

Practicals Part 1

Mandy Vogel

May 25, 2015

1 Basic Practice with R

1.1 Download and Installation

1. Download R from CRAN (Comprehensive R Archive Network) and install it in your computer. (if you haven't done it yet)
2. Download and install RStudio from the RStudio homepage

1.2 First Commands

1. Create a working directory "MyRcourse".
2. Run R and change your working directory to "MyRcourse".
3. Try to use R as a calculator.
4. What is greater e^π or π^e .
5. Try the graphical demo using `demo(graphics)` at the command line.
6. Look at the objects created by this demo by the function `ls()`. Please do NOT omit the parenthesis!
7. Delete same objects with the function `rm`.
8. Delete all objects with `rm(list=ls())`

1.3 Help

The structure of the html and the text based help differs in some minor aspects. But the main structure is the same. Type

```
> ?lm
```

Around the top you find the name of the function and its package. It is not that important if you use your local help, but the help pages are also available at the web - so you know which package you have to install.

In the text version follows a line with keywords

The first little paragraph opens the help page with a general description.

The paragraph *usage* shows you the main arguments of the function together with their defaults.

It follows a detailed list with the arguments and a description, so you can find possible values and maybe useful options.

The *details* paragraph is full of details ... you won't need it when you are looking for quick help.

The *value* paragraph is very valuable. There you come to know what information your result holds. In the most cases you see just a small part of your result if you just type the command. To have access to everything you have to assign the result like

```
> my.model <- lm(a~b, data=c)
```

The structure of this result is described in *details*

The next paragraphs hold diverse information e.g. about sources, author, related functions and topics.

At the end you find the *most important* part of the help for beginning - the examples. You can run these examples and learn how the function works...

1. go to the help page of the `lm()` command, what's the function for?
2. run the example line by line, try to understand what's going on (not every single detail)
3. add the following lines to visualize the example:

```
> plot(weight ~ group, col="lavender")
```

4. now run `example(lm)`.

1.4 Help

Sometimes you cannot remember the precise name of the function, but you know the subject on which you want help (e.g. data input in this case). Use the `help.search` function (without a question mark) with your query in double quotes like this:

```
> help.search("data input")
```

and (with any luck) you will see the names of the R functions associated with this query. Afterwards you can get detailed information by typing `?function`

`find()` tells you what package something is in:

```
> find("lm")
[1] "package:stats"
```

and `apropos()` returns a character vector giving the names of all objects in the search list that match your (potentially partial) enquiry:

```
> apropos("lm")
[1] "anova.glm"           "anova.glm.list"      "anova.list.lm"
[4] "anova.lm"            "anova.lm.list"       "anova.mlm"
[7] ".__C__anova.glm"     ".__C__anova.glm.null" ".__C__glm"
.....
```

Here you find a (of course incomplete) list of books (<http://www.r-project.org/doc/bib/R-jabref.html>).

1.5 Online Help

Online Help There is a tremendous amount of information about R on the web, but your first port of call is likely to be CRAN at <http://cran.r-project.org/> (click on *manuals* on the left)

Here you will find a variety of R manuals:

- *An Introduction to R* gives an introduction to the language and how to use R for doing statistical analysis and graphics.
- A draft of the *R Language Definition* documents the language per se that is, the objects that it works on, and the details of the expression evaluation process, which are useful to know when programming R functions.

- *Writing R Extensions* covers how to create your own packages, write R help files, and use the foreign language (C, C ++, Fortran) interfaces.
- *R Data Import/Export* describes the import and export facilities available either in R itself or via packages which are available from CRAN.
- *R Installation and Administration*, which is self-explanatory.
- *R: A Language and Environment for Statistical Computing* (referred to on the website as The R Reference Index) contains all the help files of the R standard and recommended packages in printable form.

1. go to rseek.org and "google" *cran task views*.
2. have a look on these task views (here is the direct link: <http://cran.r-project.org/web/views/>)
3. click on something you are interested in: you will find a lot of information and related packages

1.6 Look at This!

Just to get a feeling what you can do with R – and as a little break – run

```
> demo(persp)
> demo(graphics)
> demo(Hershey)
> demo(plotmath)
```

1.7 Sequences

1. Create a vector `w` with components 1, -1, 2, -2
2. Display this vector
3. Obtain a description of `w` using `str()`
4. Create the vector `w+1`, and display it.
5. Create the vector `v` with components (0, 1, 5, 10, 15, ... , 75) using `c()` and/or `seq()`.
6. Find the length of this vector.

1.8 Displaying and changing parts of a vector (indexing)

First try to understand the following commands:

```
> x <- c(2, 7, 0, 9, 10, 23, 11, 4, 7, 8, 6, 0)
> x[4]
> x[3:5]
> x[c(1, 5, 8)]
> x[x > 10]
> x[(1:6) * 2]
> x[x == 0] <- 1
> x
> ifelse(round(x/2) == x/2, "even", "odd")
```

Now try the following:

1. Display every third element in x
2. Display elements that are less than 10, but greater than 4
3. Modify the vector x, replacing by 10 all values that are greater than 10
4. Modify the vector x, multiplying by 2 all elements that are smaller than 5
5. Create a new vector y with elements 0,1,0,1, . . . (12 elements) and a vector z that equals x when y=0 and 3x when y=1. (You can do it using ifelse, but there are other possibilities)

Now try the following command lines:

```
> n <- 10
> g <- gl(n, 100, n * 100)
> x <- rnorm(n * 100) + sqrt(as.numeric(g))
> boxplot(split(x, g), col = "lavender", notch = TRUE)
```

1. Explain what you are doing in each line.
2. Save your workspace with the name "Practical01.RData" and quit R with the function q().
3. Restart R by clicking on your workspace (Windows).
4. Restart R and load your workspace (Linux) (use load()).

1.9 A bit of matrices

1. Generate a matrix from a vector with values from 1 to 12 (it was an example in the slides). Use the `matrix()` command together with the `ncol` argument
2. Plotting matrices: try the function `matplot()` with the matrix that you have generated.
3. Try again `matplot()` with options : `type='b'` and `type='l'`. What is the difference ?
4. Generate a vector `x1` with numbers from 10 to 100 every 10.
5. Generate a vector `x2` with the first 10 letters of the alphabet (use capital letters).
6. Concatenate vectors `x1` and `x2`. Explain what was happened?

1.10 Another Boxplot

Generate a data frame `test` as follows:

```
> x <- gl(5, 20)
> y <- rt(100, df=5)
> test <- data.frame(x,y)
```

Use `summary()` to summarise this data frame. Make a boxplot of `y` dependend on `x`.

1.11 Indexing in data frames

Start with creating a simple data frame:

```
> mydata <- data.frame(name = c("Joe", "Ann", "Jack", "Tom"), age = c(34,
+ 50, 27, 42), sex = c(1, 2, 1, 1), height = c(185, 170, 175,
+ 182))
```

See what happens (and why):

```
> mydata
> mydata[[2]]
> names(mydata)
> mydata[, "age"]
> mydata$age
```

```
> mydata[2, 3]
> mydata[, 2]
> mydata[1, ]
```

Often it is useful to have row names defined as unique subject indicators (or names):

```
> rownames(mydata) <- mydata$name
> mydata["Tom", ]
```

Note that only values that are unique for each individual can be row (or column) names. Now let's create another data frame with more individuals than the in first one:

```
> weights <- data.frame(weight = c(67, 81, 56, 90, 72, 79, 69))
> rownames(weights) <- c("Ann", "Peter", "Sue", "Jack", "Tom", "Joe", "Jane")
> weights[substr(rownames(weights), 1, 1) == "J", ]
```

How to add weights to individuals in mydata, using the new data frame?

```
> mydata$weight <- weights[rownames(mydata), "weight"]
```

(There is also the function `merge()` to join datasets see `help(merge)`).

Using the same idea, add the variable height to the dataset weights. See what happens with the individuals who are not in mydata.

1.12 Logical Arithmetic - Logical Indexing

Suppose that `x` is a sequence from 0 to 6 like this:

```
> x<-0:6
```

Now we can ask questions about the contents of the vector called `x`. Is `x` less than 4?

```
> x < 4
[1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE
```

Two important logical functions are `all()` and `any()`. They check an entire vector but return a single logical value

```
> all(x > 0)
[1] FALSE
> any(x == 0)
[1] TRUE
```

1. Try `sum(x>4)`! Try to explain the result!
2. Define a logical vector `x` containing three elements `NA`, `TRUE`, `FALSE`. Type now: `outer(x,x,"&")` and `outer(x,x,"|")`. Explain the result! Note the behaviour of `NA | NA` and `NA | FALSE`.
3. Use one of the help functions to find the difference between `&` and `&&`
4. What happens if the subscript vector contains `NAs`?

1.13 Working with real data frames

First remove all objects from the workspace by typing: `rm(list=ls())` We shall use the `pima` data which concerns 768 adult female Pima Indians living near Phoenix. These data are available in the `faraway` package (if it is not installed yet install it via `install.packages()`):

```
> library(faraway)
> data(pima)
> objects()
```

The function `objects()` is a alias for `ls()` shows what is in your workspace. To find out a bit more about `pima` try

```
> help(pima)
> names(pima)
> head(pima)
```

The dataframe `parstum` in the `faraway` package contains data from a study about marijuana and parent alcohol and drug use.

1. Load these data with

```
> data(parstum)
```

and print some of the content on the screen.

2. Check that you now have two objects, `pima`, and `parstum` in your work space.
3. Obtain a description of the object `parstum`.
4. Remove the object `parstum` with the command

```
> rm(parstum)
```

Check that the object `parstum` is not any more there.

1.14 Referencing parts of the data frame (indexing)

Typing `pima` will list the entire data frame - not usually very helpful. Now try

```
> pima[1, "glucose"]
```

This will list the value taken by the first subject for the `bweight` variable. Alternatively

```
> pima[1, 2]
```

will list the value taken by the first subject for the second variable (which is `bweight`). Similarly

```
> pima[2, "diastolic"]
```

will list the value taken by the second subject for `bweight`, and so on. To list the data for the first 10 subject for the `bweight` variable, try

```
> pima[1:10, "bmi"]
```

and to list all the data for this variable, try

```
> pima[, "bmi"]
```

To list the data for the first subject on all variable except the second try

```
> pima[1, -2]
```

Now

1. Display the data on the variable `age` for row 7 in the `pima` data frame.
2. Display all the data in row 7.
3. Display the first 10 rows of the data on the variable `triceps`.

1.15 Summaries

A good way to start an analysis is to ask for a summary of the data by typing

```
> summary(pima)
```

To see the names of the variables in the data frame try

```
> names(pima)
```

Variables in a data frame can be referred to by name, but to do so it is necessary also to specify the name of the data frame. Thus `pima$pregnant` refers to the variable `pregnant` in the `pima` data frame, and typing `pima$pregnant` will print the data on this variable. To summarize the variable `pregnant` try

```
> summary(pima$pregnant)
```

Alternatively you can use

```
> with(pima, summary(pregnant))
```

In most datasets there will be some missing values. R then codes the missing values using the `NA` (not available) symbol. The summary shows the number of missing values for each variable.

1.16 Turning a variable into a factor

In R categorical variables are known as factors, and the different categories are called the levels of the factor. Variables such as `test` originally coded using integer codes, and by default R will interpret these codes as numeric values taken by the variables. For R to recognize that the codes refer to categories it is necessary to convert the variables to be factors, and to label the levels. To convert the variable `test` to be a factor, try

```
> pima$test <- factor(pima$test)
> str(pima)
```

Alternatively you can use

```
> pima <- transform(pima, test = factor(test))
> str(pima)
```

Note that either way `test` is now a factor with two levels, labelled "0" and "1" which are the original values taken by the variable. It is possible to change the labels to (say) "negative" and "positive" with

```
> pima$test <- factor(pima$test, labels = c("negative", "positive"))
> str(pima)
```

or

```
> pima <- transform(pima, test = factor(test, labels = c("negative",  
+ "positive")))  
> str(pima)
```

1.17 Frequency tables

When starting to look at any new data frame the first step is to check that the values of the variables make sense and correspond to the codes defined in the coding schedule. For categorical variables (factors) this can be done by looking at one-way frequency tables and checking that only the specified codes (levels) occur. The most useful function for making simple frequency tables is `table()`. The distribution of the factor `test` can be viewed using

```
> with(pima, table(test))
```

or by specifying the data frame as in

```
> table(pima$test)
```

For simple expressions the choice is a matter of taste, but `with()` is preferable for more complicated expressions.

1. Find the frequency distribution of `pregnant`
2. Find the two-way frequency distribution of `test` and `pregnant`.

1.18 Grouping the values of a numeric variable

For a numeric variable like `age` it is often useful to group the values and to create a new factor which codes the groups.

For example we might cut the values taken by `age` into the groups 20–29, 30–39, 40–49, 50–59, 60–69, 70–100 and then create a factor called `agegrp` with 6 levels corresponding to the four groups.

The best way of doing this is with the function `cut()`. Try

```
> pima <- transform(pima, agegrp = cut(age, breaks = c(20,  
+ 30, 40, 50, 60, 70, 100), right = FALSE))  
> with(pima, table(agegrp))
```

By default the factor levels are labelled [20-30), [30-40), etc., where [20-30) refers to the interval which includes the left hand end (20) but not the right hand end (30). This is the reason for `right=FALSE`. When `right=TRUE` (which is the default) the intervals include the right hand end but not the left hand.

Observations which are not inside the range specified in the `breaks` part of the command result in missing values for the new factor. You can specify that you want to cut a variable into a given number of intervals of equal length by specifying the number of intervals. For example

```
> pima <- transform(pima, agegrp = cut(age, breaks = 5,  
+   right = FALSE))  
> with(pima, table(agegrp))
```

shows 5 intervals of width 12.

1. Summarize the numeric variable `bmi`, look at the range of values.
2. note that a `bmi` of value 0 does not make any sense, originally missing data are coded as zeros; recode these missing values by

```
pima$bmi[pima$bmi==0] <- NA
```

3. Create a new factor `pima$bmi2` which cuts `bmi` at 20, 24, 30, 33 including the left hand end, but not the right hand. Make a table of the frequencies for the five levels of `bmi2` (take care not to produce new NAs).
4. Create a new factor `bmi3` which cuts `bmi` into 5 equal intervals, and make a table of frequencies.

1.19 Generating new variables

New variables can be produced using assignment together with the usual mathematical operations and functions. For example

```
> loggl <- log(pima$glucose)
```

produces the variable `loggl` in your work space (Global environment), while

```
> pima$loggl <- log(pima$glucose)
```

produces the variable `loggl` in the `pima` data frame. Logs base 10 are obtained with `log10()`. Logical variables take the values `TRUE` or `FALSE`, and behave like factors. New variables can be created which are logical functions of existing variables. For example

```
> pima$obese <- pima$bmi > 30
> str(pima)
```

creates a logical variable `obese` (in `pima` with levels `TRUE` and `FALSE`, according to whether `bmi` is greater than 30 or not. One common use of logical variables is to restrict a command to a subset of the data. For example, to create a new dataframe restricted to women with a very height BMI, try

```
> pima.obese <- subset(pima, bmi > 30)
> summary(pima.obese)
```

1. Add a column with an id number to `pima`, use `pima$id <- rownames(pima)`
2. Create a logical variable called `high.bp` according to whether `diastolic` is more than 80 or not. Make a frequency table of `high.bp`.
3. Display the id numbers of women with `diastolic > 80`.

1.20 Using a text editor with R

When working with R it is best to use a text editor to prepare a batch file (or script) which contains R commands and then to run them from the script. You can use the built-in script editor for this, or R Studio, or you can use your favourite text editor instead if you prefer.

One major advantage of running all your R commands from a script is that you end up with a record of exactly what you did which can be repeated at any time. This will also help you redo the analysis in the (highly likely) event that your data changes before you have finished all analyses.

1.21 Working with R

When starting R it is always a good idea to use `getwd()` to print the working directory. You may not be where you think you are! The command `dir()` can be used to see what files you have in the working directory.

When exiting from R you are offered the chance of saving all the objects in your current work space. This is not recommended as the work space can fill up with temporary objects, and it is easy to forget what these are when you resume the session. It is better to build up a script file as you work, and to run this at the start of a new session.

You can save any R object to disc. For example, to save the data frame `pima` try

```
> save(pima, file = "pima2.rdata")
```

which will save the `pima` data frame in the file *pima2.rdata*. By default the data frame is saved as a binary file, but the option `ascii=TRUE` can be used to save it as a text file. To load the object from the file use

```
> load("pima2.rdata")
```

The commands `save()` and `load()` can be used with any R objects, but they are particularly useful when dealing with large data frames.

1.22 The search path

R organizes objects in different positions on a search path. The command

```
> search()
```

shows these positions. The first is the work space, or global environment, the second is the `faraway` package, the third is a package of commands called `stats`, the fourth is a package called `graphics`, and so on (order can vary). To see what is in the work space try

```
> objects() ## or ls()
```

You should see just the object `pima`. The command `ls()` does the same as `objects()`. To see what is in the `faraway` package, try

```
> objects(2) ## 2 is the position in the search path
```

There are more than 100 functions/objects in this package.

When you type the name of an object R looks for it in the order of the search path and will return the first object with this name that it finds. This is why it is best to start your session with a clean workspace, otherwise you might have an object in your workspace that masks another one later in the search path.

1.23 Attaching a Data Frame

I strongly recommend to **NOT** to use the `attach` function. If you carefully read to following sentences you understand why.

The function `objects()` shows that the data frame `pima` is in your workspace. To refer to variables in `pima` by name it is necessary to specify the name of the data frame, as in `pima$bmi`.

This is quite cumbersome, and provided you are working primarily with one data frame, it can help to put a copy of the variables from a data frame in their own position on the search path.

This is done with the function

```
> attach(pima)
```

which places a copy of the variables in the `pima` data frame in position 2. You can verify this with

```
> objects(2)
```

which shows the objects in this position are the variables from the `pima` data frame.

Note that the `faraway` package has now been moved up to position 3, as shown by the `search()` function. When you type the command:

```
> bmi
```

R will look in the first position where it fails to find `bmi`, then the second position where it finds `bmi`, which now gets printed.

Although convenient, attaching a data frame can give rise to confusion. For example, when you create a new object from the variables in an attached data frame, as in

```
> subgrp <- age[pregnant == 1]
```

the object `subgrp` will be in your workspace (position 1 on the search path) not in position 2.

To demonstrate this, try

```
> objects(1)
```

```
> objects(2)
```

Similarly, if you modify the data frame in the workspace the changes will not carry through to the attached version of the data frame. The best advice is to regard any operation on an attached data frame as temporary, intended only to produce output such as summaries and tabulations.

Beware of attaching a data frame more than once - the second attached copy will be attached in position 2 of the search path, while the first copy will be moved up to position 3. You can see this with

```
> attach(pima)
> search()
```

Having several copies of the same data set can lead to great confusion. To detach a data frame, use the command

```
> detach(pima)
```

which will detach the copy in position 2 and move everything else down one position. To detach the second copy repeat the command `detach(pima)`.

1. Use `search()` to make sure you have no data frames attached.
2. Use `objects()` to check that you have the data frame `pima` in your work space.
3. Verify that typing `pima$age` will print the data on the variable `age` but typing `age` will not.
4. Attach the `pima` data frame in position 2 and check that the variables from this data frame are now in position 2.
5. Verify that typing `age` will now print the data on the the variable `age`.
6. Summarize the variable `bmi` for hypertensive women.

2 Reading Data into R

2.1 Introduction

It is said that Mrs Beeton, the 19th century cook and writer, began her recipe for rabbit stew with the instruction First catch your rabbit. Sadly, the story is untrue, but it does contain an important moral. R is a language and environment for data analysis. If you want to do something interesting with it, you need data.

For teaching purposes, data sets are often embedded in R packages. The base R distribution contains a whole package dedicated to data which includes around 100 data sets. This is attached towards the end of the search path, and you can see its contents with

```
> objects("package:datasets")
```

A description of all of these objects is available using the `help()` function. For example


```
> help(Titanic)
```

gives an explanation of the Titanic data set, along with references giving the source of the data.

The `faraway` package also contains some data sets. These are not available automatically when you load the `faraway` package, but you can make a copy in your workspace using the `data()` function. For example

```
> library(faraway)
> data(wheat)
```

To go back to the cooking analogy, these data sets are the equivalent of microwave ready meals, carefully packaged and requiring minimal work by the consumer. Your own data will never be able in this form and you must work harder to read it in to R.

This exercise introduces you to the basics of reading external data into R. It consists of reading the same data from different formats. Although this may appear repetitive, it allows you to see the many options available to you, and should allow you to recognize when things go wrong. You will need the following files in your data directory: `fem.dat`, `fem-dot.dat`, `fem.csv`, `fem.dta`. (download them from <https://wiki.init.mpg.de/IT4Science/RstatisTik/RstatisTikPortal/RcourSe>

2.2 Data Sources

Sources of data can be classified into three groups:

1. Data in human readable form, which can be inspected with a text editor.
2. Data in binary format, which can only be read by a program that understands that format (SAS, SPSS, Stata, Excel, ...).
3. Online data from a database management system (DBMS)

This exercise will deal with the first two forms of data. If you want further details on this topic and also data from DBMSs, you can consult the R Data Import/Export manual that comes with R.

2.3 Data in Text Files

Human-readable data files are generally kept in a rectangular format, with individual records in single rows and variables in columns. Such data can be read into a data frame in R. Before reading in the data, you should inspect the file in a text editor and ask three questions:

1. How are columns in the table separated?
2. How are missing values represented?
3. Are variable names included in the file?

The file **fem.dat** contains data on 118 female psychiatric patients. The data set contains nine variables.

ID	Patient identifier
AGE	Age in years
IQ	Intelligence Quotient (IQ) score
ANXIETY	Anxiety (1=none, 2=mild, 3=moderate,4=severe)
DEPRESS	Depression (1=none, 2=milde, 3=moderate or severe)
SLEEP	Sleeping normally (1=yes, 2=no)
SEX	Lost interest in sex (1=yes, 2=no)
LIFE	Considered suicide (1=yes, 2=no)
WEIGHT	Weight change (kg) in previous 6 months

Inspect the file **fem.dat** with a text editor to answer the questions above. The most general function for reading in free-format data is **read.table()**. This function reads a text file and returns a data frame. It tries to guess the correct format of each variable in the data frame (integer, double precision, or text). Read in the table with:

```
> fem <- read.table("./yourdatadir/fem.dat", header = TRUE)
```

Note that you must assign the result of **read.table()** to an object. If this is not done, the data frame will be printed to the screen and then lost.

You can see the names of the variables with

```
> names(fem)
```

The structure of the data frame can be seen with

```
> str(fem)
```

You can also inspect the top few rows with

```
> head(fem)
```

Note that the IQ of subject 9 is -99, which is an illegal value: nobody can have a negative IQ. In fact -99 has been used in this file to represent a missing value. In R the special value NA (Not Available) is used to represent missing values. All R functions recognize NA values and will handle them appropriately, although sometimes the appropriate response is to stop the calculation with an error message.

You can recode the missing values with

```
> fem$IQ[fem$IQ == -99] <- NA
```

2.4 Things that Can Go Wrong

Sooner or later when reading data into R, you will make a mistake. The frustrating part of reading data into R is that most mistakes are not fatal: they simply cause the function to return a data frame that is *not what you wanted*. There are three common mistakes, which you should learn to recognize.

2.4.1 Forgetting the Headers

The first row of the file `fem.dat` contains the variable names. The `read.table()` function does not assume this by default so you have to specify this with the argument `header=TRUE`. See what happens when you forget to include this option:

```
> fem2 <- read.table("data/fem.dat")
> str(fem2)
> head(fem2)
```

and compare the resulting data frame with `fem`. What are the names of the variables in the data frame? What is the class of the variables?

Explanation: Remember that `read.table()` tries to guess the mode of the variables in the text file. Without the `header=TRUE` option it reads the first row, containing the variable names, as data, and guesses that all the variables are character, not numeric. By default, all character variables are coerced to factors by `read.table`. The result is a data frame consisting entirely of factors (You can prevent the conversion of character variables to factors with the argument `as.is=TRUE`).

If the variable names are not specified in the file, then they are given default names V1, V2, You will soon realise this mistake if you try to access a variable in the data frame by, for example

```
> fem2$IQ
```

as the variable will not exist

There is one case where omitting the `header=TRUE` option is harmless (apart from the situation where there is no header line, obviously). When the first row of the file contains one less value than subsequent lines, `read.table()` infers that the first row contains the variable names, and the first column of every subsequent row contains its row name.

2.4.2 Using the Wrong Separator

By default, `read.table()` assumes that data values are separated by any amount of white space. Other possibilities can be specified using the `sep` argument. See what happens when you assume the wrong separator, in this case a tab, which is specified using the escape sequence `\t`

```
> fem3 <- read.table("data/fem.dat", sep = "\t")
> str(fem3)
```

How many variables are there in the data set?

Explanation: If you mis-specify the separator, `read.table()` reads the whole line as a single character variable. Once again, character variables are coerced to factors, so you get a data frame with a single factor variable.

2.4.3 Mis-specifying the Representation of Missing Values

The file `fem-dot.dat` contains a version of the FEM dataset in which all missing values are represented with a dot. This is a common way of representing missing values, but is not recognized by default by the `read.table()` function, which assumes that missing values are represented by `NA`. Inspect the file with a text editor, and then see what happens when you read the file in incorrectly:

```
> fem4 <- read.table("data/fem-dot.dat", header = TRUE)
> str(fem4)
```

You should have enough clues by now to work out what went wrong. You can read the data correctly using the `na.strings` argument

```
> fem4 <- read.table("data/fem-dot.dat", header = TRUE, na.strings = ".")
```

2.5 Spreadsheet Data

Spreadsheets have become a common way of exchanging data. All spreadsheet programs can save a single sheet in *comma-separated variable (CSV)* format, which can then be read into R. There are two functions in R for reading in CSV data: `read.csv()` and `read.csv2()`.

To understand why there are two functions, inspect the contents of the function `read.csv()` by typing its name

```
> read.csv

function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
fill = TRUE, comment.char = "", ...)
read.table(file = file, header = header, sep = sep, quote = quote,
dec = dec, fill = fill, comment.char = comment.char, ...)
<environment: namespace:utils>
```

The first two lines show the arguments to the `read.csv()` function and their default values (`header=TRUE`, etc) The next two lines show the body of the function, which shows that the default arguments are simply passed verbatim onto the `read.table()` function. Hence `read.csv()` is a wrapper function that chooses the correct arguments for `read.table()` for you. You only need to supply the name of the CSV file and all the other details are taken care of. Now inspect the `read.csv2()` function to find the difference between this function and `read.csv()`.

Explanation: The CSV format is not a single standard. The file format depends on the locale of your computer the settings that determine how numbers are represented. In some countries, the decimal separator is a point `.` and the variable separator in a CSV file is a comma `,`. In other countries, the decimal separator is a comma `,` and the variable separator is a semi-colon `;`. The `read.csv()` function is used for the first format and the `read.csv2()` function is used for the second format.

The file `fem.csv` contains the FEM dataset in CSV format. Inspect the file to work out which format is used, and read it into R.

On Microsoft Windows, you can copy values directly from an open Excel spreadsheet using the clipboard. Highlight the cells you want to copy in the spread sheet and select copy from the pull-down edit menu.

Then type `read.table(file="clipboard")` to read the data in. Beware,

however, that the clipboard on Windows operates on the WYSIWYG principle (what-you-see-is-what-you-get). If you have a value 1.23456789 in your spreadsheet, but have formatted the cell so it is displayed to two decimal places, then the value read into R will be the truncated value 1.23.

2.6 Binary Data

The `foreign` package allows you to read data in binary formats used by other statistical packages. Since R is an open source project, it can only read binary formats that are themselves open, in the sense that the standards for reading and writing data are well-documented.

SAS is an important example. R cannot read SAS datasets. However, the SAS XPORT format is well documented and has been adopted as a data interchange format by the US Food and Drug Administration (<http://www.sas.com/govedu/fda/faq.html>).

Hence there is a function in the `foreign` package for reading SAS XPORT files.

The file `fem.dta` contains the FEM dataset in the format used by Stata. Read it into R with

```
> library(foreign)
> fem5 <- read.dta("data/fem.dta")
> head(fem5)
```

The Stata data set contains value and variable labels. Stata variables with value labels are automatically converted to factors.

There is no equivalent of variable labels in an R data frame, but the original variable labels are not lost. They are still attached to the data frame as an invisible attribute, which you can see with

```
> attr(fem5, "var.labels")
```

A lot of meta-data is attached to the data in the form of attributes. You can see the whole list of attributes with

```
> attributes(fem5)
```

or just the attribute names with

```
> names(attributes(fem5))
```

2.7 Summary

In this exercise we have seen how to create a data frame in R from an external text file. We have also reviewed some common mistakes that result in garbled data.

The capabilities of the `foreign` package for reading binary data have also been demonstrated with a sample `Stata` data set.