep(3, 4))

3. Use the rep function to define the following vectors in **R**.

(a) 6, 6, 6, 6, 6, 6
(b) 5, 8, 5, 8, 5, 8, 5, 8
(c) 5, 5, 5, 5, 8, 8, 8, 8

## 6 Descriptive Statistics

This section will demonstrate how to calculate various statistics from data. The object "x" shown below will be used. The *which.max()* and *which.min()* functions identify the location of the minimum and maximum values. The *quantile()* function is useful for identifying the associated percentile location, in this case the 25th percentile, 50th percentile, and the 75th percentile, which are known as the first, second, and third quartiles. The median is the second quartile.

```
> x <- seq(11,1)
> x
 [1] 11 10 9 8 7 6 5 4 3 2  1
> min(x)
[1] 1
> max(x)
[1] 11
> which.min(x)
[1] 11
> which.max(x)
[1] 1
> quantile(x, probs = c(0.25, 0.5, 0.75))
25% 50% 75%
3.5   6.0   8.5
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     3.5     6.0     6.0     8.5    11.0
```

Other useful functions include the *round()* function, which will round to a specified number of decimal places, whereas *signif()* will round to a specified number of digits. The *ceiling()* function finds the smallest integer greater than a value, and *floor()*finds the largest integer less than a value.

```
> x <- c(8.75, 9.45, 4.35, 6.85, 9.45, 10.55, 7.75, 8.25, 10.55, 2.45, 15.75, 7.45 )
> mean(x)
[1] 8.466667
> round(mean(x), digits = 3)
[1] 8.467
> signif(mean(x), digits = 3)
[1] 8.47
> ceiling(mean(x))
[1] 9
> floor(mean(x))
[1] 8
```

Using the data shown above, it may be that we subsequently learn that three people contributed the data in consecutive groups of four values. It is possible to extract from x the three sets of results with subscripting and examine summary statistics for each.

```
> x[1:4]
[1] 8.75 9.45 4.35 6.85
> summary(x[1:4])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.350 6.225 7.800 7.350 8.925 9.450
> x[5:8]
[1] 9.45 10.55 7.75  8.25
> summary(x[5:8])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  7.750 8.125 8.850 9.000 9.725 10.550
> x[9:12]
[1] 10.55 2.45 15.75  7.45
> summary(x[9:12])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.45    6.20    9.00    9.05 11.85 15.75
```

## Exercises

**1.** x <- c(8.75, 9.45, 4.35, 6.85, 9.45, 10.55, 7.75, 8.25, 10.55, 2.45, 15.75, 7.45 )
Determine the following and check with **R.**

(a) x[6] + x[7]
(b) x[c(5, 6, 7, 8)]
(c) x[5:8]
(d) x[c(1:4, 9:12)]
(e) The combination of the results from (c) and (d).

2. x <- c(8.75, 9.45, 9.35, 9.85, 9.45, 10.55, 9.75, 8.25, 10.55, 9.45, 9.75, 8.45 )

The vector "x" contains birth weights in ounces of puppies from two different litters. Each litter had six puppies. The first six values are from the first litter, and the last six from the second litter. Produce a statistical summary of the birth weights for each litter.

# 7 Logical Comparisons

Logical values, sometimes called ***logicals***, are a non-numeric data type often created via comparison between variables. Relational operators are typically used on numeric values to determine if a comparison is TRUE or FALSE. The table below lists these relational operators.

| Operator | Interpretation |
|---|---|
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| !x | Not x |
| x \| y | x or y |
| x & y | x and y |

Logicals allow you to test the equivalence of different R objects in typical mathematical fashion to produce TRUE or FALSE results. Such comparisons are often used as conditions for performing further operations using ”if-then” or “if-then-else” statements. While some programming languages require you to type “then” in order to actually execute the second part of an if-statement, R does not require this. Suppose you want to determine if a variable is less than or equal to zero and take note of this condition.

```
> x <- -1
> if (x > 0) signal <- "FAIL" else
+ if (x <= 0) signal <- "SUCCESS"
> signal
[1] "SUCCESS"
```

Logicals can also be combined with the operators: & (AND), and | (OR), to produce logicals that test more than one criterion. Note that the exclamation symbol ! represents NOT. In the following example, we introduce the & symbol with an additional condition.

```
> x <- -1
> if (x > 0) signal <- "FAIL" else
+ if (x <= 0 & x > -2) signal <- "SUCCESS" else
+     signal <- "FAIL"
> signal
[1] "SUCCESS"
```

Logicals may be used to determine the location of an element in a vector. To locate the value 8.2 in the object x below, and count the number of occurrences, **R** treats TRUE as the value 1 and FALSE as the value 0. The *sum()* function can be used to count occurrences. Substituting the logical vector "location" into the vector x returns 8.2, twice. The *which()* function returns the locations(s).

```
> x <- c(7.5, 8.2, 3.1, 5.6, 8.2, 9.3, 6.5, 7.0, 9.3, 1.2, 14.5)
> location <- x == 8.2
> location
 [1] FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
> sum(location)
[1] 2
> x[location]
[1] 8.2 8.2
> which(x == 8.2)
[1] 2 5
```

Similar operations are possible with non-numeric elements:

```
> x <- rep(c("A", "B", "C"), 3)
> x
[1] "A" "B" "C" "A" "B" "C" "A" "B" "C"
> location <- x == "B"
> location
[1] FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE
> which(x == "B")
[1] 2 5 8
```

**R** is capable of handling missing values. Missing data in **R** appears as NA. NA is not a string or a numeric value, but an indicator of a missing value. We can create vectors with missing values.

```
> x1 <- c(1, 4, 3, NA, 7)
> x2 <- c("a", "B", NA, "NA")
```

NA is a non-number that does not generate an error. In x2, the third value is missing, while the fourth value is the character string "NA". Use the **is.na** function to see which is missing.

```
> is.na(x1)
[1] FALSE FALSE FALSE TRUE FALSE
> is.na(x2)
[1] FALSE FALSE TRUE FALSE
```

**R** distinguishes between the NA and "NA" in x2, NA is seen as a missing value, "NA" is not.

### Exercises

1. For `y <- c(33, 44, 29, 16, 25, 45, 33, 19, 54, 22, 21, 49, 11, 24, 56)`, find the minimum and maximum of y and their location in the vector.

2. Find the median of y and use logicals to split y into two subsets. One subset will have all values in y strictly less than the median, and the other subset all values strictly greater than the median. Print the resulting subsets.

## 8 Matrices

Many of the functions in **R** utilize matrices. Multiple linear regression is an example. Creating a matrix can be accomplished in a variety of ways using the functions *cbind()*, *rbind()*, and *matrix()*. The *dim()* command can be used to check dimensions.

```
> x <- c(1, 2, 3)
> y <- c(1.4, 1.7, 4)
> z <- rbind(x, y)
> dim(z)
[1] 2 3
> z <- cbind(x, y)
> z
     x y
[1,] 1 1.4
[2,] 2 1.7
[3,] 3 4.0
> dim(z)
[1] 3 2
```

The functions *cbind()* and *rbind()* can also be passed matrices themselves (provided the dimensions match) to form larger matrices. For example,

```
> cbind(z,z)
     x y x y

[1,] 1 1.4 1 1.4
[2,] 2 1.7 2 1.7
[3,] 3 4.0 3 4.0
> rbind(z,z)
     x y
[1,] 1 1.4
[2,] 2 1.7
[3,] 3 4.0
```

```
[4,] 1 1.4
[5,] 2 1.7
[6,] 3 4.0
```

The function *matrix()* can be used to construct a matrix. Notice that the dimensions of the matrix are determined by the size of the vector and the requirement that the number of rows is 3, as specified by the argument nrow = 3. As an alternative, we could have specified the number of columns with the argument ncol = 4 (obviously, it is unnecessary to give both). Notice that the matrix is 'filled up' column-wise. It is also possible to fill up row-wise using byrow = T.

```
> x <- c(8.75, 9.45, 4.35, 6.85, 9.45, 10.55, 7.75, 8.25, 10.55, 2.45, 15.75, 7.45 )
> matrix(x, nrow = 3)
     [,1] [,2] [,3]  [,4]
[1,] 8.75 6.85 7.75  2.45
[2,] 9.45 9.45 8.25 15.75
[3,] 4.35 10.55 10.55  7.45
> matrix(x, ncol = 4)
     [,1] [,2] [,3]  [,4]
[1,] 8.75 6.85 7.75  2.45
[2,] 9.45 9.45 8.25 15.75
[3,] 4.35 10.55 10.55  7.45
> matrix(x, nr = 3, byrow = T)
      [,1] [,2] [,3] [,4]
[1,] 8.75 9.45 4.35 6.85
[2,] 9.45 10.55 7.75 8.25
[3,] 10.55 2.45 15.75 7.45
```

It is possible to add, subtract and multiply matrices. These operations are performed element-wise. **R** will try to interpret operations on matrices in a natural way.

```
> x <- c(8.75, 9.45, 4.35, 6.85)
> x <- matrix(x, nrow = 2)
> x

     [,1] [,2]
[1,] 8.75 4.35
[2,] 9.45 6.85
> y <- matrix(seq(-1,2), nrow = 2)
> y
     [,1] [,2]
[1,]   -1    1
[2,]    0    2
```

```
> x + y
     [,1] [,2]
[1,] 7.75 5.35
[2,] 9.45 8.85
> x - y
     [,1] [,2]
[1,] 9.75 3.35
[2,] 9.45 4.85
> x*y
      [,1]  [,2]
[1,] -8.75  4.35
[2,] 0.00 13.70
```

To perform matrix multiplication %*% is used. Note the difference.

```
> x%*%y
      [,1]  [,2]
[1,] -8.75 17.45
[2,] -9.45 23.15
```

The *round()* function can be passed matrices. Each element is rounded accordingly.

```
> round(x%*%y, digits = 1)
     [,1] [,2]
[1,] -8.8 17.4
[2,] -9.4 23.1
```

Useful functions for operating on matrices are *t()* to return the transpose and *solve()* to calculate inverses which can be useful for solving systems of linear equations. Suppose we wish to solve the following system of linear equations.

$$x - y = 1$$
$$x + y = 3$$

```
> y <- c(1, 3)
> y
[1] 1 3
> x <- matrix(c(1, -1, 1, 1), nr = 2, byrow = T)
> x
     [,1] [,2]
[1,]    1   -1
[2,]    1    1
```

Matrix multiplication of the inverse of x with y provides the solution which can be verified. The inverse is determined by *solve()*. Operations may be combined.

```
> solve(x, y)
[,1] 2, 1
> x%*%(solve(x, y)
     [,1]
[1,]    1
[2,]    3
```

As with vectors, it is useful to extract sub-components of matrices and pick out individual elements, rows or columns. The [ ] notation is used to subset. The following examples using x illustrates this.

```
> x[1, 1]
[1] 1
> x[c(1,2), 2]
[1] -1  1
> x[, 2]
[1] -1  1
```

It is necessary to specify which rows and columns are required, whilst omitting the integer for either dimension implies that every element in that dimension is selected. This matrix can be converted into an R object called a ***data frame***. Matrices and data frames have some similar properties, but also differences. Importantly, most data that is read in from an outside file (as opposed to typing in ourselves) will be in the form of a data frame. If you want to know whether something is a matrix or a data frame, you can test it using the *is.matrix()* or *is.data.frame()* command.

## Exercises

1. Using the following code, create the matrices x and y. Perform the indicated calculations and check your answers in **R**.

```
x <- c(4,  3, 2, 1)
x <- matrix(x, nrow = 2)
y <- c(4, 9, 1, 16)
 y <- matrix(x, nrow = 2)
```

(a) 2*x
(b) sqrt(y)
(c) x*x
(d) x%*%x
(e) solve(y)
(f) round(solve(y)%*%y, digits = 1)
(g) y[1,]
(h) y[,2]

2. Define z <- c(2, 1). Calculate z%*%y and y%*%z. Check your answers in **R**.

## 9 Accessing Example Datasets

**R** maintains a library of datasets. Type *data*() to obtain a description of what is available. To access a dataset**,** type *data(dataset)* where "*dataset"* is the name of the dataset you are interested in. For example, the following code reveals the dataset "cars" to have two variables and fifty observations. The structure of the dataset is returned using *str*(), and *head*() is used to return the first five lines of the dataset. The *summary*() function provides basic descriptive statistics.

```
> data()
> data(cars)
> str(cars)
'data.frame':          50 obs. of 2 variables:
 $ speed: num 4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num 2 10 4 22 16 10 18 26 34 17 ...
> head(cars, n = 5L)
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
> summary(cars)
     speed       dist
 Min.   : 4.0   Min.   :  2.00
 1st Qu.:12.0   1st Qu.: 26.00
 Median :15.0   Median : 36.00
 Mean   :15.4   Mean   :  2.98
 3rdQu.:19.0    3rdQu.: 56.00
 Max.   :25.0   Max.   :120.00
```

There are various ways to access the data in the dataset “cars” using indexing, and also by specifying particular variables. Note the use of the “$” operator in the code below, which identifies the variable speed with the dataset “cars” generating the variable “cars$speed”.

```
> median(cars[,2])
[1] 36
> mean(cars[,1])
[1] 15.4
> mean(cars$speed)
[1] 15.4
> speed <- cars$speed
> mean(speed)
[1] 15.4
```

## Exercises

1. Use the dataset women and find the mean height and mean weight for individuals in the dataset (type *help(women)* for a description of the variables available).

2. Use the summary() function to generate descriptive statistics for the dataset women.

## 10 The *apply()* Function

The *apply()* function provides an efficient way to use the same function on every row or column of a matrix or array. The code below shows how to calculate the average height and weight of individuals in the dataset “women”. Note the dataset has the measurements in column format with the variables constituting columns. This is why the argument “2” appears in apply(). If calculations for each row were of interest, the argument “1” would appear in apply().

```
> data(women)
> str(women)

'data.frame':	15 obs. of 2 variables:
 $ height: num 58 59 60 61 62 63 64 65 66 67 ...
 $ weight: num 115 117 120 123 126 129 132 135 139 142 ...
> women
   height weight
1      58    115
2      59    117
3      60    120
```

```
4     61    123
5     62    126
6     63    129
7     64    132
8     65    135
9     66    139
10    67    142
11    68    146
12    69    150
13    70    154
14    71    159
15    72    164
> apply(women, 2, mean)
  height  weight
 65.0000 136.7333
```

The *apply()* function is one of a set of functions in base R referred to as the "apply family". More specifically, the family is made up of *apply(), lapply(), sapply(), vapply(), mapply, rapply(),* and *tapply()*. These functions provide ways to manipulate slices of data from matrices, arrays, lists, and data frames in a repetitive way.

## Exercises

1. Repeat the analyses shown above with *apply()* using the height and age in the Loblolly dataset. Repeat using the speed and distance data in the cars dataset. Compare your results to what is obtained using the *summary()* function.

## 11 The *aggregate()* and *table()* Functions

Different formats can take advantage of grouping variables. The dataset "ToothGrowth" has three variables: len (tooth length), supp (Supplement type (VC or OJ**),** and dose (numeric dose in milligrams/day). Suppose we want the mean values of len for each combination of dose and supp. Since dose enters as a numeric at three levels, these levels can be used as labels. For purposes of illustration, it will be converted to a factor variable with levels "0.5", "1.0" and "2.0" using the *factor()* function. The quotation marks denote a character (i.e. non-numeric) variable.

```
> data(ToothGrowth)
> ToothGrowth$dose <- factor(ToothGrowth$dose)
> str(ToothGrowth)
'data.frame':        60 obs. of 3 variables:
 $ len : num 4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ dose: Factor w/ 3 levels "0.5","1","2": 1 1 1 1 1 1 1 1 1 1 ...
```

```
> aggregate(ToothGrowth$len, by = list(ToothGrowth$supp, ToothGrowth$dose), mean)

  Group.1 Group.2     x
1      OJ     0.5 13.23
2      VC     0.5  7.98
3      OJ       1 22.70
4      VC       1 16.77
5      OJ       2 26.06
6      VC       2 26.14
```

The coding can be simplified using the *with()* function. The following code will produce the table shown above: *with(ToothGrowth, aggregate(len, by = list(supp, dose), mean))*. It is possible to rename the columns for clarity using the *colnames()* function as shown below.

```
> result <- with(ToothGrowth, aggregate(len, by = list(supp, dose), mean))
> colnames(result) <- c("supp", "dose", "mean")
> result

  supp dose  mean
1   OJ  0.5 13.23
2   VC  0.5 7.98
3   OJ    1 22.70
4   VC    1 16.77
5   OJ    2 26.06
6   VC    2 26.14
```

Sample sizes may be desired for each combination of supp and dose. The *table()* function is applicable here. The *addmargins()* and *with()* functions may also be used as desired.

```
> addmargins(table(ToothGrowth$supp, ToothGrowth$dose))

      0.5  1 2  Sum
  OJ   10 10 10  30
  VC   10 10 10  30
  Sum  20 20 20  60
```

## Exercises

1. Use *aggregate()* to determine the median len for each combination of supp and dose.
2. Use *with()* and *addmargins()* to produce the table of counts for the ToothGrowth data.

## 12 Loops

An important part of **R** programming is looping. Like *apply()* and *aggregate()*, loops allow you to perform repetitive tasks on data. There are several different types of loops that can be used. In this section "for" loops and "while" loops will be discussed The "for" loop tells **R** that to conduct a task a specified number of times. The format is simple:

for (k in *values*){
*…program statements…*
}

Here *values* is a vector that gives the identifier "k" particular values to use for looping. The code between the parentheses must be executed for each value "k" assumes. There is nothing special about using "k". Other symbols may be used. Suppose we want to sum the integers from 1 to 10 and print the sum. Each subsequent value of k is added to the sum until k exceeds 10.

```
> sum <- 0
> for (k in 1:10){
+ sum <- sum + k
+ }
> sum
[1] 55
```

A "while" loop is different. Computing continues as long as the loop condition is satisfied. In the example, when k exceeds 10, the loop condition is not satisfied and the iteration ceases. If the coding is wrong, the iterations may not cease if the loop condition is not satisfied. Be careful.

```
> sum <- 0
> k <- 1
> while(k <= 10){
+ sum <- sum + k
+ k <- k + 1
+ }
> sum
[1] 55
```

### Exercises

1. Write a "for" loop to compute the value of a factorial for integers greater than zero. Execute the loop for an integer equal to 5, and print the factorials for 1, 2, 3, 4 and 5. Repeat the above, but with a "while" loop.

## 13 Writing functions

**R** has the facility to extend the language with user-supplied functions. Often, the format for a user-defined function is:

```
myfunction <- function(arg1, arg2, ... ){
        statements
          return(object)
}
```

With library-defined functions, an argument (i.e. arg1, arg2,…) provides us with a method for passing values that dictate or inform on the execution or output of the function.

Objects in the function are local to the function. The object returned can be any data type. A function must be written. Once written it can be called at any time within a program. User-defined functions can be used the same as library functions. For example, there is no function for calculating the percent coefficient of variation. We can define such a function. In this example, we assume the mean value in the calculation will never be zero.

```
> cv <- function(x){
+ object <- 100 * sd(x) / mean(x)
+  return(object)
+ }
```

Suppose we want the percent coefficient of variation for Girth, Height, and Volume from the tree data. The code below shows different ways to do this. This can be done efficiently using the *apply()* function with the *cv()* function just defined. Note that *apply()* uses the function *cv()* on each column in trees. Keep in mind, *cv()*must be defined in the program prior to calling it.

```
> data(trees)
> girth <- trees[, 1]
> cv(girth)
[1] 23.68695
> cv(trees[,1])
[1] 23.68695

> cv(trees$Girth)
[1] 23.68695
> apply(trees, 2, cv)
   Girth       Height    Volume
23.686948 8.383964 54.482331
```

## Exercise

1. Write a simple function that computes the sample variance for a vector of numerical values. Use with *apply()* to compute the sample variance for the dimensions in the tree data.

# 14 Statistical Computation, Simulation and Random Sampling

**R** provides a library of probability distributions including the normal, binomial, Poisson, t, F, Chi-squared, uniform, and hypergeometric distributions. There are functions in **R** to evaluate the density function, the distribution function and the quantile function (the inverse distribution function). There are also functions to generate random samples from these distributions. Using the normal distribution as an example, these functions are, respectively, *dnorm()*, *pnorm()*, *qnorm()* and *rnorm()*. Using *help()* in **R** can serve as a reminder of the necessary arguments for their use.

The normal distribution is frequently used. Unlike with tables, there is no need to standardize the variables first. Consider a normal variable $X$ with mean 3 and standard deviation 2. The density function *dnorm(x, mean = 3, sd = 2)* will evaluate the density at points contained in the vector x (note, *dnorm()* will assume mean = 0 and standard deviation = 1 unless these are specified). Note that the function requires specification of the standard deviation rather than the variance. As an example, the following shows the density of the $N(3,9)$ distribution at $x = 5$.

```
> dnorm(5, mean = 3, sd = 3)
[1] 0.1064827
```

The following calculates the normal density function values at intervals of 0.1 over the range [−5, 10]. The functions pnorm and qnorm work in an identical way.

```
> x <- seq(-5, 10, by = .1)
> results <- dnorm(x, mean = 3, sd = 2)
> head(results, n = 5L)
[1] 6.691511e-05 8.162820e-05 9.932774e-05 1.205633e-04 1.459735e-04
```

Similar functions exist for other distributions. For example, *dt()*, *pt()*, *qt()*, and r*t()* for the *t*-distribution, though in this case it is necessary to give the degrees of freedom rather than the mean and standard deviation. Other distributions available include the binomial, exponential, Poisson, and hypergeometric. Care is needed interpreting the functions for discrete variables. Here some background reading and experimentation are needed. Use *help()* for further information.
One important technique for many statistical applications is the simulation of data from specified probability distributions. **R** enables simulation from a wide range of distributions. To simulate 100 random observations from the $N(3, 4)$ distribution, we write *rnorm(100,3,2)*. Similarly, for example, *rt()*, *rpois()* may be used for simulation from the *t* and Poisson distributions.

Closely related to this is sampling from a data file. Random samples and sequential samples can be drawn. A vector needs to be generated that identifies the rows being selected. Using trees as an example, if we wanted to pick 10 observations at random, the following statements using the *sample()* function would suffice. In this instance, *sample()* will randomly select 10 integers between 1 and *nrow(trees)* without replacement. These integers can then be used to select the corresponding rows from trees as shown below.

Please note random sampling requires a random number generator. The *set.seed()* statement with the argument 123 tells this generator where to begin. Different arguments may be used.

```
> data(trees)
> set.seed(123)
> index <- sample(1:nrow(trees), 5, replace = FALSE)
> trees[index,]
   Girth Height Volume
9   11.1     80   22.6
24  16.0     72   38.3
12  11.4     76   21.0
25  16.3     77   42.6
26  17.3     81   55.4
```

For a systematic sample, if we wanted every tenth observation in trees. The index would be determined by the statement: index <- seq(1, nrow(trees), by = 10).

## Exercises

1. Suppose $X$ has a normal distribution with mean 2 and variance 0.25. Denote by $f$ and $F$ the density and distribution functions.
   (a) Calculate the density function value at 0.5, $f(0.5)$ (use *dnorm()*)
   (b) Calculate the distribution function value at 2.0, $F(2.0)$ (use *pnorm()*)
   (c) Calculate the $95^{th}$ percentile (use *qnorm()*)
   (d) Calculate the probability that $X$ is between -1 and , $Pr(1 <= X <= 3)$
2. Repeat question 1 in the case that $X$ has a $t$-distribution with 5 degrees of freedom.

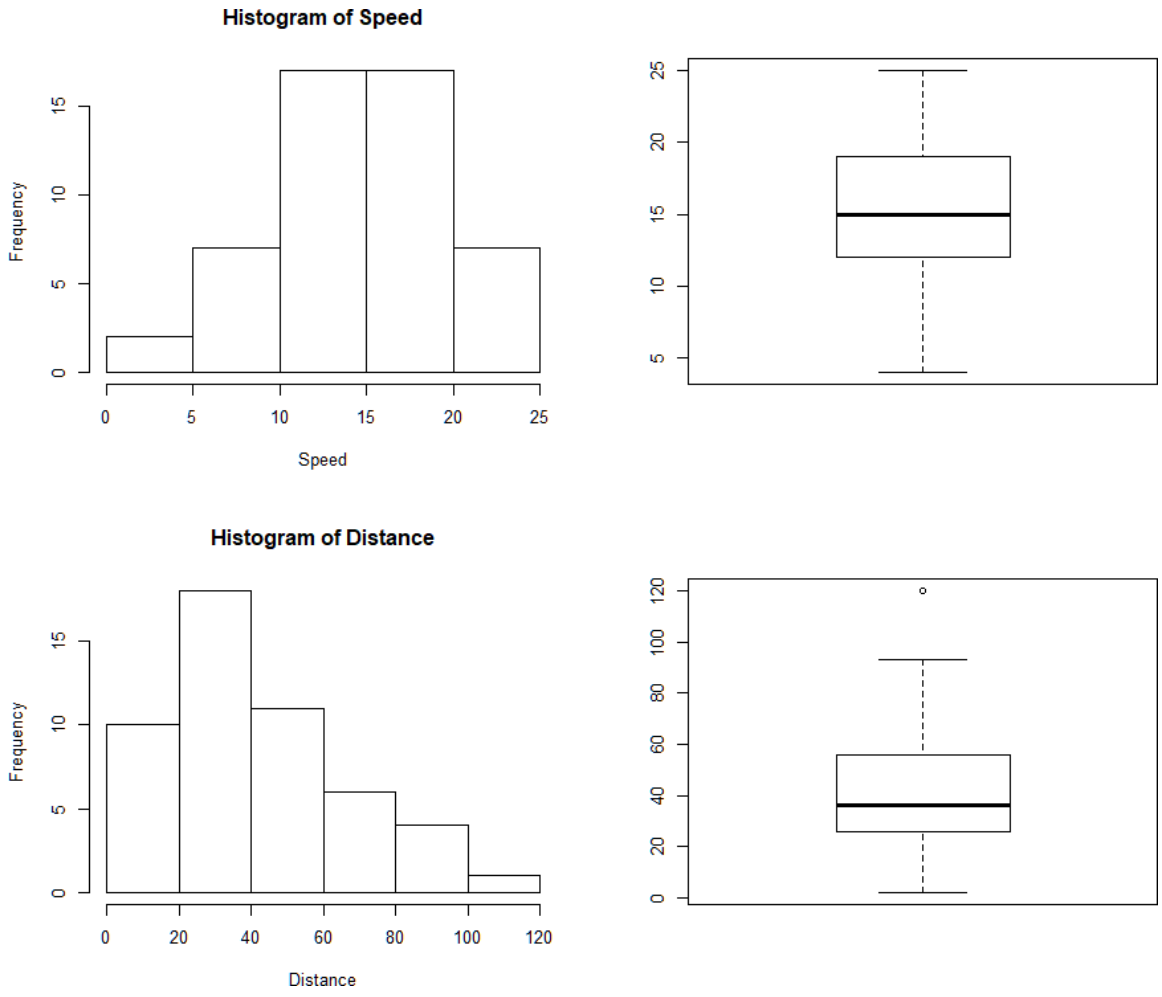
# 15 Graphics

One of the advantages of **R** is its capability to produce high quality graphics. Base **R** graphics can serve many purposes, and there are packages such as ggplot2 which can also be used. What follows is a brief overview of selected base **R** data displays useful for statistical work.

When evaluating data it can be valuable to present more than one display or visualization of the data. This is possible in base **R** using *par()* to set graphical parameters. There are many possibilities. For example, *par(mfrow=c(2,2))* creates a window of graphics with 2 rows and 2 columns. With this choice the windows are filled row-wise. Use mfcol instead of mfrow to fill column-wise. It is good practice to conclude a block of code such as this with the statement *par(mfrow = c(1,1))* to reset the function *par()*. The following code illustrates how the speed and distance of cars in the dataset "cars" can be displayed using histograms and boxplots.

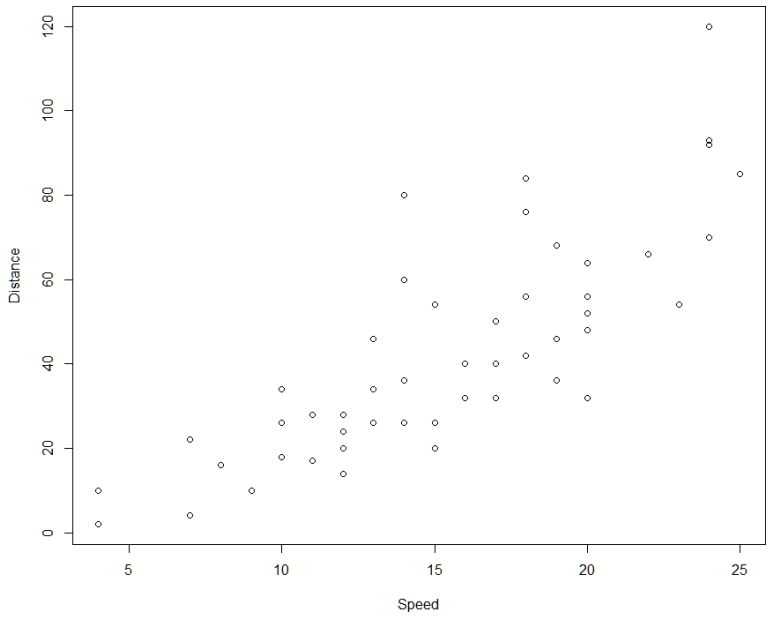
```
> data(cars)
> Speed <- cars$speed
> Distance <- cars$dist
> par(mfrow = c(2,2))
> hist(Speed)
> boxplot(Speed)
> hist(Distance)
> boxplot(Distance)
> par(mfrow = c(1,1))
```

The statements above produce the plot below. Compare the axes of the histograms and boxplots with the values produced by *summary()*.



We can also generate a scatterplot of one variable versus another using the function *plot()*.

```
> plot(Speed, Distance)
```




The *plot()* function is object-specific: its behavior depends on the object being used. There are also many optional arguments in most plotting functions that can be used to control colors, plotting characters, axis labels, titles, etc. Using these requires investigation and sometimes trial and error.

## Exercises

1. Use the dataset "faithful" and create a scatter plot of the variables.

2. Use *x <- rnorm(250)* to generate 250 standard normal random variables. Produce the histogram and compare to a stem-and-leaf plot of the data. The function *stem()* will produce the stem-and-leaf plot.

# 16 Color Applications

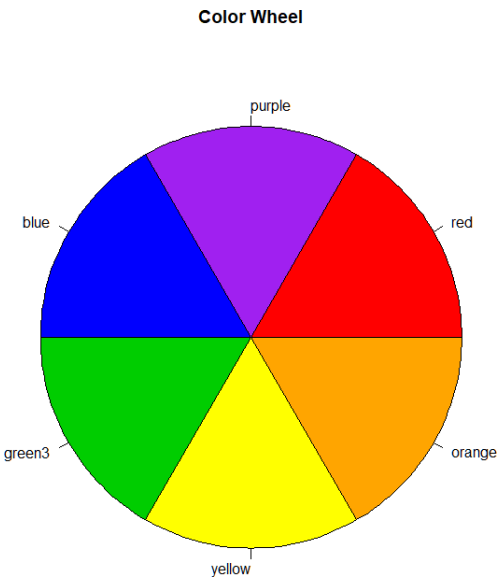
Color is a valuable feature to use for data visualization. There is a wide range of packages in R that can be used for this purpose. This handout will demonstrate applications of rgb(), and other functions from the base, "grDevice" package, and the RColorBrewer() function from the package of the same name. There are three types of color designations in R: 1) hexadecimal colors (#rrggbb), 2) named colors, and 3) integers referring to positions in the current color palette. The hexadecimal designation is the most generally machine- readable approach, and is how colors are actually stored. It is frequently more convenient to work with named colors (i.e. "red", "blue",…) or positive integers, rather than the hexadecimal representation. It is possible to convert from one designation to another, although there are more hexadecimal identifications then there are names.

A list of colors by name can be produced using the *colors()* function. There are 657 names. Individual names can be accessed by number.

```
> color <- colors()
> str(color)
 chr [1:657] "white" "aliceblue" "antiquewhite" "antiquewhite1" "antiquewhite2"
"antiquewhite3" "antiquewhite4" "aquamarine" ...
> color[1:3]
[1] "white"       "aliceblue"   "antiquewhite"
```

R provides a default palette which may be used “as is” or modified. The code below shows this default palette; and, also how to redefine it ultimately presenting a color wheel of primary colors.

```
> color <- palette("default")
> color
[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"  "gray"
> palette(c("red", "purple", "blue", "green3", "yellow", "orange"))
> color <- palette(c("red", "purple", "blue", "green3", "yellow", "orange"))
> color
[1] "red"    "purple" "blue" "green3" "yellow" "orange"
> pie(rep(1, 6), labels = c(color[1:6]), col = c(color[1:6]), main = "Color Wheel")
```




## Hexidecimal Codes

Should it be of interest to determine the hex color code, the package “gplots” provides a function, *col2hex()*, that does this. The illustration below shows three ways to determine these codes using the palette defined above:

```
> library(gplots)
> col2hex(c("red", "purple", "blue", "green3", "yellow", "orange"))
[1] "#FF0000" "#A020F0" "#0000FF" "#00CD00" "#FFFF00" "#FFA500"
> col2hex(color[1:6])
[1] "#FF0000" "#A020F0" "#0000FF" "#00CD00" "#FFFF00" "#FFA500"
> col2hex(seq(1, 6))
[1] "#FF0000" "#A020F0" "#0000FF" "#00CD00" "#FFFF00" "#FFA500"
```

Hex color codes may be used in place of names, palette numbers, and integer designations. Since there are more hex codes than names, it is sometimes necessary to find a named color that is the closest to the color indicated by the hex designation. Here are some links to sites that provide programs for decoding hex colors:

- Yellowpipe Internet Services Color Converter
- RapidTables Color Conversion

These links also utilize the *rgb()* function.

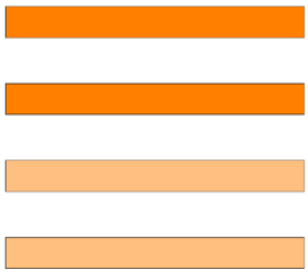
## RGB Representation

The *rgb()* function provides a means to blend red, green, and blue to form a desired color. The function has the form: *rgb(red, green, blue, alpha, names = NULL, maxColorValue)*.

- red – A number (or vector), where higher numbers mean more red.
- green – A number (or vector) , where higher numbers mean more green.
- blue – A number (or vector of numbers), where higher numbers mean more blue.
- alpha is an optional argument for transparency, and has the same intensity scale as the red, green, and blue values. It usually is a number (or vector of numbers). between 0 and 1, where 0 is fully transparent and 1 is opaque.
- names is an optional argument that will print a name with the hexadecimal value.
- maxColorValue must be specified if the maximum intensity is not 1 (for example, if intensity has a scale of 0–255). The minimum intensity must be zero.

The values for *rgb()* can be expressed as either proportions between 0 and 1 or as integers between 0 and 255 as shown below. An alpha level was introduced corresponding to the scale.

```
> par(mfrow = c(4, 1))
> new_orange = rgb(255, 127, 0, maxColorValue = 255)
> barplot(1, axes = FALSE, col = new_orange)
> new_orange = rgb(1.0, 0.5, 0, maxColorValue = 1.0)
> barplot(1, axes = FALSE, col = new_orange)
> new_orange = rgb(255, 127, 0, 127, maxColorValue = 255)
> barplot(1, axes=FALSE, col=new_orange)
> new_orange = rgb(1.0, 0.5, 0, 0.5, maxColorValue = 1.0)
> barplot(1, axes = FALSE, col = new_orange)
> par(mfrow = c(1, 1))
```

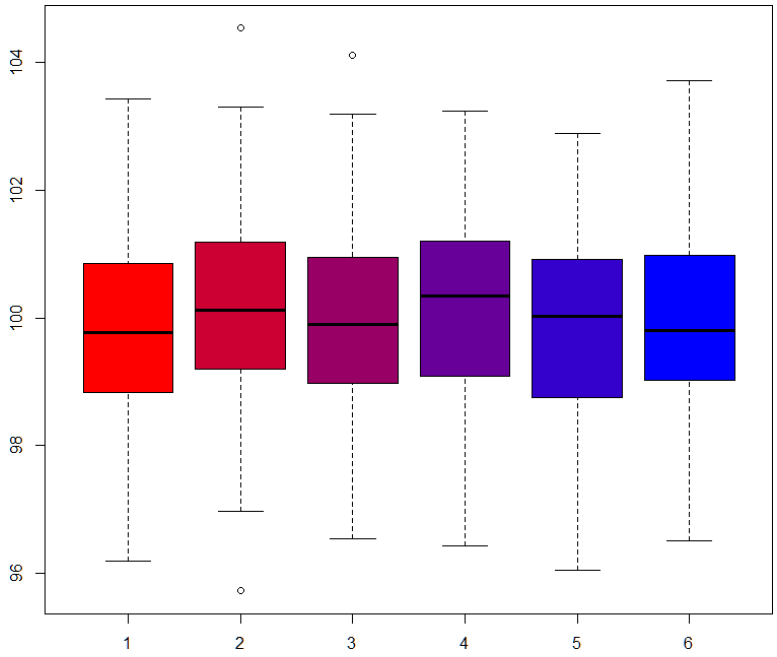


The *rgb()* function converts red, green, and blue intensities into a hexidecimal representation. This means the *rgb()* function, with specified arguments, may be used in place of hex codes, names, or integer designations.

The *col2rgb()* function converts R colors (a hex color, named color, or integer representing a palette position) to the rgb representations. The function takes either a single color or a vector of colors, and returns a matrix of three rows (red, green, blue), with one column for each color. The function has the form *col2rgb(color, alpha=FALSE)* where alpha is an optional argument to indicate whether alpha transparency values should be returned. The example below shows how to use *rgb()* to blend colors in a display, and also how to find the proportions used for each.

```
> n <- 6
> rand.data <- replicate(n, rnorm(100, 100, sd = 1.5))
> sq <- seq(0, n-1, 1)
> col.list <- rgb(1-sq/(n-1), 0, sq/(n - 1))
> boxplot(rand.data, col = col.list)
> col2rgb(rgb(1-sq/(n-1), 0, sq/(n - 1)))

      [,1] [,2] [,3] [,4] [,5] [,6]
red    255  204  153  102   51    0
green    0    0    0    0    0    0
blue     0   51  102  153  204  255
```




## RColorBrewer

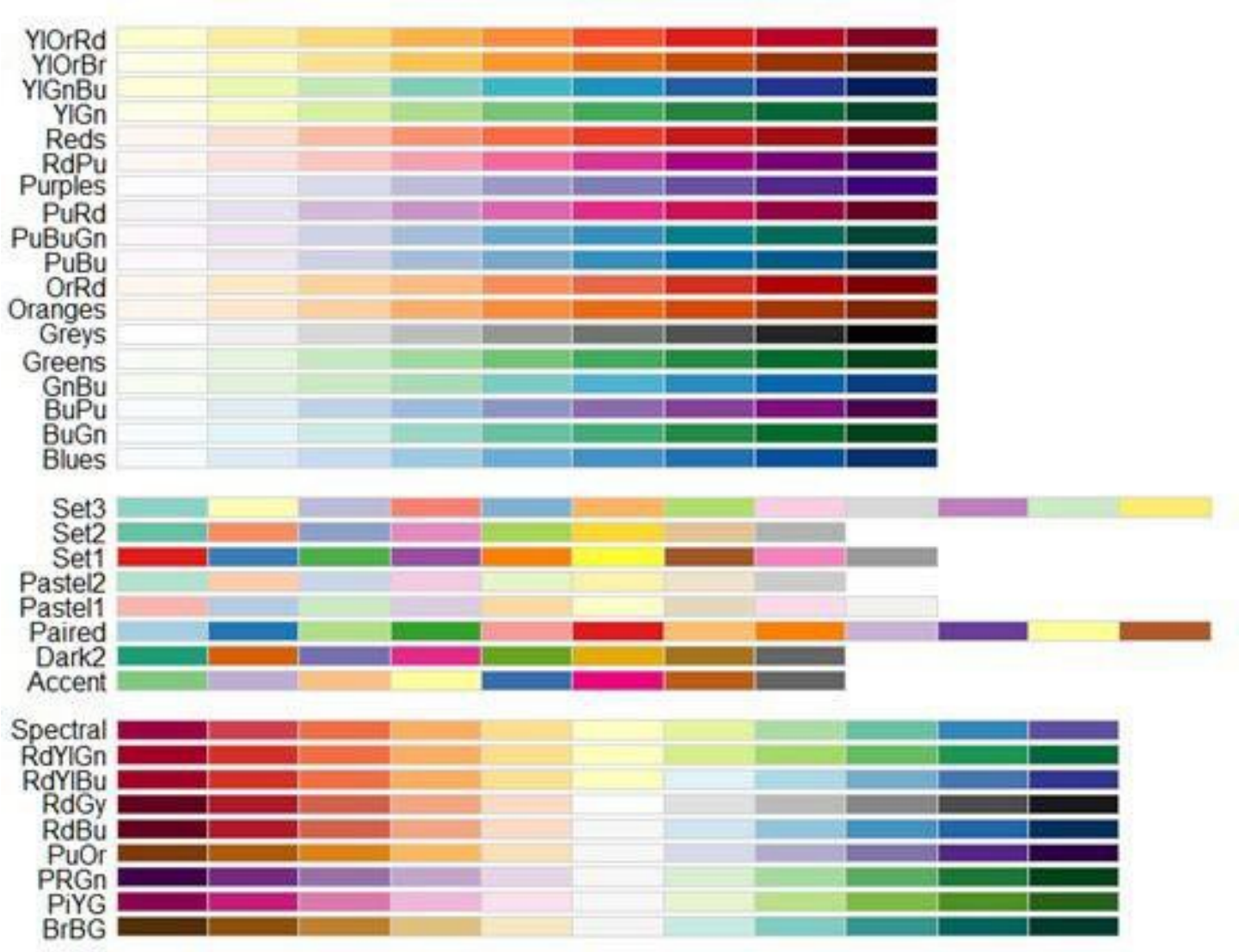
RColorBrewer is an R package that uses the work from ColorBrewer2 for development of color schemes for data visualization. The colors are split into three groups: sequential, diverging, and qualitative.

1. Sequential – Light colors for low data, dark for high data useful for interval data.
2. Diverging – Light colors for mid-range data, low and high contrasting dark colors.
3. Qualitative – Colors designed to give maximum visual difference between classes.

The following code presents the various schemes that are available. Using these schemes is easy. Each scheme has a name, and the colors are identified numerically by position.

```
if (!require("RColorBrewer")) {
  install.packages("RColorBrewer")
  library(RColorBrewer)
  }
display.brewer.all()
```

The main function is *brewer.pal()*, to which you give the number of colors you want, and the name of the palette, which you can choose from running *display.brewer.all()*. There are limits on the number of colors you can use, but if you want to extend the Sequential or Diverging groups you can do so with the *colorRampPalette()* command. Note the following examples.

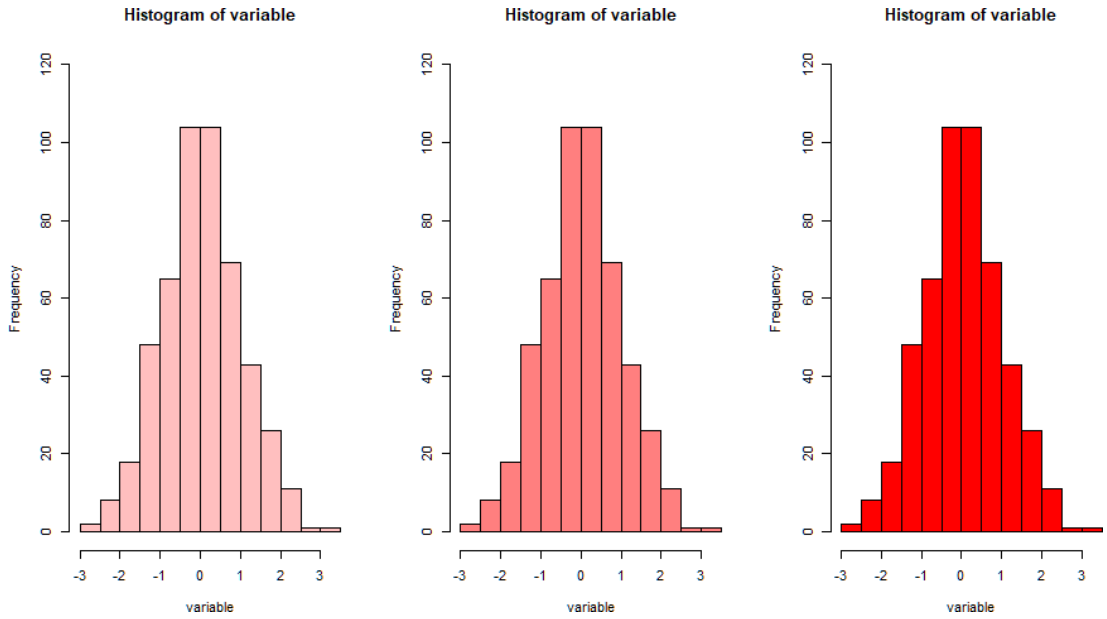



- This example shows how to use Spectral with *brewer.pal()* to generate a barplot. Using *colorRampPalette()*, it is also possible to blend (interpolate) colors from RColorBrewer to expand upon a scheme.

```
> require("RColorBrewer")
> set.seed(123)
> sample <- table(rbinom(10000, 10, 0.5))/10000
> successes <- (seq(0, 10))
> barplot(sample, main = "Binomial Sample Proportions", ylab = "Frequency",
+         ylim = c(0, 0.3), xlab = "Number of Successes", names.arg = c(factor(seq(0, 10))),
+         col = c(brewer.pal(11, "Spectral")))
```

- Transparency can be adjusted using the alpha argument in *rgb()*.

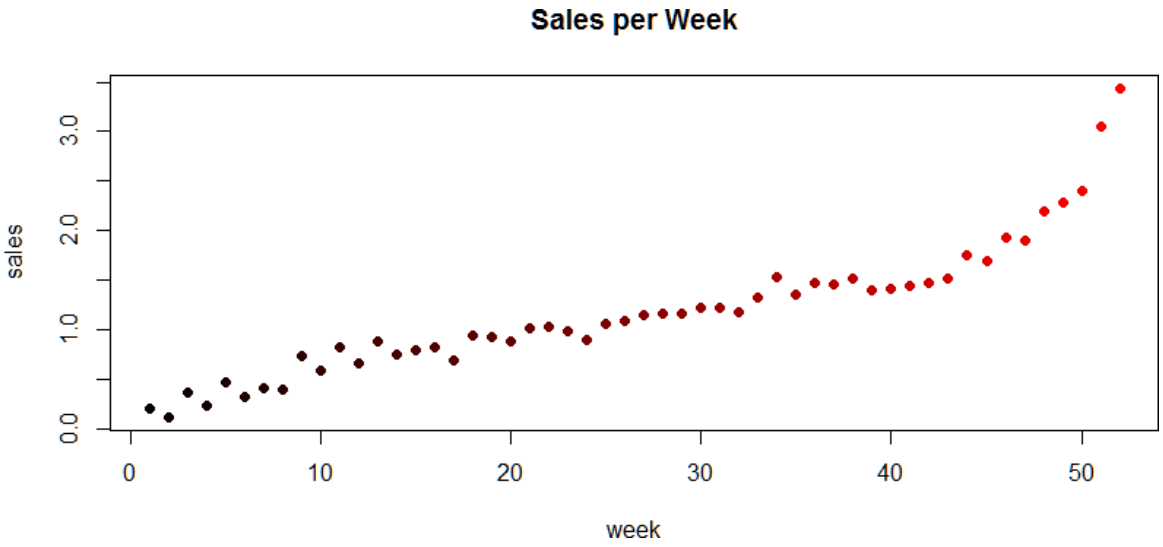
```
> set.seed(123)
> variable <- rnorm(500, mean = 0, sd = 1)
> par(mfrow = c(1, 3))
> hist(variable, ylim = c(0, 120), col = rgb(1, 0, 0, 0.25))
> hist(variable, ylim = c(0, 120), col = rgb(1, 0, 0, 0.5))
> hist(variable, ylim = c(0, 120), col = rgb(1, 0, 0))
> par(mfrow = c(1, 1)
```




- Each point of a plot can have a different color. It is necessary to specify a color list that gives a color for each point. This can be done in various ways such as with a user-defined function, or colorRampPalette() and brewer.pal().

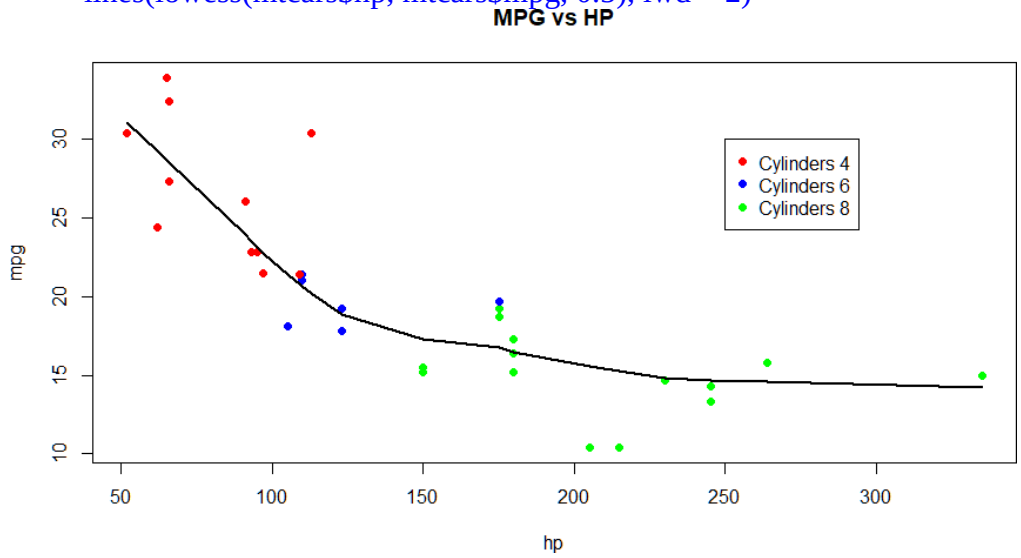
```
> sales <- c(0.70, 0.74, 0.64, 0.39, 0.70, 2.20, 1.98, 0.64, 1.22, 0.20, 1.64, 1.02, 2.95,
+            0.90, 1.76, 1.00, 1.05, 0.10, 3.45, 1.56, 1.62, 1.83, 0.99, 1.56, 0.40, 1.28,
+            0.83, 1.24, 0.54, 1.44, 0.92, 1.00, 0.79, 0.79, 1.54, 1.00, 2.24, 2.50, 1.79,
+            1.25, 1.49, 0.84, 1.42, 1.00, 1.25, 1.42, 1.15, 0.93, 0.40, 1.39, 0.30, 0.35)

> sales <- jitter(sort(sales, decreasing = FALSE),75)
> week <- seq(1:length(sales))
> c <- week/length(week) # This is a use-defined function that specifies a proportion.
>  coly <- rgb(c,0,0)      # This assigns a color to each week.
> plot(week, sales, col=coly, pch=19, cex=1, main = "Sales per Week")
```



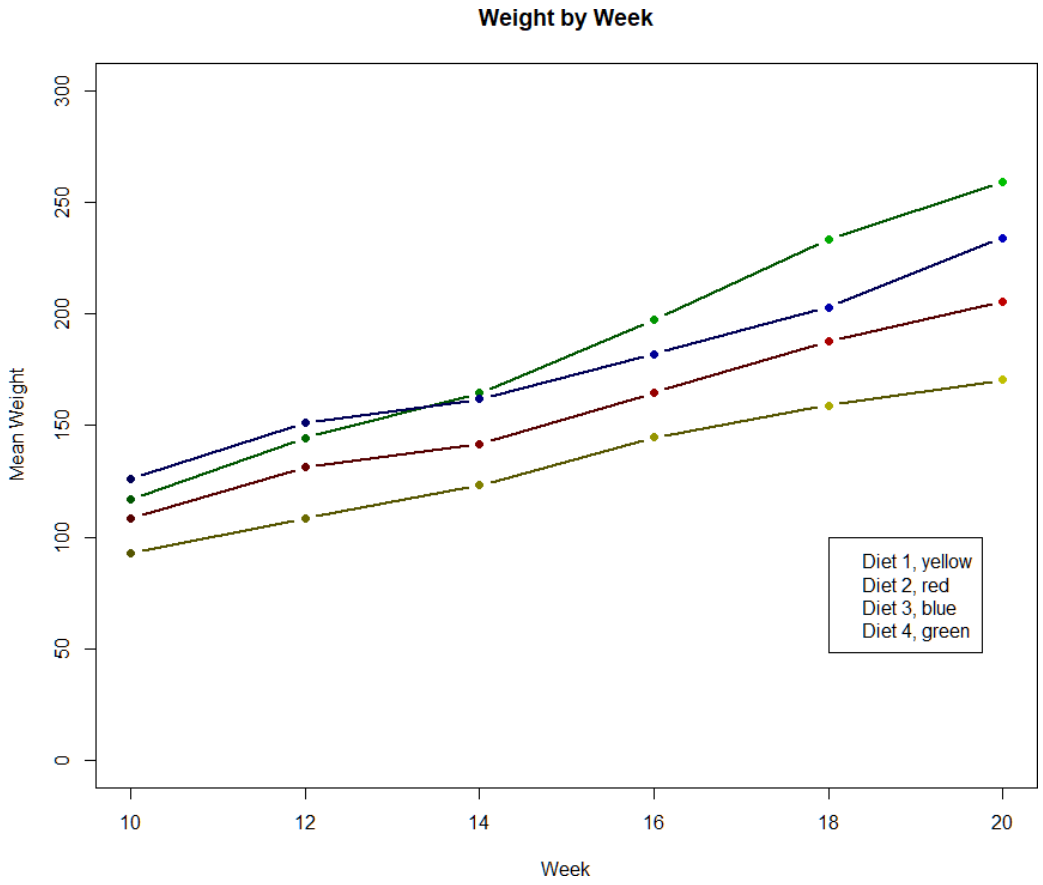

- The use of a color list can be applied when showing group identification in a scatter plot.

```
> data(mtcars)
> col.list <- numeric(0)
> cylinders <- factor(mtcars$cyl)
> hp <- mtcars$hp
> mpg <- mtcars$mpg
> color <- c("red", "blue", "green")
> col.list[cylinders == "4"] <- 1 # Denotes "red" from the palette.
> col.list[cylinders == "6"] <- 2 # Denotes "blue" from the palette.
> col.list[cylinders == "8"] <- 3 # Denotes "green" from the palette.
> plot(hp, mpg, pch = 16, col = c(color[col.list]), main = "MPG vs HP")
> legend(x = 250, y = 30,legend = paste("Cylinders", seq(4, 8, 2)),
+        col = c(color), pch = 16)
> lines(lowess(mtcars$hp, mtcars$mpg, 0.5), lwd = 2)
```



- Color variation can also be used to identify different lines in a plot simultaneously with a time trend. The display below shows average chick weight for four diets over ten weeks.

```
> data(ChickWeight)
> m.weight <- aggregate(weight ~ Time + Diet, data = ChickWeight, mean)
> m.weight <- subset(m.weight, (Time >= 10)&(Time <= 20))
> time <- m.weight$Time
> weight <- m.weight$weight
> diet <- m.weight$Diet
> c <- seq(8, 20, 2) / 24 # creates color variation by week using rgb()
> plot(time[diet == "1"], weight[diet == "1"], type = "b", ylim = c(0,300), pch = 16,
+       xlab = "Week", ylab = "Mean Weight", main = "Weight by Week", col= rgb(c, c, 0), lwd = 2)
> lines(time[diet == "2"], weight[diet == "2"], type = 'b', col = rgb(c, 0, + 0), pch = 16, lwd = 2)
> lines(time[diet == "3"], weight[diet == "3"], type = 'b', col = rgb(0, c, + 0), pch = 16, lwd = 2)
> lines(time[diet == "4"], weight[diet == "4"], type = 'b', col = rgb(0, 0, + c), pch = 16, lwd = 2)
> legend(x = 18, y = 100, legend = paste(c("Diet 1, yellow", "Diet 2, red",
+         "Diet 3, blue", "Diet 4, green")))
```



## Exercises

1. Use RColorBrewer and six colors from the Accent scheme to produce a pie chart with equal area slices. This will require use of *brewer.pal()*.

2. Use *set.seed(123)* and *rnorm(100, 100, sd = 1.5)* to generate six random samples. Use the color scheme from (1) above to produce six side-by-side boxplots.

3. Produce six histograms using the samples from (2). Color each with a different color from the *palette(c("red","purple","blue","green3","yellow","orange"))*.

4. Use *col2rgb()* and determine the *rgb()* representation for the six colors in the palette used in (3) above. Convert the rgb codes into hex codes.

## References


1) Davies, Tilman M. (2016). *The Book of R*, San Francisco, CA: No Starch Press [ISBN-13: 978-1593276515]
2) Kabacoff, R. I. (2015). *R in Action, 2nd ed.* Shelter Island, NY: Manning Co. [ISBN-13: 978-1617291388]