Introduction to R

Release 1.0

Robert Hijmans

May 07, 2018

CHAPTER

ONE

1. INTRODUCTION

R is perhaps the most powerful computer environment for data analysis that is currently available. *R* is both a computer *language*, that allows you to express instructions, and a *program* that responds to these instructions. *R* has core functionality to read and write files, manipulate and summarize data, run statistical tests and models, make fancy plots, and many more things like that. This core functionality is extended by hundreds of packages (plug-ins). Some of these packages provide more advanced generic functionality, others provide highly specialized and cutting-edge methods that are only used in highly specialized analysis.

Because of its versatility, *R* has become very popular across data analysts in many fields, from agronomy to bioinformatics, ecology, finance, geography, pharmacology and psychology. You can read about it in this article in Nature or in the New York Times. So you probably should learn *R* if you want to do modern data analysis, be a successful researcher, collaborate, get a high paying job, ... If you are not that much into *data analysis* but want to learn programming, I would suggest that you learn python instead.

This document provides a concise introduction to *R*. It emphasizes what you need to know to be able to use the language in any context. There is no fancy statistical analysis here. We just present the basics of the *R* language itself. We do not assume that you have done any computer programming before (but we do assume that you think it is about time you did). Experienced *R* users obviously need not read this. But the material may be useful if you want to refresh your memory, if you have not used *R* much, or if you feel confused.

When going through the material, it is very important to follow Norman Matloff's advice: *"When in doubt, try it out!"*. That is, copy the examples shown, and then make modifications to test if you can predict what will happen. Only then will you really understand what is going on. You are learning a language, and you will have to practice a lot to become good at it. And you just have to accept that for a while you will be stumbling.

Before going to the next chapters, we suggest that you first go through the *R* Code School. This will introduce some of the key concepts in an easy and interactive way in less than an hour. This text will then reinforce some of what you learned and take it further. If you like the interactive on-line experience, you might also try Datacamp's introduction to R.

To work with *R* on your own computer, you need to download and install it. I recommend that you also install R-Studio. R-Studio is a separate program that 'wraps around' *R* to make it easier to use. Here is a video that shows how to work in R-Studio.

If you have trouble with the material presented here, you could consult additional resources to learn *R*. There are many free resources on the web, including R for Beginners by Emmanuel Paradis and this tutorial. Or consult this brief overview by Ross Ihaka (one of the originators of R) from his Information Visualization course. You can also pick up an introductory *R* book such as A Beginner's Guide to R by Zuur, Leno and Meesters, R in a nutshell by Joseph Adler, and Norman Matloff's The Art of R Programming.

There is also a lot of very good material on rstatistics.net

If you want to take it easy, or perhaps learn while you commute in a packed train, you could watch some Google Developers videos.

If none of this appeals to you, and you already are experienced with *R*, or you have done a lot of programming with other languages, skip all of this and have a look at Hadley Wickham's Advanced R.

CHAPTER

TWO

2. BASIC DATA TYPES

This chapter briefly discusses the basic data types that are used in *R*. Here we mainly show how to create data of these types. How to manipulate them is described in the following chapters. The most important basic (primitive) data types are the "numeric" and "character" types. Additional types include the "integer", which can be used to represent whole numbers; the "logical" and the "factor". These are all discussed below.

2.1 Numeric and integer values

Let's create a variable `a` that is a vector of one number.

```
a <- 7
```

To do this yourself, type the code in an R console. Or, if you use R-Studio, use 'File / New File / R script' and type it in the new script. Then press "Run" or "Ctrl-Enter" (Apple-Enter on a Mac) to run the line (make sure your cursor is on the line that you want to run).

The "arrow" `<-` was used to **assign** the value `7` to variable `a`. You can pronounce the above as "a becomes 7".

It is also possible to use the = sign.

```
a = 7
```

`<-` is clearer and preferred, because the arrow clearly indicates the assignment action, and because the = sign is also used in another context (to pass arguments to functions).

The name `a` is entirely arbitrary. We could have used `x`, `var`, `fruit` or any other name that would help us recognize it. There are a few restrictions: variable names cannot start with a number, and they cannot contain spaces or "special" characters, such as "*".

To check the value of a, we can ask *R* to `show` or `print` it.

```
show(a)
## [1] 7
print(a)
## [1] 7
```

This is also what happens if you simply type the variable name.

```
a
## [1] 7
```

In *R*, all basic data is stored as a *vector*, a one-dimensional array of *n* values of a certain type. Even a single number is a vector (of length 1). That is why *R* shows that the value of `a` is `[1] 7`. Because 7 is the first element in vector `a`.

We can use the `class` function to find out what type of object `a` is (what class it belongs to).

```
class(a)
## [1] "numeric"
```

numeric means that `a` is a real (decimal) number. Its value is equivalent to `7.000`, but trailing zeros are not printed by default. In a few cases it can be useful, or even necessary, to use integer (whole number) values. To create a vector with a single integer you can either use the `as.integer` function, or append an `L` to the number.

```
a <- as.integer(7)
class(a)
## [1] "integer"
a <- 7L
class(a)
## [1] "integer"
```

To create a vector of several numbers, the `c` (combine) function can be used.

```
b <- c(1.25, 2.9, 3.0)
b
## [1] 1.25 2.90 3.00
```

But to create a regular sequence it is easier to use `:`.

```
d <- 5:9
d
## [1] 5 6 7 8 9
```

In reverse order:

```
6:2
## [1] 6 5 4 3 2
```

The `seq` function can also be used, and adds some additional functionality. For example it allows for different step sizes. In this case we go from 3 to 12, taking steps of 3. Try some variations!

```
e <- seq(from=6, to=12, by=3)
e
## [1]  6  9 12
```

To go in reverse order the `by` argument needs to be negative.

```
seq(from=12, to=0, by=-4)
## [1] 12  8  4  0
```

You can also reverse the order after making the sequence, using the `rev` function.

```
s <- seq(from=0, to=12, by=4)
s
## [1]  0  4  8 12
rev(s)
## [1] 12  8  4  0
```

We will discuss *functions* like `seq` in more detail later. But essentially it is a named procedure that performs a certain task. In this case the name is `seq`, and the task is to create a sequence of numbers. The exact specification of the sequence is modified by the *arguments* that are provided to `seq`, in this case: `from`, `to`, and `by`. If you are unsure what a function does, or which arguments are available, then read the function's help page. You can get to the help page for `seq` by typing `?seq` or `help(seq)`, and likewise for all other functions in *R*.

The `rep` (for repeat) function provides another way to create a vector of numbers. You can repeat a single number, or a sequence of numbers.

```
rep(9, times=5)
## [1] 9 9 9 9 9
rep(5:7, times=3)
## [1] 5 6 7 5 6 7 5 6 7
rep(5:7, each=3)
## [1] 5 5 5 6 6 6 7 7 7
```

2.2 Character values

A character variable is used to represent words. Character values are often referred to as a 'string'.

```
x <- 'Yi'
y <- 'Wong'
class(x)
## [1] "character"
x
## [1] "Yi"
```

To distinguish a character value from a variable name, it needs to be quoted. `'x'` is a character value, but `x` is a variable! Double-quoted `"Yi"` is the same as single-quoted `'Yi'`, but you cannot mix the two in one value: `"Yi'` is not valid. But you can enclose one type of quote inside a pair of the other type. For example, you can do `"Yi's dog"` or `'Wong said "hello" and left'`.

One of the most common mistakes for beginners is to forget the quotes.

```
Yi
## Error in eval(expr, envir, enclos): object 'Yi' not found
```

The error occurs because *R* tries to print the value of variable `Yi`, but there is no such variable. So remember that any time you get the error message `object 'something' not found`, the most likely reason is that you forgot to quote a character value. If not, it probably means that you have misspelled, or not yet created, the variable that you are referring to.

Keep in mind R is a case-sensitive language; `a` is not the same as `A`. In computing, these are two **entirely** different and, for most intents and purposes, unrelated characters.

Now let's create variable `countries` holding a character vector of five elements.

```
countries <- c('China', 'China', 'Japan', 'South Korea', 'Japan')
class(countries)
## [1] "character"
countries
## [1] "China"       "China"       "Japan"       "South Korea" "Japan"
```

The function `length` tells us how long the vector is (how many elements it has).

```
length(countries)
## [1] 5
```

If you want to know the number of characters of each element of the vector, you can use `nchar`.

```
nchar(countries)
## [1]  5  5  5 11  5
```

`nchar` returns a vector of integers with the same length as `x` (5). Each number is the number of characters of the corresponding element of `countries`. This is an example of why we say that most functions in *R* are `vectorized`. This means that you normally do not need to tell *R* to compute things for each individual element in a vector.

It is handy to know that `letters` (a constant value, like `pi`) returns the alphabet (`LETTERS` returns them in uppercase), and `toupper` and `tolower` can be used to change case.

```
z <- letters
z
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
toupper(z)
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Perhaps the most commonly used function for string manipulation is `paste`. This function is used to concatenate strings. For example:

```
girl <- "Mary"
boy <- "John"
paste(girl, "likes", boy)
## [1] "Mary likes John"
```

By default, paste uses a space to separate the elements. You can change that with the `sep` argument.

```
paste(girl, "likes", boy, sep = ' ~ ')
## [1] "Mary ~ likes ~ John"
```

Sometimes you do not want any separator. You can then use `sep=''` or the `paste0` function.

By using the "collapse" argument, we can concatenate all values of a vector into a single element.

```
paste(countries, collapse=' - ')
## [1] "China - China - Japan - South Korea - Japan"
```

We'll leave more advanced manipulation of strings for later, but here are two more important functions. To get a part of a string use 'substr'.

```
substr('Hello World', 1, 5)
## [1] "Hello"
substr('Hello World', 7, 11)
## [1] "World"
```

To replace characters in a string use `gsub` or `sub`.

```
gsub('l', '!!', 'Hello World')
## [1] "He!!!!o Wor!!d"
gsub('Hello', 'Bye bye', 'Hello World')
## [1] "Bye bye World"
```

To find elements that fit a particular pattern use `grep`. It returns the index of the matching elements in a vector.

```
d <- c('az20', 'az21', 'az22', 'ba30', 'ba31', 'ba32')
i <- grep('b', d)
i
## [1] 4 5 6
d[i]
## [1] "ba30" "ba31" "ba32"
```

Which elements of d include the character "2"?

```
grep('2', d)
## [1] 1 2 3 6
```

Which elements of d *end* with the character "2"? "$" has a special meaning.

```
grep('2$', d)
## [1] 3 6
```

Which elements of d *start* with the character "b"? "^" has a special meaning.

```
grep('^b', d)
## [1] 4 5 6
```

2.3 Logical values

A logical (or Boolean) value is either `TRUE` or `FALSE`. They are used very frequently in *R* and in computer programming in general.

```
z <- FALSE
z
## [1] FALSE
class(z)
## [1] "logical"
z <- c(TRUE, TRUE, FALSE)
z
## [1]  TRUE  TRUE FALSE
```

`TRUE` and `FALSE` can be abbreviated to `T` and `F`, but that is very bad practice. This is because it is possible to change the value of `T` and `F` to something else which would be extraordinarily confusing. In contrast, `TRUE` and `FALSE` are constants that cannot be changed.

Logical values are often the result of a computation. For example, here we ask if the values of `x` are larger than 3, which is `TRUE` for values 4 and 5

```
x <- 5
x > 3
## [1] TRUE
```

Likewise we can test for equality using two equal signs == (not = which would be an assignment!). <= means "smaller or equal".

```
x == 3
## [1] FALSE
x <= 2
## [1] FALSE
```

Logical values can be treated as numerical values. `TRUE` is equivalent to 1 and `FALSE` to 0.

```
y <- TRUE
y + 1
## [1] 2
```

However, if you go the other way, only zero is equivalent to `FALSE` while any number that is not zero, is `TRUE`

```
as.logical(0)
## [1] FALSE
as.logical(1)
## [1] TRUE
as.logical(2.5)
## [1] TRUE
```

2.4 Factors

A `factor` is a nominal (categorical) variable with a set of known possible values called `levels`. They can be created using the `as.factor` function. In *R* you typically need to convert (cast) a character variable to a factor to identify groups for use in statistical tests and models.

```
f1 <- as.factor(countries)
f1
## [1] China       China       Japan       South Korea Japan
## Levels: China Japan South Korea
```

But numbers can also be used. For example, they may simply indicate group membership.

```
f2 <- c(5:7, 5:7, 5:7)
f2
## [1] 5 6 7 5 6 7 5 6 7
f2 <- as.factor(f2)
f2
## [1] 5 6 7 5 6 7 5 6 7
## Levels: 5 6 7
```

Dealing with factors can be tricky. For example `f2` created above is not what it may seem. We see numbers 5, 6 and 7, but these are now just labels to identify groups. They cannot be used in algebraic expressions.

We can convert factors to something else. Here we use `as.integer`. If you want a number with decimal places, you can use `as.numeric` instead.

```
f2
## [1] 5 6 7 5 6 7 5 6 7
## Levels: 5 6 7
as.integer(f2)
## [1] 1 2 3 1 2 3 1 2 3
```

The result of as.integer(f2) may have been surprising. But it should not be, as there is no direct link between a category with label "5" and the number 5. In this case, "5" is simply the label of the first category and hence it gets converted to the integer 1. Nevertheless, we can get the numbers back as there is an established link between the character symbol '5' and the number 5. So we first create characters from the factor values, and then numbers from the characters.

```
fc2 <- as.character(f2)
fc2
## [1] "5" "6" "7" "5" "6" "7" "5" "6" "7"
as.integer(fc2)
## [1] 5 6 7 5 6 7 5 6 7
```

This is different from `as.integer(f2)`, which returned the indices of the factor values. It has no way of knowing if you want factor level `6` to represent the number 6.

At this point it is OK if you are confused about factors and *why* you might do such things as conversion from and to them.

2.5 Missing values

All basic data types can have "missing values". These are represented by the symbol `NA` for "not available". For example, we can have vector 'm'

```
m <- c(2, NA, 5, 2, NA, 2)
m
## [1]  2 NA  5  2 NA  2
```

Note that NA is *not* quoted.

2.6 Time

Representing time is a somewhat complex problem. There are different calendars, hours, days, months, and leap years to consider. As a basic introduction, here is simple way to create date values.

```
d1 <- as.Date('2015-4-11')
d2 <- as.Date('2015-3-11')
class(d1)
## [1] "Date"
d1 - d2
## Time difference of 31 days
```

And there are more advanced classes as well that capture date and time.

```
as.POSIXlt(d1)
## [1] "2015-04-11 UTC"
as.POSIXct(d1)
## [1] "2015-04-10 17:00:00 PDT"
```

See http://www.stat.berkeley.edu/~s133/dates.html for more info.

CHAPTER

THREE

3. BASIC DATA STRUCTURES

In the previous chapter we saw the most basic data types in *R*: vectors of numeric, integer, character, factor and boolean values. These were all stored in a vector. In this chapter we look at additional data structures that can store basic data: the `matrix`, `data.frame` and `list`.

3.1 Matrix

A vector is a one-dimensional array. A two-dimensional array can be represented with a matrix. Here is how you can create a matrix with two rows and three columns.

```
matrix(ncol=3, nrow=2)
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

The matrix above did not have any values: all values were missing (`NA`). Let's make a matrix with values 1 to 6.

```
matrix(1:6, ncol=3, nrow=2)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Note that by default the values are distributed column-wise. To go row-wise you can use the `byrow=TRUE` argument.

```
matrix(1:6, ncol=3, nrow=2, byrow=TRUE)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

This can also be achieved by switching the number of columns and rows and using the `t` (transpose) function.

```
m <- matrix(1:6, ncol=2, nrow=3)
t(m)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

It is common to create a matrix by column-binding and/or row-binding vectors using `cbind` and `rbind`. These are two of the most commonly used functions in *R* so pay close attention!

```
a <- c(1,2,3)
b <- 5:7
```

column binding

```
m1 <- cbind(a, b)
m1
##      a b
## [1,] 1 5
## [2,] 2 6
## [3,] 3 7
```

row binding

```
m2 <- rbind(a, b)
m2
##   [,1] [,2] [,3]
## a    1    2    3
## b    5    6    7
```

You can use `cbind` and `rbind` also to combine matrices, as long as the number of rows or columns of the two objects are the same.

```
m3 <- cbind(b, b, a)
m <- cbind(m1, m3)
m
##      a b b b a
## [1,] 1 5 5 5 1
## [2,] 2 6 6 6 2
## [3,] 3 7 7 7 3
```

We can get some of the structural properties of a matrix with functions such as `nrow`, `ncol`, `dim` and `length`.

```
nrow(m)
## [1] 3
ncol(m)
## [1] 5
# dimensions of m (nrow, ncol))
dim(m)
## [1] 3 5
# number of cells, or nrow(m) * ncol(m)
length(m)
## [1] 15
```

Columns have (variable) names that can be changed.

```
# get the column names
colnames(m)
## [1] "a" "b" "b" "b" "a"
# set the column names
colnames(m) <- c('ID', 'X', 'Y', 'v1', 'v2')
m
##      ID X Y v1 v2
## [1,]  1 5 5  5  1
## [2,]  2 6 6  6  2
## [3,]  3 7 7  7  3
```

Likewise there are row names, but these are less important.

```
rownames(m) <- paste0('row_', 1:nrow(m))
m
##       ID X Y v1 v2
## row_1  1 5 5  5  1
## row_2  2 6 6  6  2
## row_3  3 7 7  7  3
```

A matrix can only store a single data type. If you try to mix character and numeric values, all values will become character values (as the other way around may not be possible).

```
cbind(vchar=c('a','b'), vnumb=1:2)
##      vchar vnumb
## [1,] "a"   "1"
## [2,] "b"   "2"
```

You can see that 1 and 2 are character values because they are quoted. You could not use them in algebra without first converting them back to numbers. Note that the column names were set by providing them to `cbind`

A matrix is a two dimensional array. Higher dimensional arrays can also be created. See `help(array)`, but these are not that commonly used, so we do not discuss them here.

3.2 List

A `list` is a very flexible container to store data. Each element of a list can contain any type of *R* object, e.g. a vector, matrix, data.frame, another list, or more complex data types.

A simple list:

```
list(1:3)
## [[1]]
## [1] 1 2 3
```

It shows that the first element `[[1]]` contains a vector of `1, 2, 3`

Here is one with two data types.

```
e <- list(c(2,5), 'abc')
e
## [[1]]
## [1] 2 5
##
## [[2]]
## [1] "abc"
```

List elements can be named.

```
names(e) <- c('first', 'last')
e
## $first
## [1] 2 5
##
## $last
## [1] "abc"
```

And a more complex list.

```
m <- matrix(1:6, ncol=3, nrow=2)
f <- list(e, m, 'abc')
f
## [[1]]
## [[1]]$first
## [1] 2 5
##
## [[1]]$last
## [1] "abc"
##
```

```
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## [[3]]
## [1] "abc"
```

Note that the first element of list `f` is itself a list of two elements.

3.3 Data frame

The `data.frame` is the workhorse for statistical data analysis in *R*. It is rectangular like a matrix, but unlike matrices a `data.frame` can have columns (variables) of different data types. A `data.frame` is what you get when you read spreadsheet-like data into *R* with functions like `read.table` or `read.csv`. We'll show that in a later chapter. We can also create a `data.frame` with some simple code.

```
# four vectors
ID <- as.integer(1:4)
name <- c('Ana', 'Rob', 'Liu', 'Veronica')
sex <- as.factor(c('F','M','M','F'))
score <- c(10.2, 9, 13.5, 18)

d <- data.frame(ID, name, sex, score, stringsAsFactors=FALSE)
d
##   ID     name sex score
## 1  1      Ana   F  10.2
## 2  2      Rob   M   9.0
## 3  3      Liu   M  13.5
## 4  4 Veronica   F  18.0
```

I used the argument `stringsAsFactors=FALSE` to avoid converting the character variable `name` to a factor. `d` is a data.frame, but individual columns can be of any class. Note that the length of a data.frame is defined as the number of variables (columns), while the length of a matrix is defined as the number of cells! This is because a matrix is a special kind of `vector`, while a `data.frame` is a special kind of `list` in which each element has the same size.

```
class(d)
## [1] "data.frame"
length(d)
## [1] 4
```

Because a `data.frame` is a special kind of list, you can do with a data.frame what you can do with a list.

```
is.list(d)
## [1] TRUE
names(d)
## [1] "ID"    "name"  "sex"   "score"
```

But in other ways, a `data.frame` is also similar to a matrix (which normal lists are not).

```
nrow(d)
## [1] 4
dim(d)
## [1] 4 4
colnames(d)
## [1] "ID"    "name"  "sex"   "score"
```

CHAPTER

FOUR

4. INDEXING

There are multiple ways to access or replace values in vectors or other data structures. The most common approach is to use "indexing". This is also referred to as "slicing".

Note that brackets `[ ]` are used for indexing, whereas parentheses `( )` are used to call a function.

4.1 Vector

Here are some examples that show how elements of vectors can be obtained by indexing.

```
b <- 10:15
b
## [1] 10 11 12 13 14 15
# get the first element
b[1]
## [1] 10
# the second element
b[2]
## [1] 11
# elements 2 to 3
b[2:3]
## [1] 11 12
```

Now a more advanced example, return all elements except the second

```
b[-2]
## [1] 10 12 13 14 15
```

You can also use an index to change values

```
b[1] <- 11
b
## [1] 11 11 12 13 14 15
b[3:6] <- -99
b
## [1]  11  11 -99 -99 -99 -99
```

An important characteristic of *R*'s vectorization system is that shorter vectors are 'recycled'. That is, they are repeated until the necessary number of elements is reached. This applies in many circumstances, and is very practical when you are aware of it. It may, however, also lead to undetected errors, when this was not intended to happen.

Here you see recycling at work. First we assign a single number to the first three elements of `b`, so the number is used three times. Then we assign two numbers to a sequence of 3 to 6, such that both numbers are used twice.

```
b[1:3] <- 2
b
## [1]   2   2   2 -99 -99 -99
b[3:6] <- c(10,20)
b
## [1]  2  2 10 20 10 20
```

4.2 Matrix

Consider matrix `m`.

```
m <- matrix(1:9, nrow=3, ncol=3, byrow=TRUE)
colnames(m) <- c('a', 'b', 'c')
m
##      a b c
## [1,] 1 2 3
## [2,] 4 5 6
## [3,] 7 8 9
```

Like vectors, values of matrices can be accessed through indexing. There are different ways to do this, but it is generally easiest to use two numbers in a double index, the first for the row number(s) and the second for the column number(s).

```
# one value
m[2,2]
## b
## 5
# another one
m[1,3]
## c
## 3
```

You can also get multiple values at once.

```
# 2 columns and rows
m[1:2,1:2]
##      a b
## [1,] 1 2
## [2,] 4 5

# entire row
m[2, ]
## a b c
## 4 5 6

# entire column
m[ ,2]
## [1] 2 5 8
```

Or use the column names for sub-setting.

```
#single column
m[, 'b']
## [1] 2 5 8
# two columns
m[, c('a', 'c')]
##      a c
```

```
## [1,] 1 3
## [2,] 4 6
## [3,] 7 9
```

Instead of indexing with two numbers, you can also use a single number. You can think of this as a "cell number". Cells are numbered column-wise (i.e., first the rows in the first column, then the second column, etc.). Thus,

```
m[2,2]
## b
## 5
# is equivalent to
m[5]
## [1] 5
```

Note that

```
m[ ,2]
## [1] 2 5 8
```

returns a vector. This is because a single-column matrix can be simplified to a vector. In that case the matrix structure is 'dropped'. This is not always desirable, and to keep this from happening, you can use the `drop=FALSE` argument.

```
m[ , 2, drop=FALSE]
##      b
## [1,] 2
## [2,] 5
## [3,] 8
```

Setting values of a matrix is similar to how you would do that for a vector, except that you now need to deal with two dimensions.

```
# one value
m[1,1] <- 5
m
##      a b c
## [1,] 5 2 3
## [2,] 4 5 6
## [3,] 7 8 9
# a row
m[3,] <- 10
m
##       a  b  c
## [1,]  5  2  3
## [2,]  4  5  6
## [3,] 10 10 10
# two columns, with recycling
m[,2:3] <- 3:1
m
##       a b c
## [1,]  5 3 3
## [2,]  4 2 2
## [3,] 10 1 1
```

There is a function to get (or set) the values on the diagonal.

```
diag(m)
## [1] 5 2 1
diag(m) <- 0
m
##       a b c
```

```
## [1,]  0 3 3
## [2,]  4 0 2
## [3,] 10 1 0
```

4.3 List

Indexing lists can be a bit confusing as you can both refer to the elements of the list, or the elements of the data (perhaps a matrix) in one of the list elements. Note the difference that double brackets make. `e[3]` returns a list (of length 1), but `e[[3]]` returns what is inside that list element (a matrix in this case)

```
m <- matrix(1:9, nrow=3, ncol=3, byrow=TRUE)
colnames(m) <- c('a', 'b', 'c')
e <- list(list(1:3), c('a', 'b', 'c', 'd'), m)
```

We can access data inside a list element by combining double and single brackets. By using the double brackets, the list structure is dropped.

```
e[2]
## [[1]]
## [1] "a" "b" "c" "d"
e[[2]]
## [1] "a" "b" "c" "d"
```

List elements can have names.

```
names(e) <- c('zzz', 'xyz', 'abc')
```

And the elements can be extracted by their name, either as an index, or by using the `$` (dollar) operator.

```
e$xyz
## [1] "a" "b" "c" "d"
e[['xyz']]
## [1] "a" "b" "c" "d"
```

The `$` can also be used with data.frame objects (a special list, after all), but not with matrices.

4.4 Data.frame

Indexing a `data.frame` can generally be done as for matrices and for lists.

First create a `data.frame` from `matrix` `m`.

```
d <- data.frame(m)
class(d)
## [1] "data.frame"
```

You can extract a column by column number.

```
d[,2]
## [1] 2 5 8
```

Here is an alternative way to address the column number in a `data.frame`.

```
d[2]
##   b
## 1 2
```

```
## 2 5
## 3 8
```

Note that whereas `[2]` would be the second *element* in a `matrix`, it refers to the second *column* in a `data.frame`. This is because a `data.frame` is a special kind of list and not a special kind of matrix.

You can also use the column name to get values. This approach also works for a `matrix`.

```
d[, 'b']
## [1] 2 5 8
```

But with a `data.frame` you can also do

```
d$b
## [1] 2 5 8
# or this
d[['b']]
## [1] 2 5 8
```

All these return a vector. That is, the complexity of the `data.frame` structure was `dropped`. This does not happen when you do

```
d['b']
##   b
## 1 2
## 2 5
## 3 8
```

or

```
d[ , 'b', drop=FALSE]
##   b
## 1 2
## 2 5
## 3 8
```

Why should you care about this `drop` business? Well, in many cases *R* functions want a specific data type, such as a `matrix` or `data.frame` and report an error if they get something else. One common situation is that you think you provide data of the right type, such as a `data.frame`, but that in fact you are providing a `vector`, because the structure `dropped`.

4.5 Which, %in% and match

Sometimes you do not have the indices you need, and so you need to find them. For example, what are the indices of the elements in a vector that have values above 15?

```
x <- 10:20
i <- which(x > 15)
x
##  [1] 10 11 12 13 14 15 16 17 18 19 20
i
## [1]  7  8  9 10 11
x[i]
## [1] 16 17 18 19 20
```

Note, however, that you can also use a logical vector for indexing (values for which the index is `TRUE` are returned).

```
x <- 10:20
b <- x > 15
x
##  [1] 10 11 12 13 14 15 16 17 18 19 20
b
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
x[b]
## [1] 16 17 18 19 20
```

A very useful operator that allows you to ask whether a set of values is present in a vector is `%in%`.

```
x <- 10:20
j <- c(7,9,11,13)
j %in% x
## [1] FALSE FALSE  TRUE  TRUE
which(j %in% x)
## [1] 3 4
```

Another handy similar function is `match`:

```
match(j, x)
## [1] NA NA  2  4
```

telling us that the third value in `j` is equal to the second value in `x` and that the fourth value in 'j' is equal to the fourth value in `x`.

`match` is asymmetric: `match(j,x)` is not the same as `match(x,j)`.

```
match(x, j)
##  [1] NA  3 NA  4 NA NA NA NA NA NA NA
```

This tells us that the second value in `x` is equal to the third value in 'j', etc.

CHAPTER

FIVE

5. ALGEBRA

Vectors and matrices can be used to compute new vectors (matrices) with simple and intuitive algebraic expressions.

5.1 Vector algebra

We have two vectors, `a` and `b`

```
a <- 1:5
b <- 6:10
```

Multiplication works element by element. That is `a[1] * b[1]`, `a[2] * b[2]`, etc

```
d <- a * b
a
## [1] 1 2 3 4 5
b
## [1]  6  7  8  9 10
d
## [1]  6 14 24 36 50
```

The examples above illustrate a special feature of *R* not found in most other programming languages. This is that you do not need to 'loop' over elements in an array (vector in this case) to compute new values. It is important to use this feature as much as possible. In other programming languages you would need to do something like the 'for-loop' below to achieve the above (for-loops do exist in *R* and are discussed in a later chapter).

You can also multiply with a single number.

```
a * 3
## [1]  3  6  9 12 15
```

In the examples above the computations used either vectors of the same length, or one of the vectors had length 1. But be careful, you can use algebraic computations with vectors of different lengths, as the shorter ones will be "recycled". *R* only issues a warning if the length of the longer vector is not a multiple of the length of the shorter object. This is a great feature when you need it, but it may also make you overlook errors when your data are not what you think they are.

```
a + c(1,10)
## Warning in a + c(1, 10): longer object length is not a multiple of shorter
## object length
## [1]  2 12  4 14  6
```

No warning here:

```
1:6 + c(0,10)
## [1]  1 12  3 14  5 16
```

5.1.1 Logical comparisons

Recall that == is used to test for equality

```
a == 2
## [1] FALSE  TRUE FALSE FALSE FALSE
f <- a > 2
f
## [1] FALSE FALSE  TRUE  TRUE  TRUE
```

& is Boolean "AND", and | is Boolean "OR".

```
a
## [1] 1 2 3 4 5
b
## [1]  6  7  8  9 10
b > 6 & b < 8
## [1] FALSE  TRUE FALSE FALSE FALSE
# combining a and b
b > 9 | a < 2
## [1]  TRUE FALSE FALSE FALSE  TRUE
```

"Less than or equal" is <=, and "more than or equal" is >=.

```
b >= 9
## [1] FALSE FALSE FALSE  TRUE  TRUE
a <= 2
## [1]  TRUE  TRUE FALSE FALSE FALSE
b >= 9 | a <= 2
## [1]  TRUE  TRUE FALSE  TRUE  TRUE
b >= 9 & a <= 2
## [1] FALSE FALSE FALSE FALSE FALSE
```

5.1.2 Functions

There are many functions that allow us to do vectorized algebra. For example:

```
sqrt(a)
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
exp(a)
## [1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

Not all functions return a vector of the same length. The following functions return just one or two numbers:

```
min(a)
## [1] 1
max(a)
## [1] 5
range(a)
## [1] 1 5
sum(a)
## [1] 15
mean(a)
## [1] 3
```

```
median(a)
## [1] 3
prod(a)
## [1] 120
sd(a)
## [1] 1.581139
```

If you cannot guess what `prod` and `sd` do, look it up in the help files (e.g. `?sd`)

5.1.3 Random numbers

It is common to create a vector of random numbers in data analysis, and also to create example data to demonstrate how a procedure works. To get 10 numbers sampled from the uniform distribution between 0 and 1 you can do

```
r <- runif(10)
r
##  [1] 0.07003268 0.96431704 0.44251011 0.37027238 0.14118509 0.05419043
##  [7] 0.65782807 0.57816192 0.98710176 0.60379240
```

For Normally distributed numbers, use `rnorm`

```
r <- rnorm(10, mean=10, sd=2)
r
##  [1]  6.971006  9.876585  9.705458 13.083186  8.036289 10.993156 13.393896
##  [8]  9.478527  8.588143  9.677643
```

If you run the functions above, you will get different numbers. After all, they are random numbers! Well, computer generated numbers are not truly random, but 'pseudo-random'. To be able to exactly reproduce examples or data analysis we often want to assure that we take exactly the same "random" sample each time we run our code. To do that we use `set.seed`. This function initializes the random number generator to a specific point. This is illustrated below.

```
set.seed(12)
runif(3)
## [1] 0.06936092 0.81777520 0.94262173
runif(4)
## [1] 0.26938188 0.16934812 0.03389562 0.17878500
runif(5)
## [1] 0.641665366 0.022877743 0.008324827 0.392697197 0.813880559

set.seed(12)
runif(3)
## [1] 0.06936092 0.81777520 0.94262173
runif(5)
## [1] 0.26938188 0.16934812 0.03389562 0.17878500 0.64166537

set.seed(12)
runif(3)
## [1] 0.06936092 0.81777520 0.94262173
runif(5)
## [1] 0.26938188 0.16934812 0.03389562 0.17878500 0.64166537
```

Note that each time `set.seed` is called, the same sequence of (pseudo) random numbers will be generated. This is a very important feature, as it allows us to exactly reproduce results that involve random sampling. The seed number is arbitrary; a different seed number will give a different sequence.

```
set.seed(12)
runif(3)
## [1] 0.06936092 0.81777520 0.94262173
runif(5)
## [1] 0.26938188 0.16934812 0.03389562 0.17878500 0.64166537
```

5.2 Matrices

Computation with matrices is also 'vectorized'. For example, with matrix `m` you can do `m * 5` to multiply all values of m3 with 5, or do `m^2` or `m * m` to square the values of m.

```
# set up an example matrix
m <- matrix(1:6, ncol=3, nrow=2, byrow=TRUE)
m
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6

m * 2
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    8   10   12

m^2
##      [,1] [,2] [,3]
## [1,]    1    4    9
## [2,]   16   25   36
```

We can also do math with a matrix and a vector. Note, again, that computation with matrices in *R* is column-wise, and that shorter vectors are recycled.

```
m * 1:2
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    8   10   12
```

You can multiply two matrices.

```
m * m
##      [,1] [,2] [,3]
## [1,]    1    4    9
## [2,]   16   25   36
```

Note that this is "cell by cell" multiplication. For 'matrix multiplication' in the mathematical sense, you need to use the `%*%` operator.

```
m %*% t(m)
##      [,1] [,2]
## [1,]   14   32
## [2,]   32   77
```

CHAPTER

SIX

6. READ AND WRITE FILES

In most cases, the first step in data analysis is to read in a file with data. This can be pretty complicated due to the variations in file format. Here we discuss reading matrix-like (data.frame/spreadsheet) data structures, which is the most common case and relatively painless.

If you have an Excel file, use "File / Save As" and select "CSV (Comma delimited) (*.csv)". Although it is possible to directly read Excel files, we do not discuss that here.

To read a file into R, you need to know its name. That is, you need to know the full path (directory) name and the name of the file itself. Wrong path names often create confusion. On Windows, it is easy to copy the path from the top bar in Windows Explorer. On a Mac such things are harder, and it may not be so so easy to get a path name. You can do "Select, Finder—View—Show Path Bar" or perhaps drag your file to a text editor (that may paste the file name!).

Below I try to assign a Windows-type full path and file name to a variable `f` so that we can use it.

```
f <- "C:\projects\research\data\obs.csv"
#Error: '\p' is an unrecognised escape in character string starting ""C:\p"
```

Yikes, an error! The problem is the use of backslashes. In R (and elsewhere), the backslash is the "escape" symbol, which is followed by another symbol to indicate a special character. For example, `"\t"` represents a "tab" and `"\n"` is the symbol for a new line (hard return). This is illustrated below.

```
txt <- "Here is an example:\nA new line has started!\nAnd another one...\n"
message(txt)
## Here is an example:
## A new line has started!
## And another one...
```

So for path delimiters we need to use either the forward-slash `"/"` or an escaped back-slash `"\\"`. Both of the following are OK.

```
f1 <- "C:/projects/research/data/obs.csv"
f2 <- "C:\\projects\\research\\data\\obs.csv"
```

Now this file may not actually exist:

```
file.exists(f1)
## [1] FALSE
```

To make what you see here reproducible, without the need to download a file, we'll first create a file. First some example data:

```
d <- data.frame(id=1:10, name=letters[1:10], value=seq(10,28,2))
d
##    id name value
## 1   1    a    10
## 2   2    b    12
```

```
## 3   3    c    14
## 4   4    d    16
## 5   5    e    18
## 6   6    f    20
## 7   7    g    22
## 8   8    h    24
## 9   9    i    26
## 10 10    j    28
```

And now we write that to disk. Here I show how to do this with two different (but related) functions: `write.csv` and `write.table`.

```
write.csv(d, 'test.csv', row.names=FALSE)
write.table(d, 'test.dat', row.names=FALSE)
```

`write.csv` is a derived from `write.table`. The main difference between the two is that in `write.csv` the field separator is a comma ("csv" stands for "comma separated values"), while in `write.table` the default is a tab (the `\t` character).

Now we have two files on disk.

```
file.exists('test.csv')
## [1] TRUE
file.exists('test.dat')
## [1] TRUE
```

As we only specified a file name, but not a path, the files are in our working directory. We can use `getwd`(get working directory) to see where that is.

```
getwd()
## [1] "e:/bitbucket/rspatial-web/source/intr/_R"
```

The working directory is the 'default' location where *R* will look for files and write files. To get the full path name for one of the files we created, we can use `file.path`.

```
file.path(getwd(), 'test.csv')
## [1] "e:/bitbucket/rspatial-web/source/intr/_R/test.csv"
```

As this is a '.csv' (comma separated values) file we can read it with `read.csv`.

```
d <- read.csv('test.csv', stringsAsFactors=FALSE)
head(d)
##   id name value
## 1  1    a    10
## 2  2    b    12
## 3  3    c    14
## 4  4    d    16
## 5  5    e    18
## 6  6    f    20
```

Sometimes values are delimited by other characters. In that case you can try `read.table`. `read.table` has an argument `sep` (separator) to indicate the field delimiter. Here we do not use that, because the default (whitespace, tab) works fine.

```
d <- read.table('test.dat', stringsAsFactors=FALSE)
head(d)
##   V1   V2    V3
## 1 id name value
## 2  1    a    10
## 3  2    b    12
```

```
## 4  3    c    14
## 5  4    d    16
## 6  5    e    18
```

Note that `read.table` did not automatically recognize the header row with the variable names. Instead, the variable names have become the first records, and new dummy variable names `V1`, `V2` and `V3` were assigned. To fix that, we use the `header` argument:

```
d <- read.table('test.dat', header=TRUE, stringsAsFactors=FALSE)
head(d)
##   id name value
## 1  1    a    10
## 2  2    b    12
## 3  3    c    14
## 4  4    d    16
## 5  5    e    18
## 6  6    f    20
```

`read.table` can also read csv files; you just need to tell it that the field delimiter is a comma.

```
d <- read.table('test.dat', sep=',', stringsAsFactors=FALSE)
```

In the examples above, I used `stringsAsFactors=FALSE`. This is not required, but it is helpful. Otherwise, all character variables are converted to factors, which in many cases is a nuisance. If `read.table` fails, there are other options, such as `readLines`.

```
d <- readLines('test.csv')
class(d)
## [1] "character"
d
##  [1] "\"id\",\"name\",\"value\"" "1,\"a\",10"
##  [3] "2,\"b\",12"                "3,\"c\",14"
##  [5] "4,\"d\",16"                "5,\"e\",18"
##  [7] "6,\"f\",20"                "7,\"g\",22"
##  [9] "8,\"h\",24"                "9,\"i\",26"
## [11] "10,\"j\",28"
```

But this makes it more difficult to process the data. You may need to do something like:

```
x <- sapply(d, function(i){ unlist(strsplit(i, ',')) }, USE.NAMES=FALSE)
t(x)
##       [,1]     [,2]       [,3]
##  [1,] "\"id\"" "\"name\"" "\"value\""
##  [2,] "1"      "\"a\""    "10"
##  [3,] "2"      "\"b\""    "12"
##  [4,] "3"      "\"c\""    "14"
##  [5,] "4"      "\"d\""    "16"
##  [6,] "5"      "\"e\""    "18"
##  [7,] "6"      "\"f\""    "20"
##  [8,] "7"      "\"g\""    "22"
##  [9,] "8"      "\"h\""    "24"
## [10,] "9"      "\"i\""    "26"
## [11,] "10"     "\"j\""    "28"
```

And then some. (`sapply` is explained later on).

For some files, readLines is the way to go. For example, you may want to read 'html' files if you are scraping a website. We can read this very page:

```
webpage <- readLines("http://rspatial.org/intr/rst/6-files.html")
#show some of the first lines:
webpage[150:155]
## [1] "  <div class=\"section\" id=\"read-and-write-files\">"
## [2] "<h1>6. Read and write files<a class=\"headerlink\" href=\"#read-and-write-files\" title=\"Pe
## [3] "<p>In most cases, the first step in data analysis is to read in a file with"
## [4] "data. This can be pretty complicated due to the variations in file"
## [5] "format. Here we discuss reading matrix-like (data.frame/spreadsheet)"
## [6] "data structures, which is the most common case and relatively painless.</p>"
```

Another relevant function in this context is `list.files`. That shows you which files are present in your working directory (or another path of choice) and perhaps all other subdirectories (when using the argument `recursive=TRUE`). By doing something like `ff <- list.files(pattern = 'csv$')` you can get a vector of all files with names that end with "csv" (that is what the `$` sign is for).

Before writing files (e.g. with `write.csv`, `write.table`, or `writeLines`), you may want to use `dir.exists` to assure that the path you are using exists and perhaps use `dir.create` if it does not.

Filenames need to be the *full* file name, including the path, unless you are reading from or writing to the working directory. You can set the working directory with `setwd`. For example: `setwd("C:/projects/research/)`.

CHAPTER

SEVEN

7. DATA EXPLORATION

After reading data from a file (see the previous chapter), the next thing to do is to look at some summary statistics. This is in the first place just to check your data. Very often you discover some strange values that are not quite right. Careful inspection of your data after you read it is very important. It can help avoid a lot of trouble later on when you are trying to explain odd results!

Sometimes you can correct errors in *R* but in other cases you first need to fix the data file. Fixing a data file can be the way to go if you are dealing with your own primary data. However when you are working with a file provided by someone else it is generally better to not change the file, but to correct things via *R* code, if possible. That leaves an exact trail of what you have done, and allows you to apply the same corrections to a new version of the data.

Inspecting your data is also important to understand it better, and such "exploratory data analysis" can be an important step before doing the specific analyses of interest. There are many general and specialized tools for this, and here we only cover some of the very basic tools.

7.1 Summary and table

Consider `data.frame` `d`

```
d <- data.frame(id=1:10,
        name=c('Bob', 'Bobby', '???', 'Bob', 'Bab', 'Jim', 'Jim', 'jim', '', 'Jim'),
        score1=c(8, 10, 7, 9, 2, 5, 1, 6, 3, 4),
        score2=c(3,4,5,-999,5,5,-999,2,3,4), stringsAsFactors=FALSE)
d
##    id  name score1 score2
## 1   1   Bob      8      3
## 2   2 Bobby     10      4
## 3   3   ???      7      5
## 4   4   Bob      9   -999
## 5   5   Bab      2      5
## 6   6   Jim      5      5
## 7   7   Jim      1   -999
## 8   8   jim      6      2
## 9   9             3      3
## 10 10   Jim      4      4
```

`d` is very small and we can simply look at the values in `d` to see if they look all-right. But with real data you may have 100s or 1000s of rows and many columns, making it very hard, if not impossible, to spot errors by eye-balling.

The summary function is an easy way to start, at least for numeric variables.

```
summary(d)
##        id            name               score1          score2
##  Min.   : 1.00   Length:10          Min.   : 1.00   Min.   :-999.00
```

```
##  1st Qu.: 3.25   Class :character   1st Qu.: 3.25   1st Qu.:   2.25
##  Median : 5.50   Mode  :character   Median : 5.50   Median :   3.50
##  Mean   : 5.50                      Mean   : 5.50   Mean   :-196.70
##  3rd Qu.: 7.75                      3rd Qu.: 7.75   3rd Qu.:   4.75
##  Max.   :10.00                      Max.   :10.00   Max.   :   5.00
```

The minimum value of variable `score2` is -999. That was probably used in data entry to indicate a missing value. These should be changed to `NA`.

```
# which values in score2 are -999?
i <- d$score2 == -999
# set these to NA
d$score2[i] <- NA
summary(d)
##        id            name               score1          score2
##  Min.   : 1.00   Length:10          Min.   : 1.00   Min.   :2.000
##  1st Qu.: 3.25   Class :character   1st Qu.: 3.25   1st Qu.:3.000
##  Median : 5.50   Mode  :character   Median : 5.50   Median :4.000
##  Mean   : 5.50                      Mean   : 5.50   Mean   :3.875
##  3rd Qu.: 7.75                      3rd Qu.: 7.75   3rd Qu.:5.000
##  Max.   :10.00                      Max.   :10.00   Max.   :5.000
##                                                     NA's   :2
```

The two steps used above: `i <- d$score2 == -999` and `d$score2[i] <- NA` are usually done in a single line: `d$score2[d$score2 == -999] <- NA`.

For character (and integer) variables it can be useful to use `unique` and `table`:

```
unique(d$name)
## [1] "Bob"   "Bobby" "???"   "Bab"   "Jim"   "jim"   ""
table(d$name)
##
##         ???   Bab   Bob Bobby   jim   Jim
##     1     1     1     2     1     1     3
```

Often you will discover slight variations in spelling that need to be corrected. In this case, let's assume that 'Bobby' and 'Bab' should both be 'Bob'. We replace 'Bab' and 'Bobby' with 'Bob'.

```
d$name[d$name %in% c('Bab', 'Bobby')] <- 'Bob'
table(d$name)
##
##     ??? Bob jim Jim
##   1   1   4   1   3
```

'jim' should be "Jim". It is easy enough to replace as done above. But what if there were many cases like that? It would be easy to make all character values lower- or uppercase with `d$name <- toupper(d$name)` but I want to keep the normal name capitalization of the first letter only. So let's assure that all names start with an uppercase letter.

```
# get the first letters
first <- substr(d$name, 1, 1)
# get the remainder
remainder <- substr(d$name, 2, nchar(d$name))
# assure that the first letter is upper case
first <- toupper(first)
# combine
name <- paste0(first, remainder)
# assign back to the variable
d$name <- name
table(d$name)
```

```
##
##      ??? Bob Jim
##    1   1   4   4
```

The question marks in `name` should probably also be replaced with `NA`.

```
d$name[d$name == '???'] <- NA
table(d$name)
##
##      Bob Jim
##    1   4   4
```

You can force `table` to also count the `NA` values:

```
table(d$name, useNA='ifany')
##
##        Bob  Jim <NA>
##     1    4    4    1
```

Note that there is one 'empty' value.

```
d$name[9]
## [1] ""
```

That should also be a missing value in this case.

```
d$name[d$name == ''] <- NA
table(d$name, useNA='ifany')
##
##  Bob  Jim <NA>
##    4    4    2
```

You can also use `table` to make a contingency table of two variables.

```
table(d[ c('name', 'score2')])
##       score2
## name  2 3 4 5
##   Bob 0 1 1 1
##   Jim 1 0 1 1
```

7.2 Quantile, range, and mean

Other useful functions include `quantile`, `range`, and `mean`.

```
quantile(d$score1)
##    0%   25%   50%   75%  100%
##  1.00  3.25  5.50  7.75 10.00
range(d$score1)
## [1]  1 10
mean(d$score1)
## [1] 5.5
```

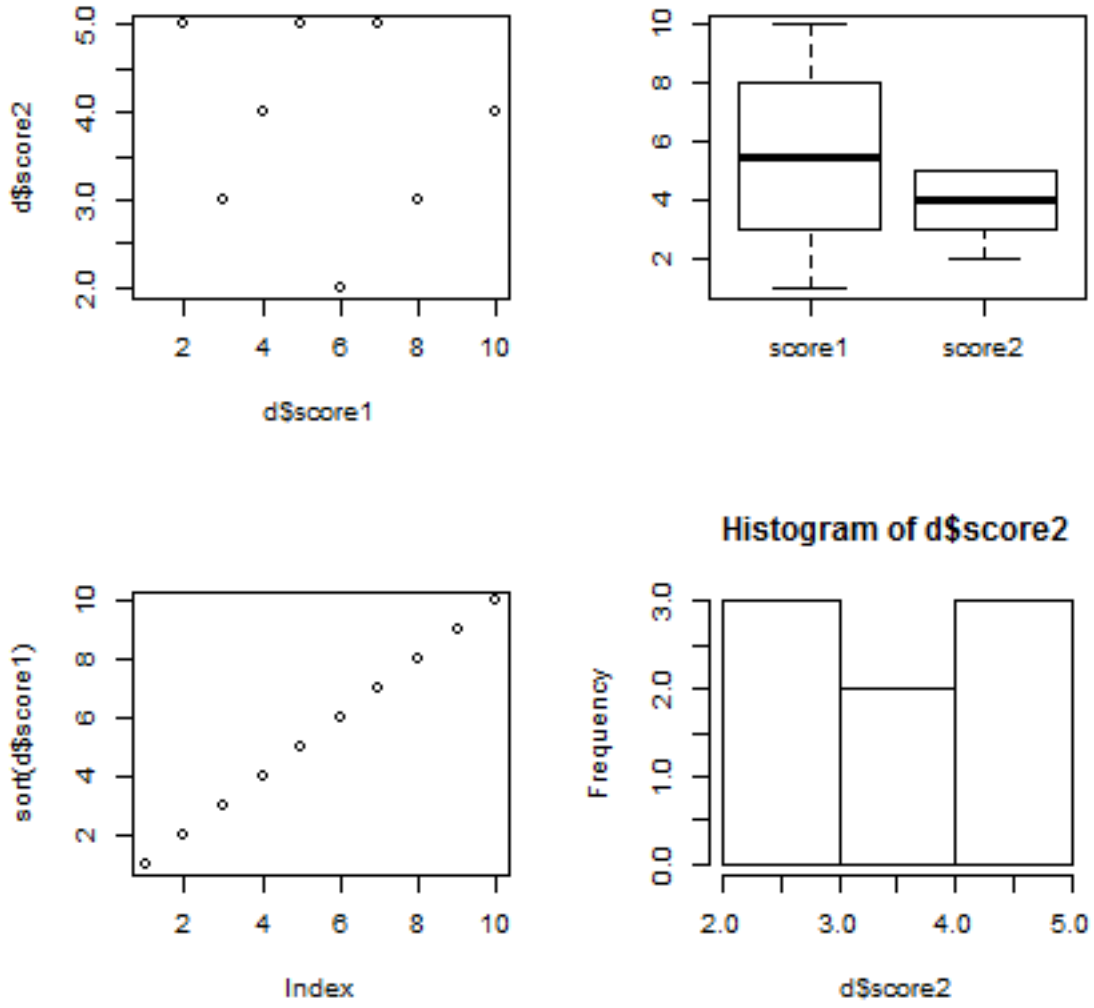
Note that in some functions you may need to use `na.rm=TRUE` if there are `NA` values. Otherwise, as soon as there is a single `NA` value in a computation, the results becomes `NA`. This is very common in *R* — so keep that in mind if all your results are `NA`.

```
quantile(d$score2)
## Error in quantile.default(d$score2): missing values and NaN's not allowed if 'na.rm' is FALSE
```

```
range(d$score2)
## [1] NA NA
quantile(d$score2, na.rm=TRUE)
##   0%  25%  50%  75% 100%
##    2    3    4    5    5
range(d$score2, na.rm=TRUE)
## [1] 2 5
```

In this data exploration phase it is very useful to make plots. We'll discuss plotting in a later chapter, but here are four example plots. Note how `par(mfrow=c(2,2))` sets up the canvas for two rows and columns, that is for four plots.

```
par(mfrow=c(2,2))
plot(d$score1, d$score2)
boxplot(d[, c('score1', 'score2')])
plot(sort(d$score1))
hist(d$score2)
```



CHAPTER

EIGHT

8. FUNCTIONS

We now have used many functions that come with *R*. For example `c`, `matrix`, `read.csv`, and `sum`. Functions are always used ('called') by typing their name, followed by parenthesis. In most, but not all, cases you supply 'arguments' within the parenthesis. If you do not type the parenthesis the function is not called. Instead, either the function definition or some of type of reference to it is shown.

8.1 Existing functions

To see the content of a function, type its name:

```
nrow
## function (x)
## dim(x)[1L]
## <bytecode: 0x000000000eadadb8>
## <environment: namespace:base>
```

We see that `nrow` has a single argument called `x`. It calls another function, `dim` to which it provides the same argument (`x`) and returns its first element (`1L`) (recall that adding `L` ('literal') is a way to create an integer). Can you guess how `ncol` is implemented? (See for yourself if you are right!). Now, let's see what `dim` looks like.

```
dim
## function (x)  .Primitive("dim")
```

It is a 'primitive' (low level) *R* function that we cannot easily learn more about. Well, you could, by looking at the source code of *R* — but that is way out of scope of this tutorial.

To run (instead of inspect) `nrow` we add parentheses:

```
nrow()
## Error in nrow(): argument "x" is missing, with no default
```

But this fails, because the function requires a valid argument, like this:

```
m <- matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
nrow(m)
## [1] 2
```

Note `nrow(m)` and that this is equivalent to

```
nrow(x=m)
## [1] 2
```

because the first argument of `nrow` is called `x`.

8.2 Writing functions

R comes with thousands of functions for you to use. Nevertheless, it is often necessary to write your own functions. For example, you may want to write a function to:

- more clearly describe and isolate a particular task in your data analysis workflow.
- re-use code. Rather than repeating the same steps several times (e.g. for each of 200 cases you are analysing), you can write a function that gets called 200 times. This should lead to faster development of scripts and to fewer mistakes. And if there is a mistake it only needs to be fixed in one place.
- create a function that is an argument to another function (!). This is quite commonly done when using 'apply' type functions (see next chapter).

Writing your own functions is not difficult. The below is a very simple function. It is called `f`. This is an entirely arbitrary name. You can also call it `myFirstFunction`. It takes no arguments, and always returns 'hello'.

```
f <- function() {
    return('hello')
}
```

Look carefully how we assign a function to name `f` using the `function` keyword followed by parenthesis that enclose the arguments (there are none in this case). The *body* of the function is enclosed in braces (also known as "curly brackets" or "squiggly brackets").

Now that we have the function, we can inspect it, and use it.

```
#inspect
f
## function() {
##  return('hello')
## }
## <environment: 0x000000001f5d46b8>
#use 2 times
f()
## [1] "hello"
f()
## [1] "hello"
```

`f` is a very boring function. It takes no arguments and always returns the same result. Let's make it more interesting.

```
f <- function(name) {
    x <- paste('hello', name)
    return(x)
}

f('Jasmin')
## [1] "hello Jasmin"
```

Note the `return` statement. This indicates that variable `x` (which is only known inside of the function) is returned to the caller of the function. Simply typing `x` would also suffice, and ending the function with `paste('hello', name)` would also do! So the below is equivalent but shorter, at the expense of being less explicit.

```
f <- function(name) {
    paste('hello', name)
}

f('Sviatoslav')
## [1] "hello Sviatoslav"
```

Here is a function that returns a sequence of letters. The length is determined by argument `n`.

```
frs <- function(n) {
    s <- sample(letters, n, replace=TRUE)
    r <- paste0(s, collapse='')
    return(r)
}
```

Because the function uses randomization, I use `set.seed` to always get the same result (as we discussed here.

```
set.seed(0)
frs(5)
## [1] "xgjox"
frs(5)
## [1] "fxyrq"
x <- frs(10)
x
## [1] "bferjumszj"
```

Now an example of a functions that manipulates numbers. This function squares the sum of two numbers.

```
sumsquare <- function(a, b) {
    d <- a + b
    dd <- d * d
    return(dd)
}
```

We can now use the sumsquare function. Note that it is vectorized (each argument can be more than one number)

```
sumsquare(1,2)
## [1] 9
x <- 1:3
y <- 5
sumsquare(x,y)
## [1] 36 49 64
```

You can name the arguments when using a function; that often makes your intentions clearer.

```
sumsquare(a=1, b=2)
## [1] 9
```

But the names must match

```
sumsquare(a=1, d=2)
## Error in sumsquare(a = 1, d = 2): unused argument (d = 2)
```

And both arguments need to be present

```
sumsquare(1:5)
## Error in sumsquare(1:5): argument "b" is missing, with no default
```

Unless we redefine the function with default arguments that will be used if a value for the argument is not provided.

```
sumsquareD <- function(a=0, b=1) {
    d <- a + b
    dd <- d * d
    return(dd)
}

sumsquareD(1:5, 2)
## [1]  9 16 25 36 49
```

As both arguments have a default value, we can call `sumsquareD` without providing arguments

```
sumsquareD()
## [1] 1
```

Or with a single argument

```
sumsquareD(5)
## [1] 36
```

Above the value `5` was assigned to argument `a` because the argument was matched "by position". If we only wanted to provide a value for `b`, we need to match "by name".

```
sumsquareD(b=3)
## [1] 9
```

Just another example, a function to compute the number of unique values in a vector:

```
nunique <- function(x) {
    length(unique(x))
}

data <- c('a', 'b', 'a', 'c', 'b')
nunique(data)
## [1] 3
```

Of course, these were toy examples, but if you understand these, you should be able to write much longer and more useful functions. It can be difficult to "debug" (find errors in) a function. It is often best to first write the sequence of commands that you need outside a function, and only when it all works, wrap that code inside of a function block (`function( ) { }`).

8.3 Ellipses (...)

Ellipses `...` are a special argument to many functions. It allows to pass optional additional arguments and/or arguments that are passed on to other functions. Consider these two functions (this is a bit advanced).

```
f1 <- function(x, y=10) {
    x * y
}

# f2 calls f1
f2 <- function(x, ...) {
    f1(x, ...)
}

f2(5)
## [1] 50
f2(5, y=5)
## [1] 25
```

Even though `f2` does not have an argument `y` it can be provided and it is passed on to `f1`. This call returns an error :

```
f2(5, z=5)
## Error in f1(x, ...): unused argument (z = 5)
```

because `f1` does not have an argument `z`.

8.4 Functions overview

A list of much used functions that we discuss in this introduction to *R*:

`c`, `cbind`, `rbind` `length`, `dim`, `nrow`, `ncol`

`sum`, `mean`, `prod`, `sqrt`

`apply`, `sapply`, `tapply`, `aggregate` `rowSums`, `rowMeans`

`merge`, `reshape`

Also see this cheatsheet

CHAPTER

NINE

9. APPLY

The "apply family" of functions (`apply`, `tapply`, `lapply` and others) are central to using *R*. They provide an concise, elegant and efficient approach to apply (sometimes referred to as "mapping") a function to a set of cases, be they rows or columns in a matrix or data.frame, or elements in a list. ## apply

Consider matrix `m`

```
m <- matrix(1:15, nrow=5, ncol=3)
m
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15
```

9.1 apply

Computation with matrices is 'vectorized'. For example you can do `m * 5`to multiply all values of m with 5 or do `m^2` or `m * m` to square the values of `m`. But often we need to compute values for the margins of a matrix, that is, a single value for each row or column. The `apply` function can be used for that:

```
# sum values in each row
apply(m, 1, sum)
## [1] 18 21 24 27 30

# get mean for each column
apply(m, 2, mean)
## [1]  3  8 13
```

Note that the apply uses at least three arguments: a matrix, a 1 or 2 indicating whether the computation is for rows or for columns, and a function that computes a new value (or values) for each row or column. You can read more about this in the help file of the function (type `?apply`). In most cases you will also add the argument `na.rm=TRUE` to remove `NA` (missing) values as any computation that includes an `NA` value will return `NA`. In this case we used existing basic functions `mean` and `sum` but we could also supply a function that we wrote ourselves.

Note that `apply`(and related functions such as `tapply` and `sapply` are ways to avoid writing a loop. In the `apply` examples above you could have written a loop to do the computations row by row (or column by column) but using `apply` is more compact and efficient.

The rowSums and colSums functions are (fast) shorthand functions for apply(, , sum)

```
rowSums(m)
## [1] 18 21 24 27 30
```

9.2 tapply

`tapply` can be used to compute a summary statistic, e.g. a mean value, for groups of rows in a data.frame. You need one column that indicates the group, and then you can compute, for example, the mean value for that group.

```
colnames(m) <- c('v1', 'v2', 'v3')
d <- data.frame(name=c('Yi', 'Yi', 'Yi', 'Er', 'Er'), m, stringsAsFactors=FALSE)
d$v2[1] <- NA
d
##   name v1 v2 v3
## 1   Yi  1 NA 11
## 2   Yi  2  7 12
## 3   Yi  3  8 13
## 4   Er  4  9 14
## 5   Er  5 10 15
```

Imagine that you would like to compute the average value of `v1`, `v2` and `v3` for each individual (`name`). You can use `tapply` for that.

```
tapply(d$v1, d$name, mean)
##  Er  Yi
## 4.5 2.0
tapply(d$v1, d$name, max)
## Er Yi
##  5  3
tapply(d$v2, d$name, mean)
##  Er  Yi
## 9.5  NA
tapply(d$v2, d$name, mean, na.rm=TRUE)
##  Er  Yi
## 9.5 7.5
```

9.3 aggregate

`aggregate` is similar to `tapply` but more convenient if you want to compute a summary statistic for multiple variables. It does have the annoying problem that the second argument cannot be a vector:

```
aggregate(d[, c('v1', 'v2', 'v3')], d$name, mean, na.rm=TRUE)
## Error in aggregate.data.frame(d[, c("v1", "v2", "v3")], d$name, mean, : 'by' must be a list
```

You can fix that in two ways

```
aggregate(d[, c('v1', 'v2', 'v3')], d[, 'name', drop=FALSE], mean, na.rm=TRUE)
##   name  v1  v2   v3
## 1   Er 4.5 9.5 14.5
## 2   Yi 2.0 7.5 12.0
# or
aggregate(d[, c('v1', 'v2', 'v3')], list(d$name), mean, na.rm=TRUE)
##   Group.1  v1  v2   v3
## 1      Er 4.5 9.5 14.5
## 2      Yi 2.0 7.5 12.0
```

As explained before, this is why the first one works: when you extract a single column from a `matrix` or `data.frame`, the structure (class) "drops" to a simpler form, it becomes a vector. `drop=FALSE` stops that from happening.

9.4 sapply and lapply

To iterate over a list, we can use `lapply` or `sapply` . The difference is that `lapply` always returns a list while `sapply` tries to simplify the result to a vector or matrix.

```
names <- list('Antoinette', 'Mary', 'Duncan', 'Obalaya', 'Jojo')
nchar('Jim')
## [1] 3

lapply(names, nchar)
## [[1]]
## [1] 10
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 4
sapply(names, nchar)
## [1] 10  4  6  7  4
```

In all cases `(t)(s)(l)apply` and `aggregate` (and many more functions) we provided some data and a function, such as `mean` or `nchar`. You can also provide your own custom function. For example

```
shortname <- function(name) {
    if (nchar(name) < 5) {
        name <- toupper(name)
        return(name)
    } else {
        name <- substr(name,1,5)
        return(name)
    }
}

sapply(names, shortname)
## [1] "Antoi" "MARY"  "Dunca" "Obala" "JOJO"
```

More examples: https://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/

CHAPTER

TEN

10. FLOW CONTROL

Most programs have two general control-flow features: iteration and alternation. Iteration is done via "loops", alteration via "if-then-else" branches.

10.1 Looping

Loops are typically used to repeat the same code a number of times for a set of cases. For example, compute the average grade for each student.

In *R* we avoid loops wherever we can, as they tend to be slower than 'vectorized' computation. We also generally prefer functions like `apply` (see the previous chapter), as this is more concise. At first code using for-loops may seem easier to read, but after using *R* for a while, the reverse is true is most cases. Nevertheless, there are cases where loops are much easier to write and clearer to read than using vectorized approaches.

There are two types of loops: 'for-loops' and 'while-loops'. A 'for-loop' repeats the same code a predefined number of times. A 'while-loop' continues until a certain condition has been met (and is therefore prone to the dreaded "infinite loop" that never finishes!)

10.1.1 for-loops

Here is a basic *for-loop* that does not do anything useful. The trick is that in the parenthesis after `for`, you define a sequence of values. This sets the number of repetitions (the length of the sequence) and potentially provides a value that changes what is done in each loop.

Note that the braces `{ }` are used to open and close a 'block' of code.

```
for (i in 1:3) {
    print('hi')
}
## [1] "hi"
## [1] "hi"
## [1] "hi"
```

Now let's do something with `i`. You normally do not use `print` inside a loop. I am only doing that to illustrate what is going on.

```
j <- 0
for (i in 1:3) {
    print(i)
    j <- j + i
}
## [1] 1
## [1] 2
```

```
## [1] 3
j
## [1] 6
```

The loop above was used to sum the values 1, 2, and 3. Of course, it would have been easier to use `sum(1:3)`.

Another example.

```
for (i in 1:3) {
    txt <- paste('the square of', i, 'is', i * i)
    print(txt)
}
## [1] "the square of 1 is 1"
## [1] "the square of 2 is 4"
## [1] "the square of 3 is 9"
```

The example below is a bit more complex. It shows how iterator `i` is typically used in a loop (to get a single case from a collection of cases and compute a new value for that case).

```
s <- 0
a <- 1:6
b <- 6:1

# initialization of output variables
res <- vector(length=length(a))

# i goes from 1 to 6 (the length of b)
for (i in 1:length(b)) {
  s <- s + a[i]
  res[i] <- a[i] * b[i]
}
s
## [1] 21
res
## [1]  6 10 12 12 10  6
```

Again, for this simple problem, it would have been simpler to do `s <- sum(a)` and `res <- a * b`.

10.1.2 break and next

Sometimes you want to include a condition to either "break out" of a for loop, or to skip the remainder of the for block and go to the next iteration. How to do that is illustrated below:

```
for (i in 1:10) {
    if (i %in% c(1,3,5,7)) {
        next
    }
    if (i > 8) {
        break
    }
    print(i)
}
## [1] 2
## [1] 4
## [1] 6
## [1] 8
```

10.1.3 while-loops

"while-loops" are not nearly as common as "for-loops". Here is an example.

```
i <- 0
while (i < 4) {
   print(paste(i, 'and counting ...'))
   i <- i + 1
}
## [1] "0 and counting ..."
## [1] "1 and counting ..."
## [1] "2 and counting ..."
## [1] "3 and counting ..."
```

And one that is less predictable, as it depends on the value of a random number.

```
set.seed(1)
i <- 0
while(i < 0.5) {
    i <- runif(1)
    print(i)
}
## [1] 0.2655087
## [1] 0.3721239
## [1] 0.5728534
```

You can also combine `while` with `break`.

```
set.seed(1)
while(TRUE) {
    i <- runif(1)
    print(i)
    if (i > 0.5) {
        break
    }
}
## [1] 0.2655087
## [1] 0.3721239
## [1] 0.5728534
```

10.2 Branching

Branching is an important mechanism in computer programs. A branch allows you to execute some code if certain conditions are met, and do something else in other cases. This is illustrated below. Note that the braces `{ }` are used to open and close a 'block' of code.

We have two variables, x and y

```
x <- 5
y <- 10
```

We want to change y, depending on the value of x.

We need to branch our *R* code using one or more conditional statements (`if`, `then`, `else`) and some boolean logic (a statement that can evaluate to `TRUE` or `FALSE`.

```
if (x == 5) {
    y <- 15
```

```
}
y
## [1] 15
```

We tested for one condition, `x==5`. If this condition evaluated to `TRUE`, the code within the block, `{ y <- 15 }` is executed. If it evaluates to `FALSE`, the code within the block is ignored. Note that the expression within the parenthesis `if()`, or `else()` should always evaluate to a single value of either `TRUE` or `FALSE` (not to `NA` or to multiple values).

Here is a more complex example, where we evaluate three cases. `x > 20` (x is larger than 20), `x >= 5 & x < 10` (x is in between 5 and 10, including 5, but not 10), and all other cases.

```
if (x > 20) {
    y <- y + 2
} else if (x > 5 & x < 10) {
    y <- y - 1
} else {
    y <- x
}

y
## [1] 5
```

If we have a boolean variable

```
b <- TRUE
```

You can do

```
if (b == TRUE) {
    print('hello')
}
## [1] "hello"
```

But it is more elegant to do

```
if (b) {
    print('hello')
}
## [1] "hello"
```

Now combining the previous chapter with this one, a for loop with an if/else branch:

```
a <- 1:5
f <- vector(length=length(a))
for (i in 1:length(a)) {
    if (a[i] > 2) {
        f[i] <- a[i] / 2
    } else {
        f[i] <- a[i] * 2
    }
}
f
## [1] 2.0 4.0 1.5 2.0 2.5
```

CHAPTER

ELEVEN

11. DATA PREPARATION

A very large part of research work can consist of data gathering, cleaning, combining and formatting. You may spend much more time on data preparation than on analysis, modeling or visualization. R provides some tools to make this process easier; but doing this well also requires that you carefully consider your options. See Wickham (2014) for a discussion. Two very important functions in this context are `reshape` and `merge`. Sorting of data can also be helpful.

11.1 reshape

`reshape` allows you to rearrange data from a *wide* to a *long* form and vice versa. This can be a very important step to simplify data analysis. In the *wide* form, each variable is a column and each case (e.g. individual) is a singe row. This is the common 'spreadsheet' approach. In the *long* form there is column that indicates the variable name and a column that has its value. Other columns identify the cases, and these can be repeated many times. The long form can be much easier for use in data analysis than the wide form. In many cases you may want to go from one form to the other and back.

The function to go between *wide* and *long* form is called `reshape`. Unfortunately `reshape` is rather difficult to use. There is only one function to go from wide to long and vice versa, creating confusion about what arguments to use when. It is also poorly documented, and its error messages are bewildering. (There is a package called `reshape` that you can use instead. But in this Introduction I want to stick with base R idiom.)

There is a good example of how to use reshape here. And I will re-use that and expand on it.

11.1.1 wide to long

Get some example data on student's performance in different subjects. Note how you can read a text on a web server (http address).

```
dd <- read.csv('http://www.ats.ucla.edu/stat/r/faq/hsb2.csv')
d <- dd[1:3, c(1:2, 8:9)]
## Error in `[.data.frame`(dd, 1:3, c(1:2, 8:9)): undefined columns selected
d
## Error in eval(expr, envir, enclos): object 'd' not found
```

This is a "wide" form dataset. To go to a long form, you need to provide an argument `varying` that indicates the variables that are combined into one new variable. `v.names` is the name of this new variable.

```
wvars <- c("write", "math")
x <- reshape(d, varying=wvars, v.names="score", direction = "long")
## Error in idvar %in% names(data): object 'd' not found
x
## Error in eval(expr, envir, enclos): object 'x' not found
```

x has two new variables "time", and "score". As you can see, "score" has the values for "write" and "math" in the d. The "time" variable identifies which is which. "write" is identified with a 1, and "math" is identified with a 2. To have clear labels rather than such numbers, you can use the times argument; and add the timevar argument to rename "time" to something more meaningful. Note that the argument names are based on the idea that the data refer to different measurements over time. But this is not always the case.

```
x <- reshape(d, varying=wvars, v.names="score", times=wvars, timevar = "subject", direction = "long")
## Error in idvar %in% names(data): object 'd' not found
x
## Error in eval(expr, envir, enclos): object 'x' not found
```

The row names also identify how the records were created. To get rid of the row names do

```
rownames(x) <- NULL
## Error in rownames(x) <- NULL: object 'x' not found
x
## Error in eval(expr, envir, enclos): object 'x' not found
```

Variables "id" and "female" were unchanging, but they were duplicated because two variables ("write" and "math") were combined into one ("subject")

11.1.2 long to wide

To go from long to wide we need to use "idvar" and "timevar". "idvar" identifies the variables that identify a single case (e.g. a single person, or other observational unit). In contrast, "timevar" indicates the variable that has the identifiers that become variables in the wide format. The remaining variable should have the values that match these new variables.

```
w <- reshape(x, idvar=c("id", "female"), timevar = "subject", direction = "wide")
## Error in reshape(x, idvar = c("id", "female"), timevar = "subject", direction = "wide"): object '
w
## Error in eval(expr, envir, enclos): object 'w' not found
```

Note that w is identical to d, except for the last two column names that now have "score." prepended to them. We can change that like this:

```
cn <- colnames(w)
## Error in is.data.frame(x): object 'w' not found
cn <- gsub("score.", "", cn)
## Error in gsub("score.", "", cn): object 'cn' not found
colnames(w) <- cn
## Error in eval(expr, envir, enclos): object 'cn' not found
w
## Error in eval(expr, envir, enclos): object 'w' not found
```

11.2 merge

A common situation is to have multiple data.frames with data for the same cases (e.g., individuals, fields, countries, ...). Such data.frames may need to be joined such that they can be analyzed. For example here we have a and b:

```
a <- dd[, 1:3]
## Error in `[.data.frame`(dd, , 1:3): undefined columns selected
# random sample of 100 records)
set.seed(1)
b <- dd[sample(nrow(dd), 100), c(1, 7:10)]
## Error in `[.data.frame`(dd, sample(nrow(dd), 100), c(1, 7:10)): undefined columns selected
```

```
```

a has 200 records. It has a unique identifier for each student and information about their sex (female or not) and race (4 groups).

```
dim(a)
## Error in eval(expr, envir, enclos): object 'a' not found
head(a)
## Error in head(a): object 'a' not found
table(a$female)
## Error in table(a$female): object 'a' not found
table(a$race)
## Error in table(a$race): object 'a' not found
```

b has the same unique identifier as a (but only for 100 students) and it has the grades for four subjects.

```
dim(b)
## Error in eval(expr, envir, enclos): object 'b' not found
head(b)
## Error in head(b): object 'b' not found
```

Imagine we were interested in differences in reading by sex or race. We would need to combine a and b. That, fortunately, is very simple, because we have the merge function.

```
ab <- merge(a, b, by='id')
## Error in merge(a, b, by = "id"): object 'a' not found
head(ab)
## Error in head(ab): object 'ab' not found
```

Always check the dimensions of the result

```
dim(ab)
## Error in eval(expr, envir, enclos): object 'ab' not found
```

In this case we expected 100 records (the lower number of the two; a had 200 records, but b only 100). Sometimes you get fewer than expected, suggesting that the identifiers do not match. In other cases you might want to keep *all* records, and create missing values where these are not available. You can do that like this:

```
ab <- merge(a, b, by='id', all.x=TRUE)
## Error in merge(a, b, by = "id", all.x = TRUE): object 'a' not found
dim(ab)
## Error in eval(expr, envir, enclos): object 'ab' not found
head(ab)
## Error in head(ab): object 'ab' not found
```

Note that the "x" in all.x refers to the first argument, hence a in this case. In other cases you might need to say all.y=TRUE or all=TRUE. Consider these extreme cases (with no matching records):

```
merge(a[1:3,], b[1:3, ], by='id')
## Error in merge(a[1:3, ], b[1:3, ], by = "id"): object 'a' not found
merge(a[1:3,], b[1:3, ], by='id', all.x=T)
## Error in merge(a[1:3, ], b[1:3, ], by = "id", all.x = T): object 'a' not found
merge(a[1:3,], b[1:3, ], by='id', all.y=T)
## Error in merge(a[1:3, ], b[1:3, ], by = "id", all.y = T): object 'a' not found
merge(a[1:3,], b[1:3, ], by='id', all=T)
## Error in merge(a[1:3, ], b[1:3, ], by = "id", all = T): object 'a' not found
```

Now that we have ab we can compute what we needed:

```
tapply(ab$read, ab$female, mean, na.rm=TRUE)
## Error in tapply(ab$read, ab$female, mean, na.rm = TRUE): object 'ab' not found
tapply(ab$read, ab$race, mean, na.rm=TRUE)
## Error in tapply(ab$read, ab$race, mean, na.rm = TRUE): object 'ab' not found
```

11.3 sort

It is often useful to sort data to make it easier to inspect it. R has a sort function but that is only for vectors. For matrices or data.frames you need to use the `order` function.

`sort` is straightforward:

```
x <- sample(10)
x
##  [1]  3  4  5  7  2  8  9  6 10  1
sort(x)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Now consider `order`:

```
i <- order(x)
i
##  [1] 10  5  1  2  3  8  4  6  7  9
x[i]
##  [1]  1  2  3  4  5  6  7  8  9 10
```

`order` returns a vector that allows you to sort. The first value of `i` is 7. This means that `x[7]` should be the lowest number in `x`. The next number is `x[6]` followed by `x[3]` and `x[2]` and so forth.

Consider data.frame `x`:

```
set.seed(0)
x <- a[sample(nrow(a), 10), ]
## Error in eval(expr, envir, enclos): object 'a' not found
x
##  [1]  3  4  5  7  2  8  9  6 10  1
```

Here is how you can use `order` to sort it by one column ("id" in this case):

```
oid <- order(x$id)
## Error in x$id: $ operator is invalid for atomic vectors
y <- x[oid, ]
## Error in eval(expr, envir, enclos): object 'oid' not found
y
## Error in eval(expr, envir, enclos): object 'y' not found
```

Or by multiple columns. In this case, we want to sort first by "race", then by "female" and then by "id":

```
oid <- order(x$race, x$female, x$id)
## Error in x$race: $ operator is invalid for atomic vectors
x[oid, ]
## Error in eval(expr, envir, enclos): object 'oid' not found
```

CHAPTER

TWELVE

12. GRAPHICS

With R you can make beautiful plots. You have a lot of control over what you want your plots to look like. But all the control is via code, and this does make things pretty complicated at times.

Moreover, there are entirely different approaches to make plots. Here we only discuss scatter-plots with the "base" package. The next chapter shows other basic plot types. The chapter thereafter shows how you can make plots with additional packages `levelplot` and `ggplot`. It is very useful to learn about "base" plotting first before you get into the more complicated (and sometimes, but not always more fancy) approaches. There are many websites with cool examples.

Here we use the `cars` data set that comes with R. It has two variables: the speed of cars and the distances taken to stop (data recorded in the 1920s), see `?cars`

12.1 Scatter plots

```
data(cars)
head(cars)
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

As we only have two variables, we can simply do

```
plot(cars)
```

But to be more explicit:

```
plot(cars[,1], cars[,2])
```

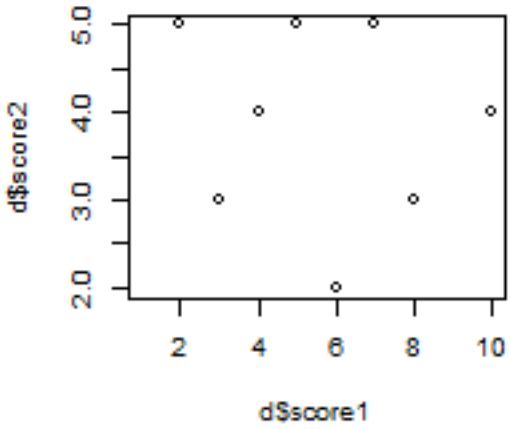
And now to embellish, add axes labels.

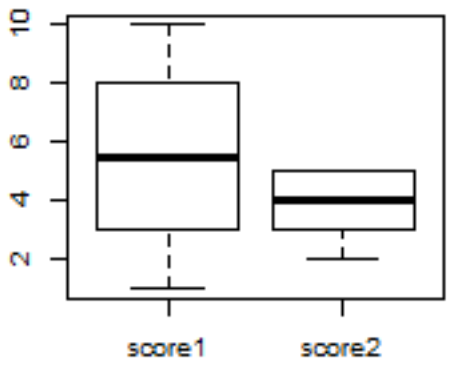
```
plot(cars[,1], cars[,2], xlab='Speed of car (miles/hr)', ylab='Stopping distance (feet)')
```

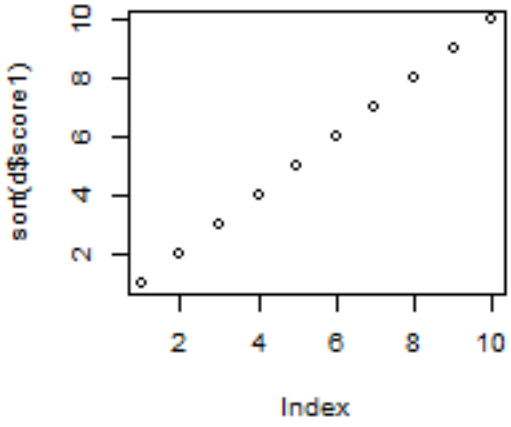
Different symbols (`pch` is the symbol type, `cex` is the size).

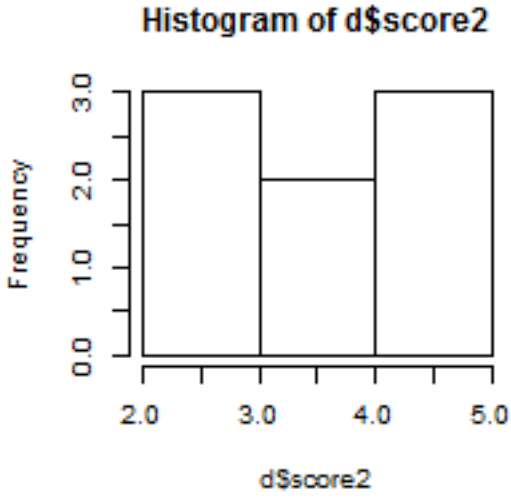
```
plot(cars, xlab='Speed of car (miles/hr)', ylab='Stopping distances (feet)', pch=20, cex=2, col='red
```

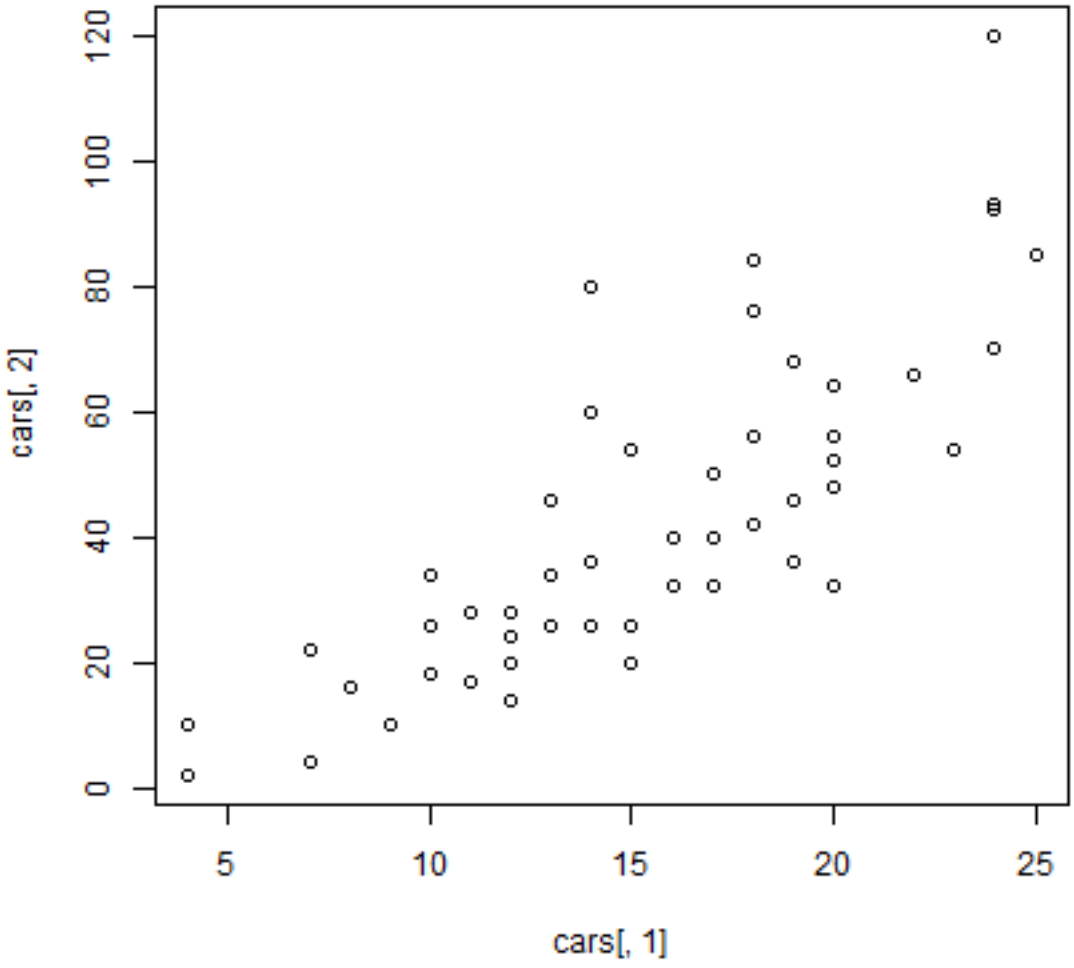
Let's change some things about the axes. Use `xlim` and `ylim` to set the start and end of an axis. `las=1` changes the orientation of the y-axis labels to horizontal.

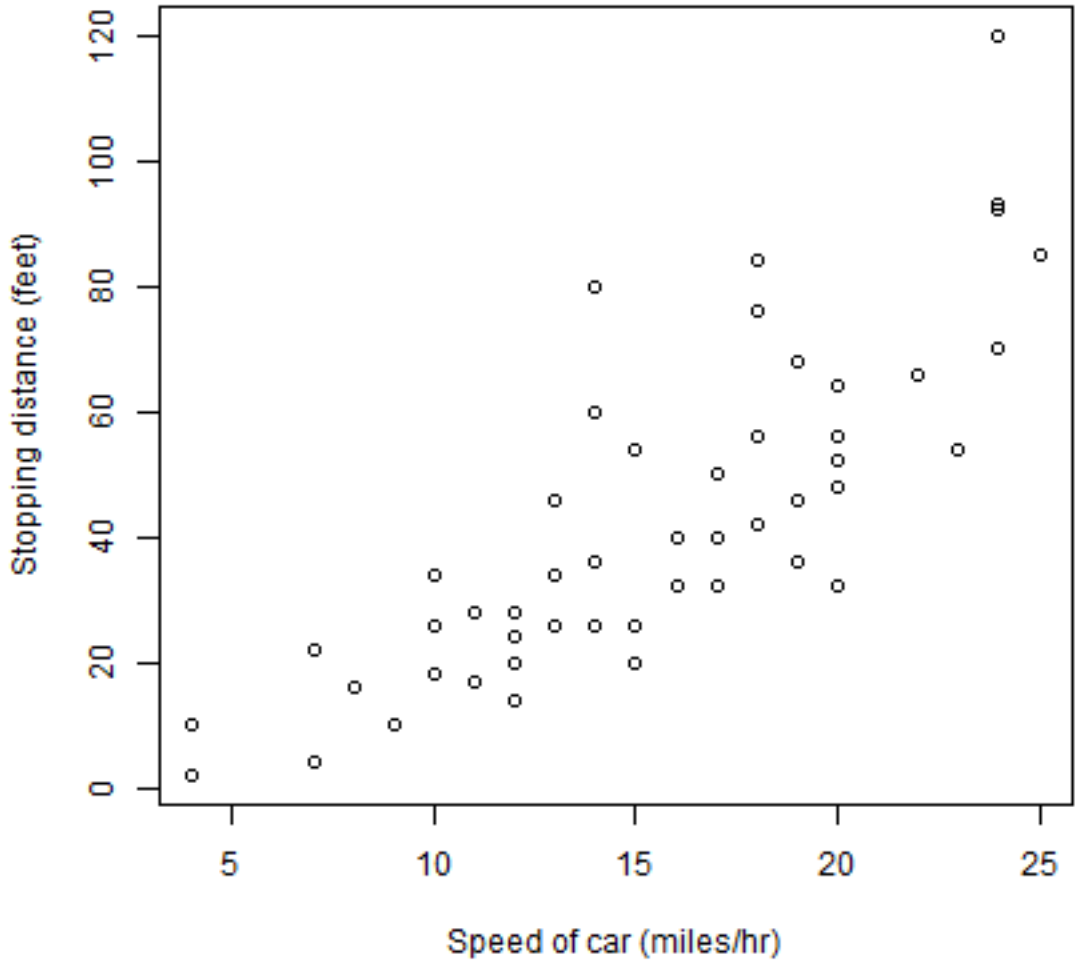

d$score2
2.0
3.0
4.0
5.0
2
4
6
8
10
d$score1

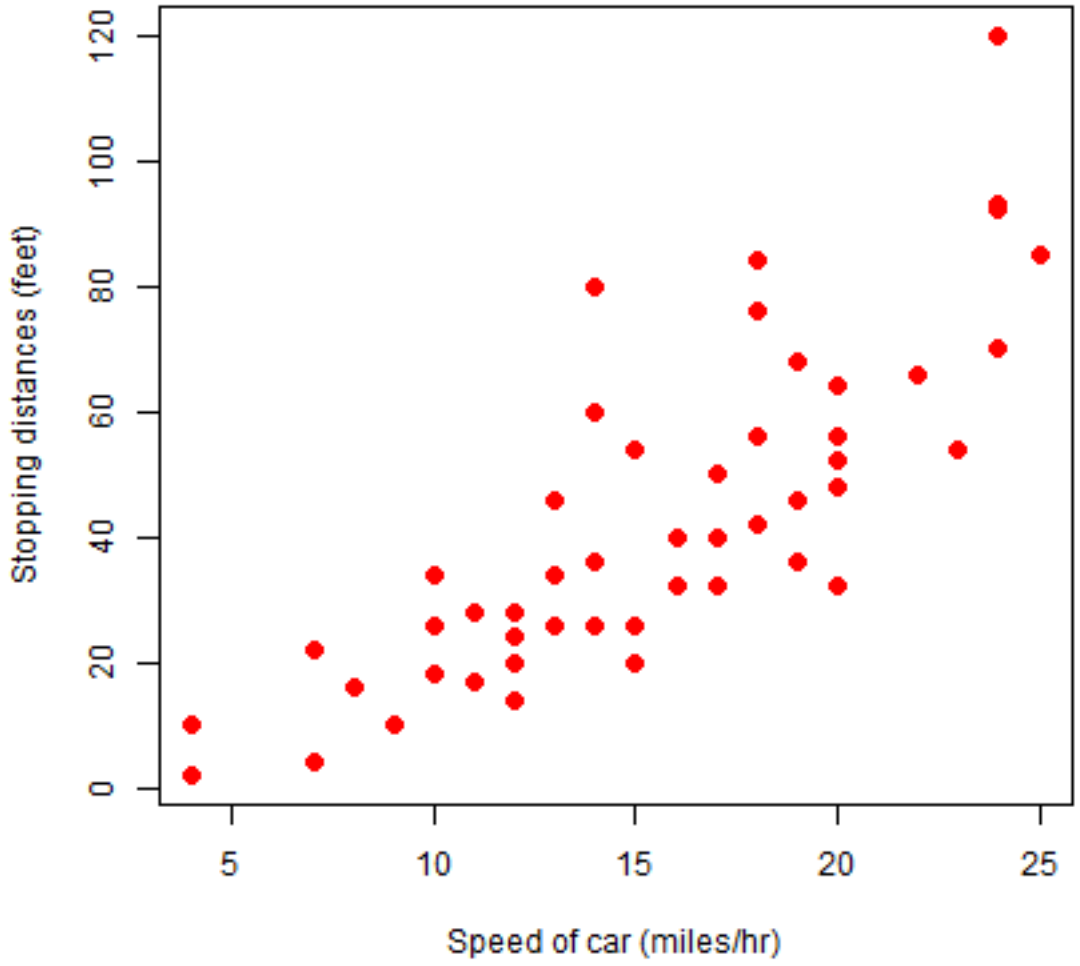

2
4
6
8
10
score1
score2


sort(d$score1)
2
4
6
8
10
Index

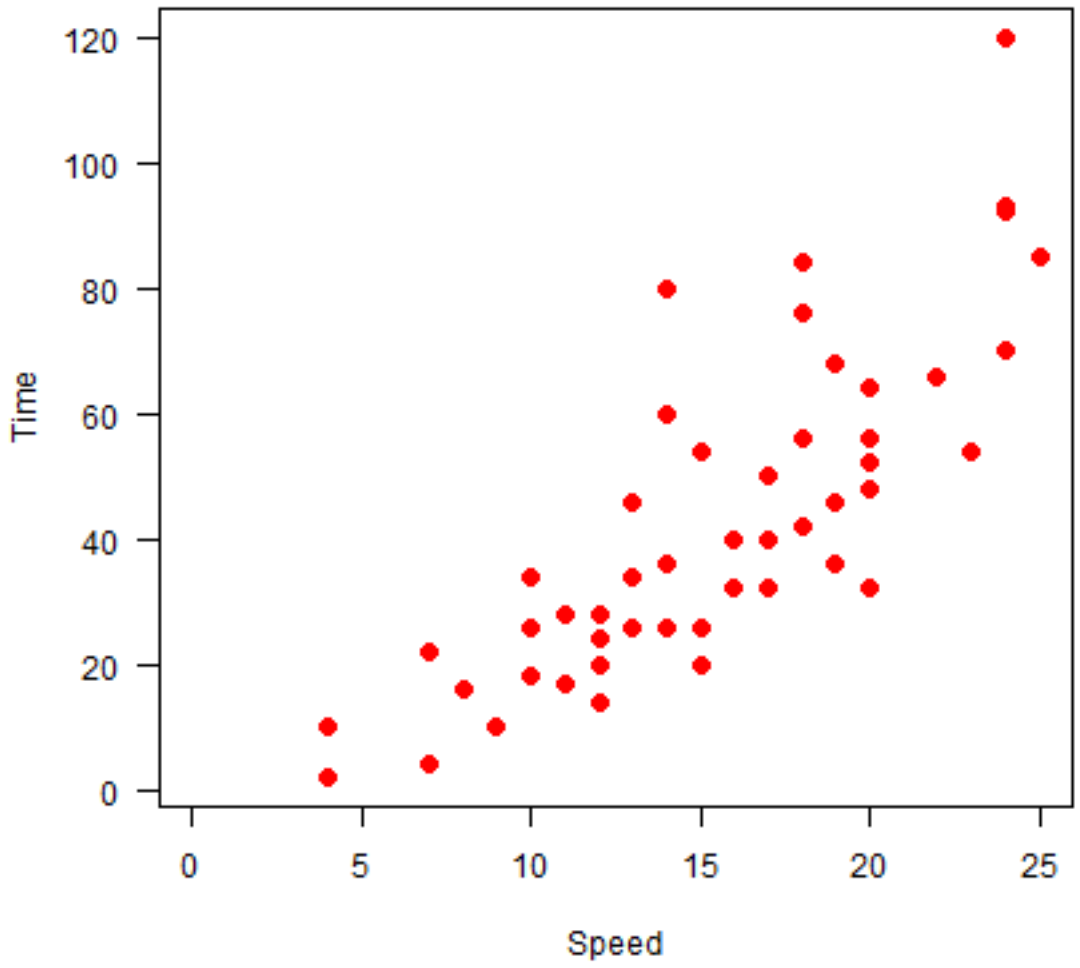

Histogram of d$score2
Frequency
0.0
1.0
2.0
3.0
2.0
3.0
4.0
5.0
d$score2


cars[, 2]
0
20
40
60
80
100
120
5
10
15
20
25
cars[, 1]


Stopping distance (feet)
0
20
40
60
80
100
120
5
10
15
20
25
Speed of car (miles/hr)


Stopping distances (feet)
0
20
40
60
80
100
120
5
10
15
20
25
Speed of car (miles/hr)

```
plot(cars, xlab='Speed', ylab='Time', pch=20, cex=2, col='red', xlim = c(0,25), las=1)
```

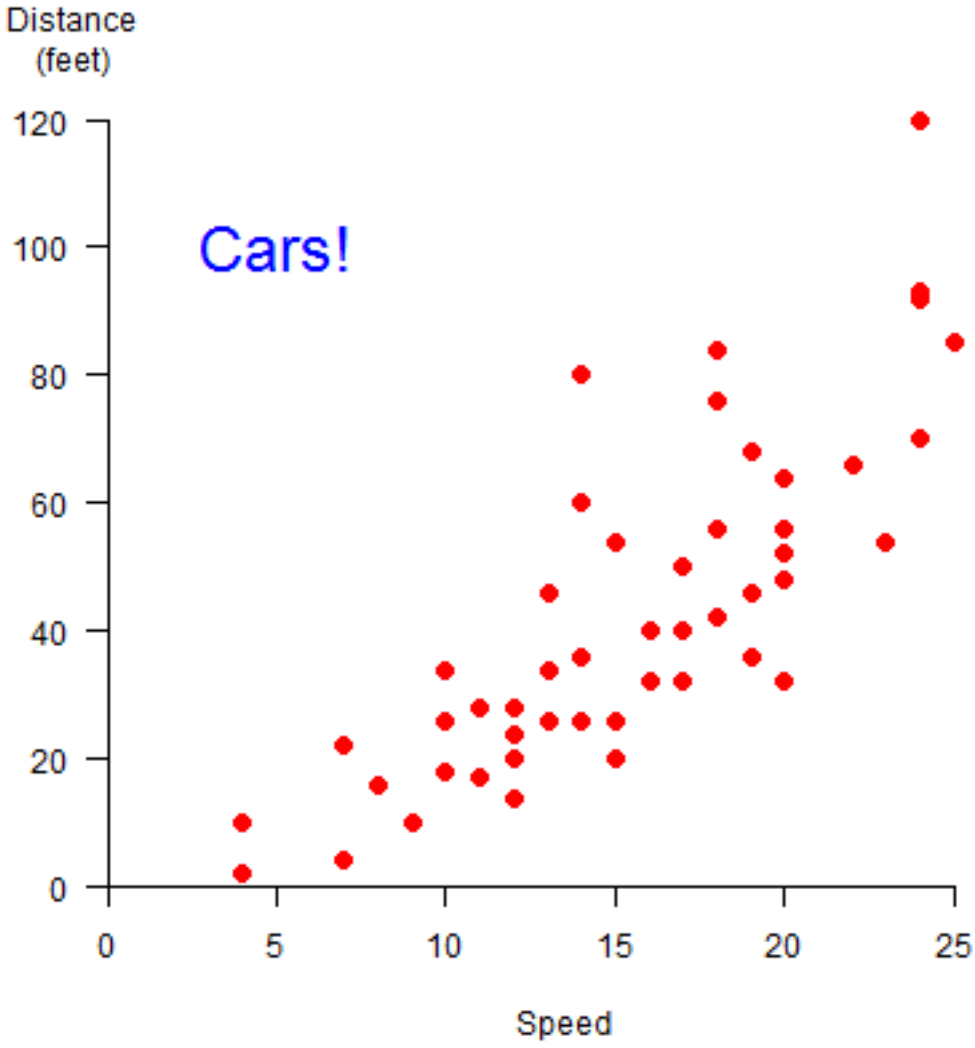


Here we do not draw axes at first, and then add the lower (1) and left (2) axis, to avoid drawing the clutter from the unnecessary "upper" and "right" axis. Arguments `xaxs="i"` and `yaxs="i"` force the axis to touch at `(0,0)`.

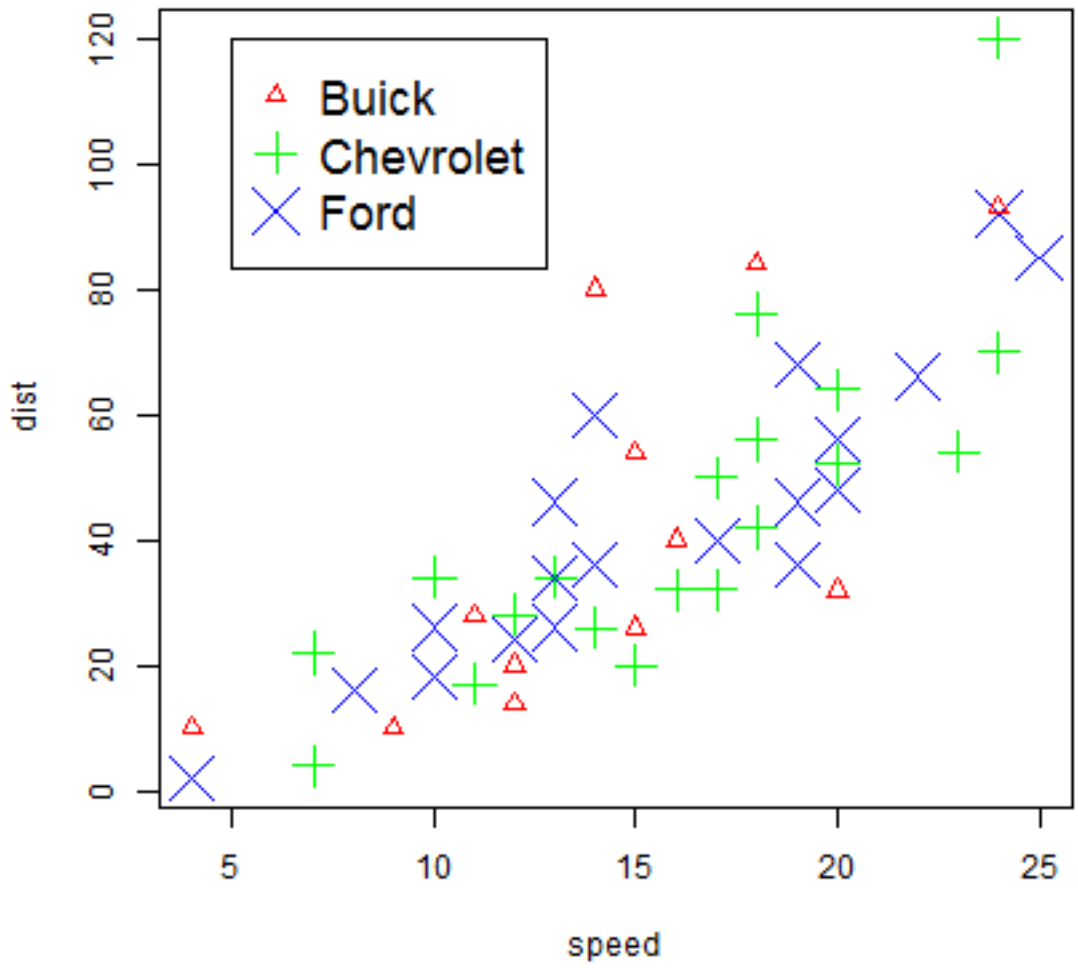
```
plot(cars, xlab='Speed', ylab='', pch=20, cex=2, col='red', xlim = c(0,27), ylim=c(0,125), axes=FALSE
axis(1)
axis(2, las=1)
text(5, 100, 'Cars!', cex=2, col='blue')
par(xpd=NA)
text(-1, 133, 'Distance\n(feet)')
```

We can change the symbols using another variable. Let's say we have three car brands and that we want to vary the symbol type, color, and size by brand (typically one of these changes should suffice to distinguish them!).

```
set.seed(0)
brands <- c('Buick', 'Chevrolet', 'Ford')
b <- sample(brands, nrow(cars), replace=TRUE)
```


Distance
(feet)
120
100
80
60
40
20
0
Cars!
0
5
10
15
20
25
Speed

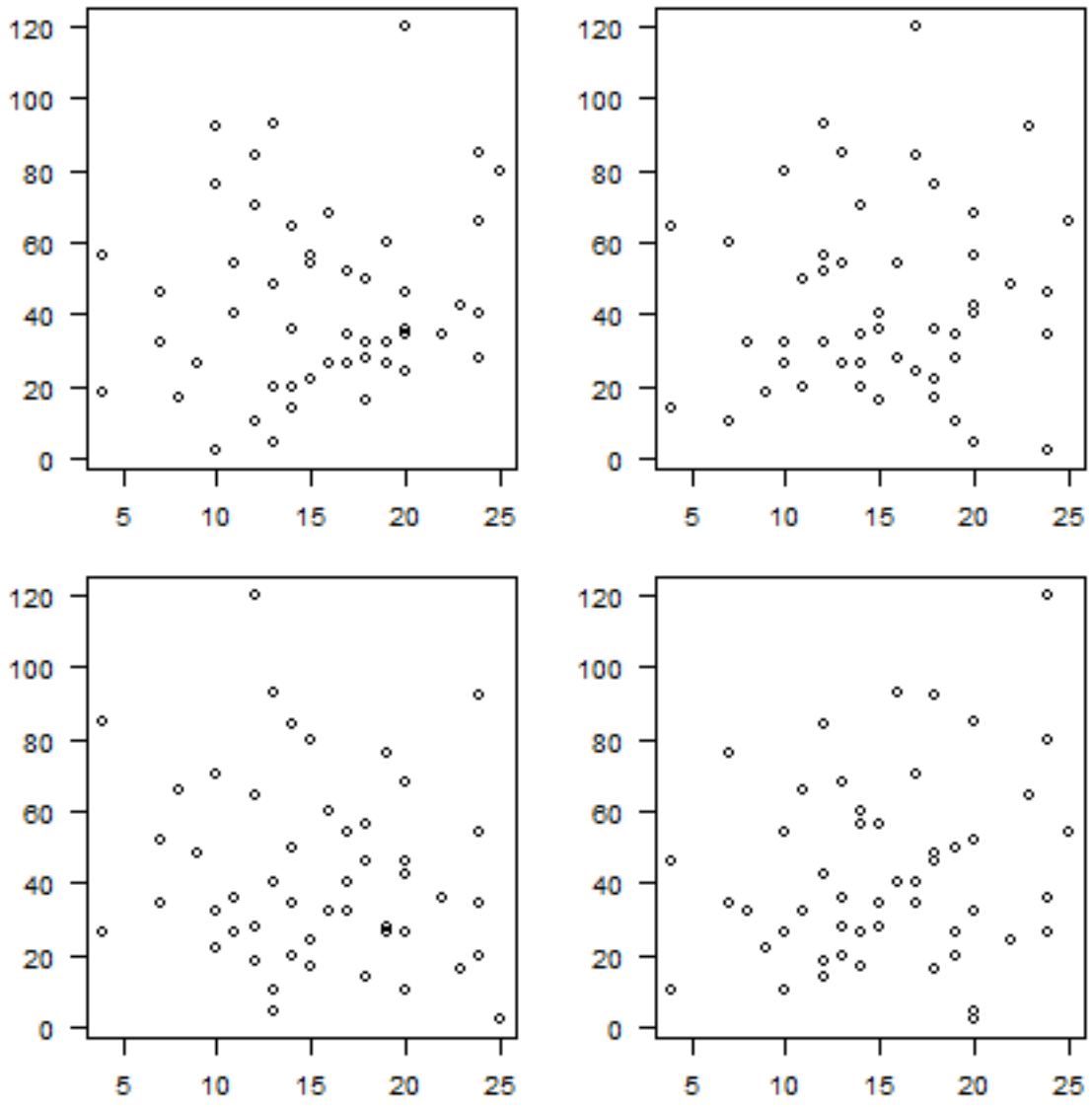
```
i <- match(b, brands)
plot(cars, pch=i+1, cex=i, col=rainbow(3)[i])
j <- 1:length(brands)
legend(5, 120, brands, pch=(j+1), pt.cex=j, col=rainbow(3), cex=1.5)
```



The important step is the use of `match`, that creates for each character string a matching number that can be used to set the character type desired.

As you have seen above, `plot` takes many variables. Several other parameters influencing your plot, can be set with `par`. See `?par` for details. Here I use it to create 4 subplots (`mfrow=c(2,2)` with non-default margins (`mar=c(2,3,1.5,1.5)`).

```
par(mfrow=c(2,2), mar=c(2,3,1.5,1.5))
for (i in 1:4) {
    plot(sample(cars[,1]), sample(cars[,2]), xlab='', ylab='', las=1)
}
```

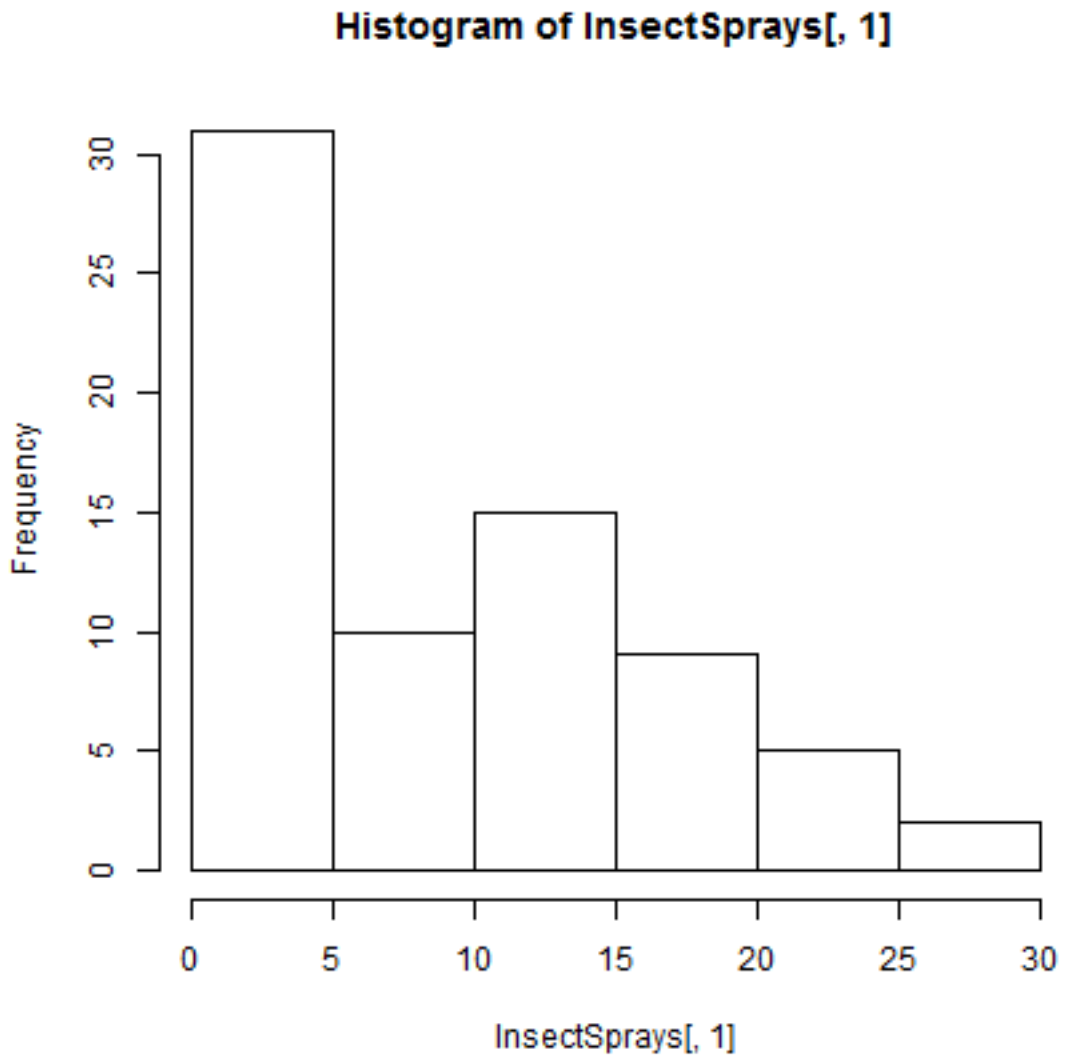

120
100
80
60
40
20
0
5
10
15
20
25

12.2 Some other base plots

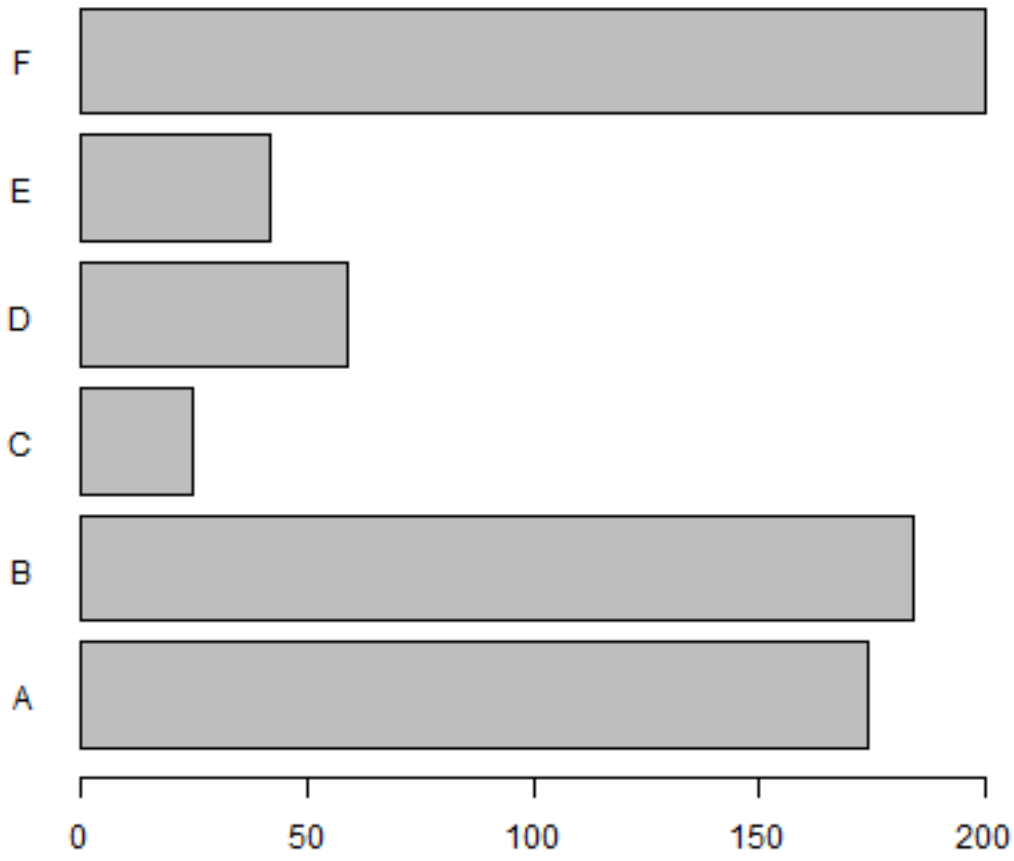
Consider the `InsectSprays` dataset

```
head(InsectSprays)
##   count spray
## 1    10     A
## 2     7     A
## 3    20     A
## 4    14     A
## 5    14     A
## 6    12     A
```

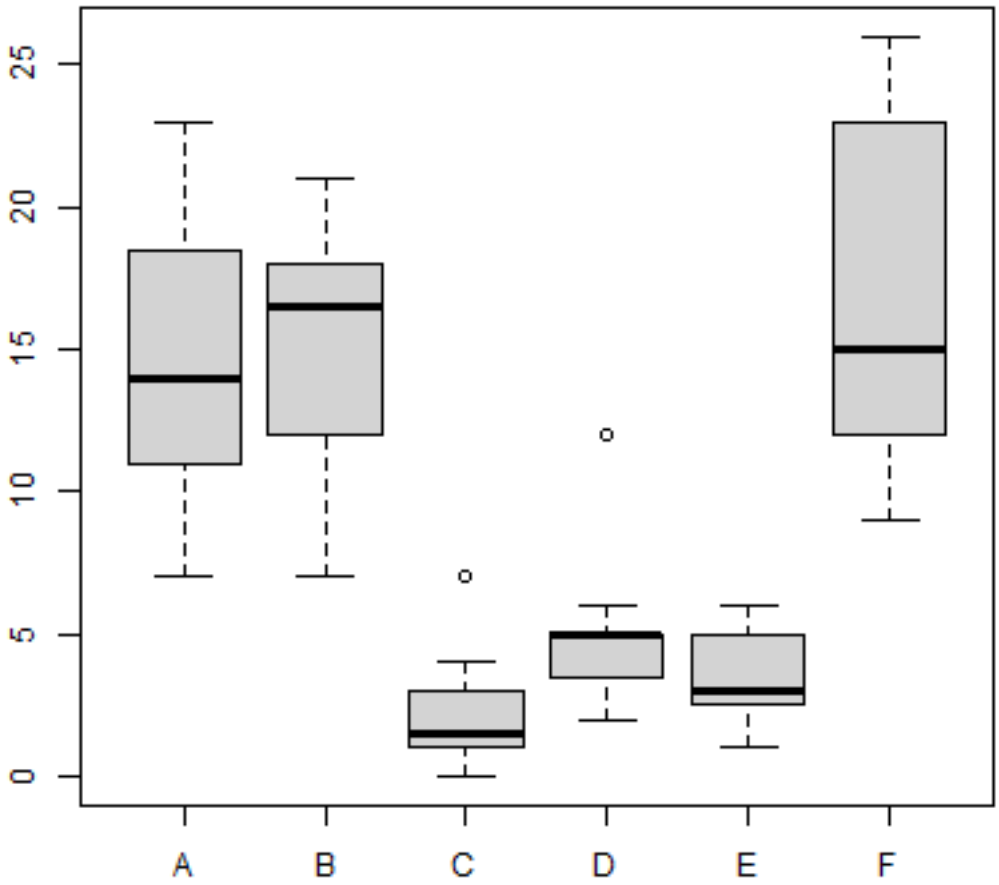
```
hist(InsectSprays[,1])
```




```
x <- aggregate(InsectSprays[,1,drop=F], InsectSprays[,2,drop=F], sum)
barplot(x[,2], names=x[,1], horiz=T, las=1)
```


F
E
D
C
B
A
0
50
100
150
200

```
boxplot(count ~ spray, data = InsectSprays, col = "lightgray")
```



CHAPTER

THIRTEEN

13. STATISTICAL MODELS

There are many type of statistical models. Here we show how to make simple regression models with *R*. Other modeling approaches tend to use similar syntax.

The most common way to specify a regression model in *R* is by creating a formula. For example `y ~ x` means `y` is a function of `x`. `y ~ a + b` means that `y` is a function of `a` and `b`.

Let's use the cars data that come with *R*. This dataset has measurements on the distance needed to stop given the speed a car was driven when the driver stepped on the breaks. We use the `lm` (linear model) function.

```
head(cars)
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
m <- lm(dist ~ speed, data=cars)
m
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Coefficients:
## (Intercept)        speed
##     -17.579        3.932
```

Note that the data is provided by data.frame `cars`, and that the names in formula are column names in this `data.frame`. The functions returned a model (`lm`) object. When printed it shows the coefficients of the regression model (`dist = -17.579 + 3.932 * speed`). `m` has quite a bit more information, but that is not shown, by default.

There are several functions that can be used to extract this information.

```
summary(m)
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
```

```
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
anova(m)
## Analysis of Variance Table
##
## Response: dist
##           Df Sum Sq Mean Sq F value   Pr(>F)
## speed      1  21186 21185.5  89.567 1.49e-12 ***
## Residuals 48  11354   236.5
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
residuals(m)[1:10]
##         1         2         3         4         5         6         7
##  3.849460 11.849460 -5.947766 12.052234  2.119825 -7.812584 -3.744993
##         8         9        10
##  4.255007 12.255007 -8.677401
```

You can use `abline` to draw a simple regression line like this.

```
plot(cars, col='blue', pch='*', cex=2)
abline(m, col='red', lwd=2)
```

More generally, you can use the `predict` function to use the model to predict values of `y` for any `x`.

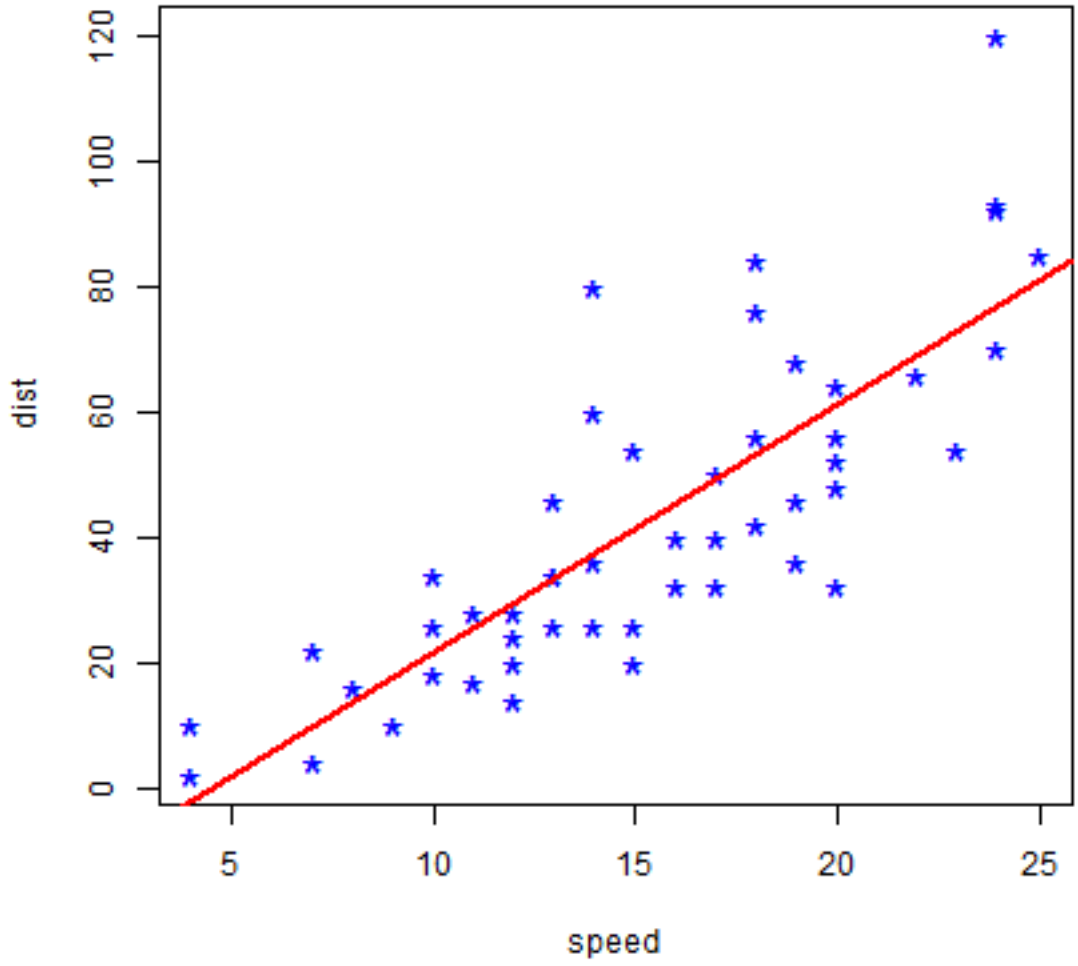
```
p <- predict(m, data.frame(speed=1:30))
p
##          1          2          3          4          5          6
## -13.646686  -9.714277  -5.781869  -1.849460   2.082949   6.015358
##          7          8          9         10         11         12
##   9.947766  13.880175  17.812584  21.744993  25.677401  29.609810
##         13         14         15         16         17         18
##  33.542219  37.474628  41.407036  45.339445  49.271854  53.204263
##         19         20         21         22         23         24
##  57.136672  61.069080  65.001489  68.933898  72.866307  76.798715
##         25         26         27         28         29         30
##  80.731124  84.663533  88.595942  92.528350  96.460759 100.393168
plot(1:30, p, xlab='speed', ylab='distance', type='l', lwd=2)
points(cars)
```

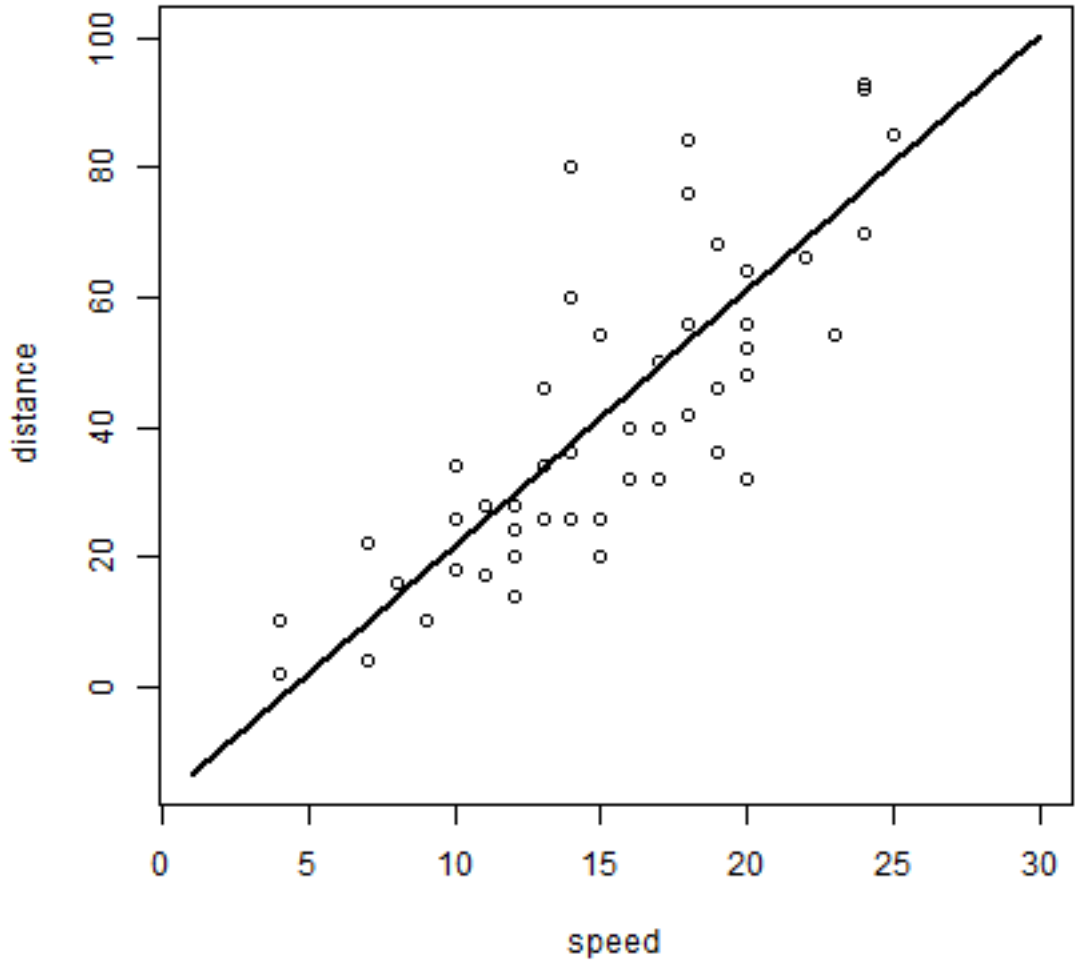
The `glm` (generalized linear models) function can do what `lm` can, but it is much more versatile. For example you can also use it for logistic regression. In logistic regression the response variable is normally binomial (0 or 1) or at least between 0 and 1. I create such a variable here (was the stopping distance above 40 or not?).

```
cars$above40 <- cars$dist > 40
```

Now we can use this variable in a `glm` model. By stating that `family='binomial'` we indicate that we want logistic regression. (The default is `family=gaussian` which indicates standard (normal) regression.

```
mlog <- glm(above40 ~ speed, data=cars, family='binomial')
mlog
##
## Call:  glm(formula = above40 ~ speed, family = "binomial", data = cars)
##
## Coefficients:
```


dist
0
20
40
60
80
100
120
5
10
15
20
25
speed

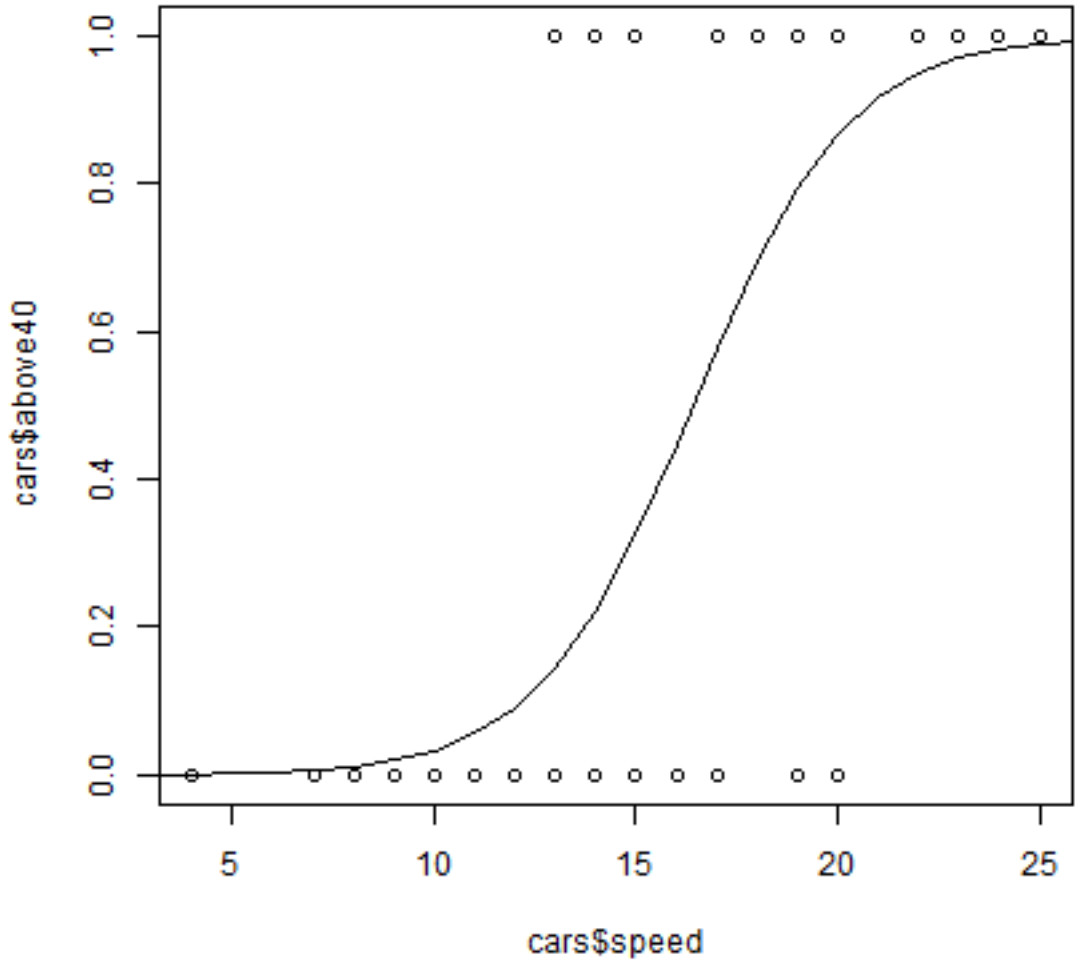

distance
0
20
40
60
80
100
0
5
10
15
20
25
30
speed

```
## (Intercept)        speed
##      -8.553        0.521
##
## Degrees of Freedom: 49 Total (i.e. Null);  48 Residual
## Null Deviance:       68.59
## Residual Deviance: 36.37     AIC: 40.37
```

Because a logistic model operates with logistically transformed numbers, we need to tell the predict function that we want the predicted values on the original scale (`type='response'`).

```
p <- predict(mlog, data.frame(speed=1:30), type='response')
```

```
plot(cars$speed, cars$above40)
lines(1:30, p)
```



For a more in-depth discussion of regression models with R go here.

CHAPTER

FOURTEEN

14. MISCELLANEOUS

14.1 Packages

Packages are collections of *R* functions. Typically around a related set of tasks. *R* comes with a standard "base" set of packages. Others are available for download and installation. These packages are developed by (groups of) individuals independently of the "core R" development team. Most of these packages are developed by volunteers, who write them to support their research or other work. For that reason the are highly variable in design and quality. There is also a lot of overlap between packages and it can take a while to find the ones you need to best accomplish a task.

To install a package do `install.packages("packagename")`

where "packagename" is replaced with the name of an actual package. For example `install.packages("raster")`

Or install multiple packages at the same time `install.packages(c("randomForest", "raster", "gstat"))`

The directory where packages are stored is called the library. Once installed, they have to be loaded into the session to be used. For example:

```
library(raster)
```

So you install a package only once (for each R version), or once in a while (to get updates), but you load a package every time you start a new R session (script) that needs it.

It is very important to stay up to date with R and the packages, as they improve every day.... You should update the main *R* program every 6 months and update your packages more regularly, perhaps once a month. To update all your packages you can run `update.packages()`

14.2 How to write a good script?

Read and follow this style guide.

Only use a path name at the very top of your script. After that, all path names should be relative.

Do not copy and paste the same code (and make minor changes). Rather, write functions to put code together. Perhaps store these in a separate .R file and access them via `source('myfunctions.R)`

14.3 Help!

How to get help when you are stuck? How to find and fix errors? That is the hardest part for beginners. You can start by checking the list of frequently used functions. And, there is always Google... Any question you may have as a beginner has been asked and answered before. Often on Stackoverlfow.

But at first it is hard to find the right search terms, and to distinguish between good answers, and "solutions" that just pull you down further into the hole you are in.

When asking a question about how to do something in *R*, it is very important to simplify it as much as possible, and focus on the nucleus of the problem only. That is, do not show a long script where all kinds of things happen that are OK. Show a short script that gets you up to the point where you are stuck. Such as script should be reproducible and self-contained.

Reproducible means that anyone can run it in *R* and get the same results. So you need to include `set.seed`. Self-contained means that it should not point to files you only have. You could make these available, but why complicate things? Just create some data with code or use a data set that comes with *R* (e.g. "cars", "iris", but there are many). See the examples in the *R* help files for 1000s of ideas of how you can do this.

For more discussion see this posting guide and this discussion