

# A short introduction to for Epidemiology

Autumn 2007.

Michael Hills    Retired  
Highgate, London

Martyn Plummer    International Agency for Research on Cancer, Lyon  
[plummer@iarc.fr](mailto:plummer@iarc.fr)

Bendix Carstensen    Steno Diabetes Center, Gentofte, Denmark  
& Department of Biostatistics, University of Copenhagen  
[bxc@steno.dk](mailto:bxc@steno.dk)  
[www.pubhealth.ku.dk/~bxc](http://www.pubhealth.ku.dk/~bxc)



# Contents

<b>1</b>	<b>Some basic commands in R</b>	<b>1</b>
1.1	Preliminaries . . . . .	1
1.2	Using R as a calculator . . . . .	1
1.3	Objects and functions . . . . .	2
1.4	Sequences . . . . .	2
1.5	The births data . . . . .	3
1.6	Reading the data . . . . .	3
1.7	Referencing parts of the data frame . . . . .	4
1.8	Summaries . . . . .	5
1.9	Turning a variable into a factor . . . . .	5
1.10	Frequency tables . . . . .	6
1.11	Grouping the values of a metric variable . . . . .	6
1.12	Tables of means and other things . . . . .	7
1.13	Generating new variables . . . . .	8
1.14	Logical variables . . . . .	8
<b>2</b>	<b>Dates in R</b>	<b>9</b>
<b>3</b>	<b>Working with R</b>	<b>11</b>
3.1	Saving the work space . . . . .	11
3.2	Saving output in a file . . . . .	11
3.3	Saving R objects in a file . . . . .	11
3.4	Using a text editor with R . . . . .	12
3.5	The search path . . . . .	13
3.6	Attaching a data frame . . . . .	13
<b>4</b>	<b>Graphs in R</b>	<b>15</b>
4.1	Simple plot on the screen . . . . .	15
4.2	Colours . . . . .	16
4.3	Adding to a plot . . . . .	16
4.4	Interacting with a plot . . . . .	17
4.5	Saving your graphs for use in other documents . . . . .	18
4.6	The <code>par()</code> command . . . . .	18
4.7	The Lexis diagram . . . . .	19
<b>5</b>	<b>Follow-up data in the Epi package</b>	<b>20</b>
5.1	Timescales . . . . .	20
5.2	Splitting the follow-up time along a timescale . . . . .	24
5.3	Splitting time at a specific date . . . . .	24
5.4	Competing risks — multiple types of events . . . . .	25
<b>6</b>	<b>R command sheet</b>	<b>26</b>



# Chapter 1

## Some basic commands in R

### 1.1 Preliminaries

The purpose of these notes is to describe a small subset of the R language, sufficient to allow someone new to R to get started. The exercises are important because they reinforce basic aspects of R. For further details about R we refer the reader to **An Introduction to R** by W.N.Venables, D.M.Smith, and the R development team. This can be downloaded from the R website at <http://www.r-project.org>.

To start R click on the R icon. To change your working directory click on the tab with this name and select the directory you want to work in. To get out of R click on the File menu and select Exit. You will be offered the chance to save the work space, but at this stage just exit without saving, then start R again, and change the working directory, as before.

R is case sensitive, so that **A** is different from **a**. Commands in R are generally separated by a newline, although a semi-colon can also be used. When using R it makes sense to avoid as much typing as possible by recalling previous commands using the vertical arrow key and editing them.

### 1.2 Using R as a calculator

Typing `2+2` will return the answer 4, typing `2^3` will return the answer 8 (2 to the power of 3), typing `log(10)` will return the natural logarithm of 10, which is 2.3026, and typing `sqrt(25)` will return the square root of 25.

Instead of printing the result you can store it in an object, say

```
> a <- 2 + 2
```

which can be used in further calculations. The expression `<-`, pronounced "gets", is called the assignment operator, and is obtained by typing `<` and then `-`. The assignment operator can also be used in the opposite direction, as in

```
> a <- 2 + 2
```

The contents of `a` can be printed by typing `a`.

Standard probability functions are readily available. For example, the probability below 1.96 in a standard normal (i.e. Gaussian) distribution is obtained with

```
> pnorm(1.96)
```

while

```
> pchisq(3.84, 1)
```

will return the probability below 3.84 in a  $\chi^2$  distribution on 1 degree of freedom, and

```
> pchisq(3.84, 1, lower.tail = FALSE)
```

will return the probability above 3.84.

### Exercise 1.

1. Calculate  $\sqrt{3^2 + 4^2}$ .
2. Find the probability above 4.3 in a chi-squared distribution on 1 degree of freedom.

## 1.3 Objects and functions

All commands in R are *functions* which act on *objects*. One important kind of object is a *vector*, which is an ordered collections of numbers, or an ordered collection of character strings. Examples of vectors are 4, 6, 1, 2.2, which is a numeric vector with 4 components, and “Charles Darwin”, “Alfred Wallace” which is a vector of character strings with 2 components. The components of a vector must be of the same type (numeric or character). The combine function `c()`, together with the assignment operator, is used to create vectors. Thus

```
> v <- c(4, 6, 1, 2.2)
```

creates a vector `v` with components 4, 6, 1, 2.2 by first combining the 4 numbers 4, 6, 1, 2.2 in order and then assigning the result to the vector `v`. Collections of components of different types are called *lists*, and are created with the `list()` function. Thus

```
> m <- list(4, 6, "name of company")
```

creates a list with 3 components. The main differences between the numbers 4, 6, 1, 2.2 and the vector `v` is that along with `v` is stored information about what sort of object it is and hence how it is printed and how it is combined with other objects. Try

```
> v
> 3 + v
> 3 * v
```

and you will see that R understands what to do in each case. This may seem trivial, but remember that unlike most statistical packages there are many different kinds of object in R.

You can get a description of the structure of any object using the function `str()`. For example, `str(v)` shows that `v` is numeric with 4 components.

## 1.4 Sequences

It is not always necessary to type out all the components of a vector. For example, the vector (15, 20, 25, ... ,85) can be created with

```
> seq(15, 85, by = 5)
```

and the vector (5, 20, 25, ... ,85) can be created with

```
> c(5, seq(20, 85, by = 5))
```

You can learn more about functions by typing `?`  followed by the function name. For example `?seq` gives information about the syntax and usage of the function `seq()`.

### Exercise 2.

1. Create a vector `w` with components 1, -1, 2, -2
2. Print this vector (to the screen)
3. Obtain a description of `w` using `str()`
4. Create the vector `w+1`, and print it.
5. Create the vector (0, 1, 5, 10, 15, ... , 75) using `c()` and `seq()`.

## 1.5 The births data

Table 1.1: *Variables in the births dataset*

Variable	Units or Coding	Type	Name
Subject number	—	categorical	<code>id</code>
Birth weight	grams	metric	<code>bweight</code>
Birth weight < 2500 g	1=yes, 0=no	categorical	<code>lowbw</code>
Gestational age	weeks	metric	<code>gestwks</code>
Gestational age < 37 weeks	1=yes, 0=no	categorical	<code>preterm</code>
Maternal age	years	metric	<code>matage</code>
Maternal hypertension	1=hypertensive, 0=normal	categorical	<code>hyp</code>
Sex of baby	1=male, 2=female	categorical	<code>sex</code>

The most important example of a vector in epidemiology is the data on a variable recorded for a group of subjects. To introduce R we use the births data which concern 500 mothers who had singleton births in a large London hospital. These data are available as an R object called `births` in the Epi package.

Some of the variables which make up these data take integer values while others are numeric taking measurements as values. For most variables the integer values are just codes for different categories, such as "male" and "female" which are coded 1 and 2 for the variable `sex`. You can browse the "births.txt" file by clicking on *Display files* under the *File menu*. It is a tab delimited file, that is the individual items of data are separated by the code for Tab. This is the sort of file you get from a spreadsheet such as Excel. Missing values are left blank, so they appear as TabTab.

## 1.6 Reading the data

The easiest way to access the births data is first to load the Epi package with

```
> library(Epi)
```

and then to load the data with

```
> data(births)
```

Try

```
> objects()
```

to make sure that you have an object called `births` in your working directory. The function

```
> str(births)
```

shows that the object `births` is a data frame with 500 observations of 8 variables. The names and types of the variables are also shown together with the first 10 values of each variable.

### Exercise 3.

1. The dataframe "`diet`" in the `Epi` package contains data from a follow-up study with coronary heart disease as the end-point. Load these data with  

```
> data(diet)
```

and print the contents of the data frame to the screen..
2. Check that you now have two objects, `births`, and `diet` in your work space.
3. Obtain a description of the object `diet`.
4. Remove the object `diet` with the command  

```
> rm(diet)
```

Check that you only have the object `births` left.

## 1.7 Referencing parts of the data frame

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

```
> births[1, "bweight"]
```

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

```
> births[2, "bweight"]
```

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

```
> births[1:10, "bweight"]
```

and to list all the data for this variable, try

```
> births[, "bweight"]
```

### Exercise 4.

1. Print the data on the variable `gestwks` for subject 7 in the `births` data frame.
2. Print all the data for subject 7.
3. Print all the data on the variable `gestwks`.



## 1.8 Summaries

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

```
> summary(births)
```

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

```
> names(births)
```

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 `births$hyp` refers to the variable `hyp` in the `births` data frame, and typing `births$hyp` will print the data on this variable. To summarize the variable `hyp` try

```
> summary(births$hyp)
```

In most datasets there will be some missing values. These are usually coded using tab delimited blanks to mark the values which are missing. R then codes the missing values using the `NA` (not available) symbol. The summary shows the number of missing values for each variable.

## 1.9 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 `hyp` and `sex` are 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 `hyp` to be a factor, try

```
> hyp <- factor(births$hyp)
> str(births)
> objects()
```

which shows that `hyp` is both in your work space (as a factor), and in in the `births` data frame (as a numeric variable). It is better to use the transform function on the data frame, as in

```
> births <- transform(births, hyp = factor(hyp))
> str(births)
```

which shows that `hyp`, in the `births` data frame, 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) `"normal"` and `"hyper"` with

```
> births <- transform(births, hyp = factor(hyp, labels=c("normal",
+ "hyper"))))
> str(births)
```

### Exercise 5.

1. Convert the variable `sex` into a factor
2. Label the levels of `sex` as `"male"` and `"female"`.

## 1.10 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 tables is `stat.table`. This is currently part of the Epi package, so you will need to load this package first with

```
> library(Epi)
```

The distribution of the factors `hyp` and `sex` can be viewed by typing

```
> stat.table(hyp, data = births)
> stat.table(sex, data = births)
```

Their cross-tabulation is obtained by typing

```
> stat.table(list(hyp, sex), data = births)
```

Cross-tabulations are useful when checking for consistency, but because no distinction is drawn between the response variable and any explanatory variables, they are not useful as a way of presenting data.

## 1.11 Grouping the values of a metric variable

For a numeric variable like `matage` 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 `matage` into the groups 20–29, 30–34, 35–39, 40–44, and then create a factor called `agegrp` with 4 levels corresponding to the four groups. The best way of doing this is with the function `cut`. Try

```
> births <- transform(births, agegrp = cut(matage, breaks = c(20,
+ 30, 35, 40, 45), right = FALSE))
> stat.table(agegrp, data = births)
```

By default the factor levels are labelled [20-25), [25-30), etc., where [20-25) refers to the interval which includes the left hand end (20) but not the right hand end (25). 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.

It is important to realize that observations which are not inside the range specified in the `breaks()` part of the command result in missing values for the new factor. For example, try

```
> births <- transform(births, agegrp = cut(matage, breaks = c(20,
+ 30, 35), right = FALSE))
> summary(births)
```

Only observations from 20 up to, but not including 35, are included. For the rest, `agegrp` is coded missing. 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

```
> births <- transform(births, agegrp = cut(matage, breaks = 5,
+ right = FALSE))
> stat.table(agegrp, data = births)
```

shows 5 intervals of width 4.

**Exercise 6.**

1. Summarize the numeric variable `gestwks`, which records the length of gestation for the baby, and make a note of the range of values.
2. Create a new factor `gest4` which cuts `gestwks` at 20, 35, 37, 39, and 45 weeks, including the left hand end, but not the right hand. Make a table of the frequencies for the four levels of `gest4`.
3. Create a new factor `gest5` which cuts `gestwks` into 5 equal intervals, and make a table of frequencies.

**1.12 Tables of means and other things**

To obtain the mean of `bweight` by `sex`, try

```
> stat.table(sex, mean(bweight), data = births)
```

The headings of the table can be improved with

```
> stat.table(sex, list("Mean birth weight" = mean(bweight)), data = births)
```

To make a two-way table of mean birth weight by `sex` and `hypertension`, try

```
> stat.table(list(sex, hyp), mean(bweight), data = births)
```

and to tabulate the count as well as the mean, try

```
> stat.table(list(sex, hyp), list(count(), mean(bweight)), data = births)
```

Available functions for the cells of the table are `count`, `mean`, `weighted.mean`, `sum`, `min`, `max`, `quantile`, `median`, `IQR`, and `ratio`. The last of these is useful for rates and odds. For example, to make a table of the odds of low birth weight by `hypertension`, try

```
> stat.table(hyp, list(odds = ratio(lowbw, 1 - lowbw, 100)), data = births)
```

The scale factor 100 makes the odds per 100. Margins can be added to the tables, as required. For example,

```
> stat.table(sex, mean(bweight), data = births, margins = TRUE)
```

for a one-way table, and

```
> stat.table(list(sex, hyp), mean(bweight), data = births, margins = c(TRUE,
+ FALSE))
> stat.table(list(sex, hyp), mean(bweight), data = births, margins = c(FALSE,
+ TRUE))
> stat.table(list(sex, hyp), mean(bweight), data = births, margins = c(TRUE,
+ TRUE))
```

for a two-way table.

**Exercise 7.**

1. Make a table of median birth weight by `sex`.
2. Do the same for gestation time, but include `count` as a function to be tabulated along with `median`. Note that when there are missing values for the variable being summarized the count refers to the number of non-missing observations for the row variable, not the summarized variable.
3. Create a table showing the mean gestation time for the baby by `hyp` and `lowbw`, together with margins for both.
4. Make a table showing the odds of hypertension by sex of the baby.

## 1.13 Generating new variables

New variables can be produced using assignment together with the usual mathematical operations and functions:

+      -      \*      log      exp      ^      sqrt

The sign `^` means “to the power of”, `log` means “natural logarithm”, and `sqrt` means “square root”.

The `transform()` function allows you to transform or generate variables in a data frame. For example, try

```
> births <- transform(births, num1 = 1, num2 = 2, logbw = log(bweight))
```

The variable `logbw` is the natural logarithm of birth weight. Logs base 10 are obtained with `log10( )`.

## 1.14 Logical variables

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

```
> births <- transform(births, low = bweight < 2000)
> str(births)
```

creates a logical variable `low` with levels TRUE and FALSE, according to whether `bweight` is less than 2000 or not. The logical expressions which R allows are

==      <      <=      >      >=      !=

The first is logical equals and the last is not equals. One common use of logical variables is to restrict a command to a subset of the data. For example, to list the values taken by `bweight` for hypertensive women, try

```
> births$bweight[births$hyp == "hyper"]
```

If you want the entire dataframe restricted to hypertensive women try:

```
> births[births$hyp == "hyper", ]
```

The `subset()` function also allows you to take a subset of a data frame. Try

```
> subset(births, hyp == "hyper")
```

### Exercise 8.

1. Create a logical variable called `early` according to whether `gestwks` is less than 30 or not. Make a frequency table of `early`.
2. Print the `id` numbers of women with `gestwks` less than 30 weeks.

## Chapter 2

# Dates in R

Epidemiological studies often contain date variables which take values such as 2/11/1962. We shall use the diet data to illustrate how to deal with variables whose values are dates.

The important variables in the dataset are `chd`, which takes the value 1 if the subject develops coronary heart disease during the study the value 0 if the observation is censored, and the three date variables which are date of birth (`dob`), date of entry (`doe`) and date of exit (`dox`). The command

```
> str(diet)
```

shows that these three variables are `Date` variables.

You will also see that the values are just numbers, but if you try

```
> head(diet)
```

you will see these variables printed as “real” dates. The variables are internally stored as number of days since 1/1/1970.

To convert a character string (or a character variable) to date format try:

```
> as.Date("14/07/1952", format = "%d/%m/%Y")
> as.numeric(as.Date("14/07/1952", format = "%d/%m/%Y"))
```

The first form shows the date form and the latter the number of days since 1/1/1970, which is a negative number for dates prior to 1/1/1970.

The format parts, “%d” etc., identify elements of the dates, whereas the “/”s are just the separator characters that are in the character string. There are other possibilities for formats, see `?strftime` or the section on dates and times in the R command sheet at the end of this document.

Reading dates from an external file is done by reading the fields as character variables and then transforming them to date variables by the function `as.Date`

If you want to enter a fixed date, for example if you want to terminate follow-up at 1st April 1975 you could say:

```
> newx <- pmin(diet$dox, as.Date("1975-4-1", format = "%F"))
```

The format `%F` is shorthand for the ISO-standard date representation `%Y-%m-%d`, which is the default, so it can be omitted altogether:

```
> newx <- pmin(diet$dox, as.Date("1975-4-1"))
```

You can print dates in the format you like by using the function `format.Date()`, try for example:

```
> bdat <- as.Date("1952-7-14", format = "%F")  
> format.Date(bdat, format = "%A %d %B %Y")
```

**Exercise 9.**

1. Convert `doe` and `dox` to date variables.
2. Generate a new variable `y` which is the elapsed time in years between the date of entry and the date of exit.
3. The file `getdiet.R` reads the diet data, converts all three date variables to standard form using the `transform` function, and generates the variable `y`. Run this script and check the results are what you want.
4. Enter your own birthday as a date. Print it using `format.Date()` with the format `"%A %d %B %Y"`. Did you learn anything new?
5. Enter the birthday of your husband/wife/... as a date too. When will you be (were you) 100 years old together? (Hint: `mean()` works on vectors of dates as well.)

## Chapter 3

# Working with R

### 3.1 Saving the work space

When exiting from R you are offered the chance of saving all the objects in your current work space. If you do so, the work space is re-instated next time you start R. It can be useful to do this, but before doing so it is worth tidying things up, because the work space can fill up with temporary objects, and it is easy to forget what these are when you resume the session.

### 3.2 Saving output in a file

To save the output from an R command in a file, for future use, the `sink()` command is used. For example,

```
> sink("output.txt")
> summary(births)
```

first instructs R to re-direct output away from the R terminal to the file "output.txt" and then summarizes the births data frame, the output from which goes to the sink. While a sink is open all output will go to it, replacing what is already in the file. To append output to a file, use the `append=TRUE` option with `sink()`. To close a sink, use

```
> sink()
```

#### Exercise 10.

1. Sink output to a file called "output1.txt".
2. Make frequency tables of `hyp` and `sex`
3. Make a table of mean birth weight by sex
4. Close the sink
5. From windows, have a look inside the file `output1.txt` and check that the output you expected is in the file.

### 3.3 Saving R objects in a file

The command `read.table()` is relatively slow because it carries out quite a lot of processing as it reads the data. To avoid doing this more than once you can save the data frame, which includes the R information, and read from this saved file in future. For example,

```
> save(births, file = "births.Rdata")
```

will save the `births` data frame in the file `births.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("births.Rdata")
```

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

### Exercise 11.

1. Use `read.table()` to read the data in the file `diet.txt` into a data frame called `diet`.
2. Save this data frame in the file `"diet.Rdata"`
3. Remove the data frame
4. Load the data frame from the file `"diet.Rdata"`.

## 3.4 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. This means you can use the cut and paste facilities of the editor to cut down on typing. For Windows we recommend using the text editor Tinn-R, but you can use your favourite text editor instead if you prefer. Start up the editor and enter the following lines:

```
> births <- transform(births, lowbw = factor(lowbw, labels = c("normal",
+   "low")), hyp = factor(hyp, labels = c("normal", "hyper")),
+   sex = factor(sex, labels = c("male", "female")))
```



Now save the script as `mygetbirths.R` and run it. 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.

### Exercise 12.

1. Create a script called `mytab.R` which includes the lines
 

```
> stat.table(hyp, data = births)
> stat.table(sex, data = births)
```

 and run just these two lines.
2. Edit the script to include the lines
 

```
> stat.table(sex, mean(bweight), data = births)
> stat.table(hyp, mean(bweight), data = births)
```

 and run these two lines.



3. Edit the script to create a factor cutting **matage** at 20, 30, 35, 40, 45 years, and run just this part of the script.
4. Edit the script to create a factor cutting **gestwks** at 20, 35, 37, 39, 45 weeks, and run just this part of the script.
5. Save and run the entire script.

### 3.5 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 Epi package, the third is a package of commands called methods, the fourth is a package called stats, and so on. To see what is in the work space try

```
> objects()
```

You should see just the objects **births** and **diet**. The command **objects(1)** does the same as **objects()**. To see what is in the Epi package, try

```
> objects(2)
```

There are 29 functions 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.

### 3.6 Attaching a data frame

The function **objects(1)** shows that the only objects in the workspace are **births** and **diet**. To refer to variables in the **births** data frame by name it is necessary to specify the name of the data frame, as in **births\$hyp**. 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(births)
```

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

```
> objects(2)
```

which shows the objects in this position are the variables from the **births** data frame. Note that the methods package has now been moved up to position 3, as shown by the **search()** function.

When you type the command:

```
> hyp
```

R will look in the first position where it fails to find **hyp**, then the second position where it finds **hyp**, 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 <- bweight[hyp == 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(births)
> search()
```

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

```
> detach(births)
```

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

### Exercise 13.

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

```
> setwd(sweave.wd)
```

## Chapter 4

# Graphs in R

There are three kinds of plotting functions in R:

1. Functions that generate a new plot, e.g. `hist()` and `plot()`.
2. Functions that add extra things to an existing plot, e.g. `lines()` and `text()`.
3. Functions that allow you to interact with the plot, e.g. `locator()` and `identify()`.

The normal procedure for making a graph in R is to make a fairly simple initial plot and then add on points, lines, text etc., preferably in a script.

### 4.1 Simple plot on the screen

Load the births data and get an overview of the variables:

```
> library(Epi)
> data(births)
> str(births)
```

Now attach the dataframe and look at the birthweight distribution with

```
> attach(births)
> hist(bweight)
```

The histogram can be refined – take a look at the possible options with

```
> `?`(hist)
```

and try some of the options, for example:

```
> hist(bweight, col = "gray", border = "white")
```

To look at the relationship between birthweight and gestational weeks, try

```
> plot(gestwks, bweight)
```

You can change the plot-symbol by the option `pch=`. If you want to see all the plot symbols try:

```
> plot(1:25, pch = 1:25)
```

#### Exercise 14.

1. Make a plot of the birth weight versus maternal age with  

```
> plot(matage, bweight)
```
2. Label the axes with  

```
> plot(matage, bweight, xlab = "Maternal age", ylab = "Birth weight (g)")
```

## 4.2 Colours

There are many colours recognized by **R**. You can list them all by `colours()` or, equivalently, `colors()` (**R** allows you to use British or American spelling). To colour the points of birthweight versus gestational weeks, try

```
> plot(gestwks, bweight, pch = 16, col = "green")
```

This creates a solid mass of colour in the centre of the cluster of points and it is no longer possible to see individual points. You can recover this information by overwriting the points with black circles using the `points()` function.

```
> points(gestwks, bweight)
```

## 4.3 Adding to a plot

The `points()` function is one of several functions that add elements to an existing plot. By using these functions, you can create quite complex graphs in small steps.

Suppose we wish to recreate the plot of birthweight *vs* gestational weeks using different colours for male and female babies. To start with an empty plot, try

```
> plot(gestwks, bweight, type = "n")
```

Then add the points with the `points` function.

```
> points(gestwks[sex == 1], bweight[sex == 1], col = "blue")
> points(gestwks[sex == 2], bweight[sex == 2], col = "red")
```

To add a legend explaining the colours, try

```
> legend("topleft", pch = 1, legend = c("Boys", "Girls"), col = c("blue",
+ "red"))
```

which puts the legend in the top left hand corner.

Finally we can add a title to the plot with

```
> title("Birth weight vs gestational weeks in 500 singleton births")
```

## Using indexing for plot elements

One of the most powerful features of **R** is the possibility to index vectors, not only to get subsets of them, but also for repeating their elements in complex sequences.

Putting separate colours on males and female as above would become very clumsy if we had a 5 level factor instead.

Instead of specifying one color for all points, we may specify a vector of colours of the same length as the `gestwks` and `bweight` vectors. This is rather tedious to do directly, but **R** allows you to specify an expression anywhere, so we can use the fact that `sex` takes the values 1 and 2, as follows:

First create a colour vector with two colours, and take look at `sex`:

```
> c("blue", "red")
> sex
```

Now see what happens if you index the colour vector by `sex`:

```
> c("blue", "red")[sex]
```

For every occurrence of a 1 in `sex` you get "blue", and for every occurrence of 2 you get "red", so the result is a long vector of "blue"s and "red"s corresponding to the males and females. This can now be used in the plot:

```
> plot(gestwks, bweight, pch = 16, col = c("blue", "red")[sex])
```

The same trick can be used if we want to have a separate symbol for mothers over 40 say. We first generate the indexing variable:

```
> oldmum <- (matage >= 40) + 1
```

Note we add 1 because ( `matage >= 40` ) generates a logic variable, so by adding 1 we get a numeric variable with values 1 and 2, suitable for indexing:

```
> plot(gestwks, bweight, pch = c(16, 3)[oldmum], col = c("blue",
+ "red")[sex])
```

so where `oldmum` is 1 we get `pch=16` (a dot) and where `oldmum` is 2 we get `pch=3` (a cross).

R will accept any kind of complexity in the indexing as long as the result is a valid index, so you don't need to create the variable `oldmum`, you can create it on the fly:

```
> plot(gestwks, bweight, pch = c(16, 3)[(matage >= 40) + 1], col = c("blue",
+ "red")[sex])
```

### Exercise 15.

1. Make a three level factor for maternal age with cutpoints at 30 and 40 years.
2. Use this to make the plot of gestational weeks with three different plotting symbols. (Hint: Indexing with a factor automatically gives indexes 1,2,3 etc.).

## Generating colours

R has functions that generate a vector of colours for you. For example,

```
> rainbow(4)
```

produces a vector with 4 colours (not immediately human readable, though). There are a few other functions that generates other sequences of colours, type `?rainbow` to see them.

Gray-tones are produced by the function `gray` (or `grey`), which takes a numerical argument between 0 and 1; `gray(0)` is black and `gray(1)` is white. Try:

```
> plot(0:10, pch = 16, cex = 3, col = gray(0:10/10))
> points(0:10, pch = 1, cex = 3)
```

## 4.4 Interacting with a plot

The `locator()` function allows you to interact with the plot using the mouse. Typing `locator(1)` shifts you to the graphics window and waits for one click of the left mouse button. When you click, it will return the corresponding coordinates.

You can use `locator()` inside other graphics functions to position graphical elements exactly where you want them. Recreate the birth-weight plot,

```
> plot(gestwks, bweight, pch = c(16, 3)[(matage >= 40) + 1], col = c("blue",
+ "red")[sex])
```

```
> print(1:10)
```

and then add the legend where you wish it to appear by typing

```
> legend(locator(1), pch = 1, legend = c("Boys", "Girls"), col = c("blue",
+ "red"))
```

The `identify()` function allows you to find out which records in the data correspond to points on the graph. Try

```
> identify(gestwks, bweight)
```

When you click the left mouse button, a label will appear on the graph identifying the row number of the nearest point in the data frame `births`. If there is no point nearby, **R** will print a warning message on the console instead. To end the interaction with the graphics window, right click the mouse: the `identify` function returns a vector of identified points.

#### Exercise 16.

1. Use `identify()` to find which records correspond to the smallest and largest number of gestational weeks.
2. View all the variables corresponding to these records with

```
births[identify(gestwks, bweight), ]
```

## 4.5 Saving your graphs for use in other documents

Once you have a graph on the screen you can click on `File` → `Save as`, and choose the format you want your graph in. The PDF (Acrobat reader) format is normally the most economical, and Acrobat reader has good options for viewing in more detail on the screen. The **Metafile** format will give you an enhanced metafile `.emf`, which can be imported into a Word document by `Insert` → `Picture` → `From File`. Metafiles can be resized and edited inside Word.

If you want exact control of the size of your plot-file you can start a graphics device *before* doing the plot. Instead of appearing on the screen, the plot will be written directly to a file. After the plot has been completed you will need to close the device again in order to be able to access the file. Try:

```
> win.metafile(file = "plot1.emf", height = 3, width = 4)
> plot(gestwks, bweight)
> dev.off()
```

This will give you an enhanced metafile `plot1.emf` with a graph which is 3 inches tall and 4 inches wide.

## 4.6 The `par()` command

It is possible to manipulate any element in a graph, by using the graphics options. These are collected on the help page of `par()`. For example, if you want axis labels always to be horizontal, use the command `par(las=1)`. This will be in effect until a new graphics device is opened.

Look at the typewriter-version of the help-page with

```
> ??(par)
```

or better, use the the html-version through `Help` → `Html help` → `Packages` → `base` → `P` → `par`.



It is a good idea to take a print of this (having set the text size to “smallest” because it is long) and carry it with you at any time to read in buses, cinema queues, during boring lectures etc. Don’t despair, few R-users can understand what all the options are for.

`par()` can also be used to ask about the current plot, for example `par("usr")` will give you the exact extent of the axes in the current plot.

If you want more plots on a single page you can use the command

```
> par(mfrow = c(2, 3))
```

This will give you a layout of 2 rows by 3 columns for the next 6 graphs you produce. The plots will appear by row, i.e. in the top row first. If you want the plots to appear columnwise, use `par( mfcol=c(2,3) )` (you still get 2 rows by 3 columns). To restore the layout to a single plot per page use

```
> par(mfrow = c(1, 1))
```

## 4.7 The Lexis diagram

If you load the Epi package you can draw a Lexis diagram by:

```
> Lexis.diagram()
```

You can draw the lifelines of the members of the diet study in the diagram by:

```
> data(diet)
> Lexis.lines(entry.date = diet$doe, exit.date = diet$dox, birth.date = diet$dob,
+           fail = diet$chd)
```

`Lexis.lines` recognizes variables of format `Date` and automatically converts them to fractional years when plotting them. The age and date axes are in years, so if you want to add text or other things to the diagram you must give coordinates in years.

You would probably want to adjust the axes of the Lexis diagram, for example by:

```
> Lexis.diagram(age = c(30, 75), date = c(1950, 1995))
```

## Chapter 5

# Follow-up data in the Epi package

In the `Epi`-package, follow-up data is represented by adding some extra variables to a dataframe. The tools for handling follow-up data then use these for special plots and tabulations etc.

Follow-up data basically consists of a time of entry, a time of exit and an indication of the status at exit (frequently “alive” or “dead”). Implicitly is also assumed a status *during* the follow-up (usually “alive”).

These three variables are specific for each *type* of outcome, i.e. cancer, cardiovascular event, death, ...

### 5.1 Timescales

A timescale is a variable that varies deterministically *within* each person during follow-up, *e.g.*:

- Age
- Calendar time
- Time since treatment
- Time since relapse

All timescales advance at the same pace, so the time followed is the same on all timescales. Therefore, it suffices to use only the entry point on each of the time scale, for example:

- Age at entry.
- Date of entry.
- Time since diagnosis.
- Time since entry.

In the `Epi` package, follow-up in a cohort is represented in a `Lexis` object. A `Lexis` object is a dataframe with a bit of extra structure representing the follow-up. For the `nickel` data we would define:

```
> data( nickel )
> nicL <- Lexis( entry = list( per=agein+dob,
+                               age=agein,
```



```
+               tfh=agein-age1st ),
+               exit = list( age=ageout ),
+               exit.status = ( icd %in% c(162,163) )*1,
+               data = nickel )
```

The `entry` argument is a named list with the entry points on each of the timescales we want to use. It defines the names of the timescales and the entry points. The `exit` argument gives the exit time on *one* of the timescales. This is sufficient, because the follow-up time on all time scale is the same, in this case `ageout - agein`. Take a look at the result:

```
> str( nickel )
> str( nicL )
> head( nicL )
```

The `Lexis` object `nicL` has a variable for each timescale which is the entry point on this timescale. The follow-up time is in the variable `lex.dur` (**d**uration).

We defined the exit status to be death from lung cancer (ICD7 162,163), i.e. this variable is 1 if follow-up ended with a death from this cause. If follow-up ended alive or by death from another cause, the exit status is coded 0, i.e. aa a censoring.

Note that the exit status is in the variable `lex.Xst` (**eXit status**). The variable `lex.Csat` is the state where the follow-up takes place (**C**urrent **s**tatus), in this case 0 (alive).

It is possible to get a visualization of the follow-up along the timescales chosen by using the `plot` method for `Lexis` objects. `nicL` is an object of *class* `Lexis`, so using the function `plot()` on it means that **R** will look for the function `plot.Lexis` and use this function.

```
> pdf( file="nicL1.pdf" )
> plot( nicL )
> dev.off()
```

The function allows a lot of control over the output, and a `points.Lexis` function allows plotting of the endpoints of follow-up.

```
> pdf( file="nicL2.pdf", height=7, width=7)
> par( mar=c(3,3,1,1), mgp=c(3,1,0)/1.6 )
> plot( nicL, 1:2, lwd=1, col=c("blue","red")[(nicL$exp>0)+1],
+       grid=TRUE, lty.grid=1, col.grid=gray(0.7),
+       xlim=1900+c(0,90), xaxs="i",
+       ylim= 10+c(0,90), yaxs="i", las=1 )
> points( nicL, 1:2, pch=c(NA,3)[nicL$lex.Xst+1],
+         col="lightgray", lwd=3, cex=1.5 )
> points( nicL, 1:2, pch=c(NA,3)[nicL$lex.Xst+1],
+         col=c("blue","red")[(nicL$exp>0)+1], lwd=1, cex=1.5 )
> dev.off()
```

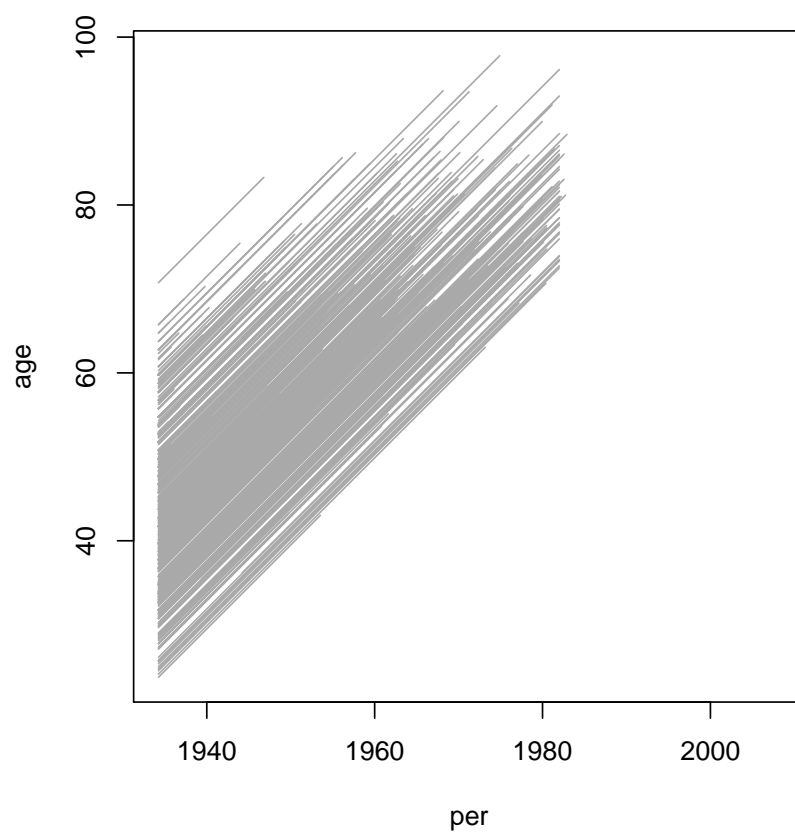


Figure 5.1: *Lexis diagram of the **nickel** dataset.*

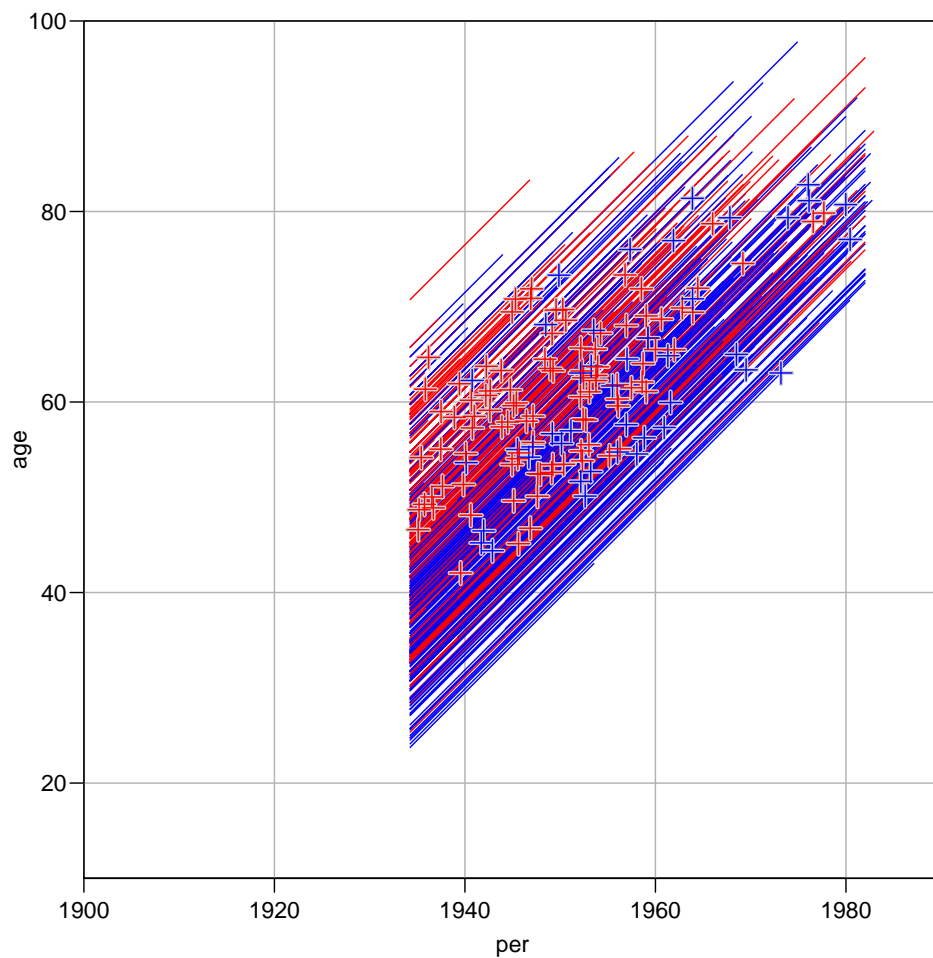


Figure 5.2: *Lexis diagram of the `nickel` dataset, with bells and whistles. The red lines are for persons with `exposure > 0`, so it is pretty evident that the oldest ones are the exposed part of the cohort.*

## 5.2 Splitting the follow-up time along a timescale

The follow-up time in a cohort can be subdivided by for example current age. This is achieved by the `splitLexis` (note it is *not* called `split.Lexis`). This requires that the timescale and the breakpoints on this timescale are supplied. Try:

```
> nicS1 <- splitLexis( nicL, "age", breaks=seq(0,100,10) )
> str( nicL )
> str( nicS1 )
> round( subset( nicS1, id %in% 8:10 ), 2 )
```

The resulting object is again a `Lexis` object, and so follow-up may be split further along another timescale. Try this and list the result for individuals 4 and 6:

```
> nicS2 <- splitLexis( nicS1, "tfh", breaks=c(0,1,5,10,20,30,100) )
> round( subset( nicS2, id %in% 8:10 ), 2 )
```

If we want to model the effect of these timescales we will for each interval use either the value of the left endpoint in each interval or the middle. There is a function `timeBand` which returns these. Try:

```
> timeBand( nicS2, "age", "middle" )[1:20]
> # For nice printing and column labelling use the data.frame() function:
> data.frame( nicS2[,c("id", "lex.id", "per", "age", "tfh", "lex.dur")],
+           mid.age=timeBand( nicS2, "age", "middle" ),
+           mid.tfh=timeBand( nicS2, "tfh", "middle" ) )[1:20,]
```

Note that these are the midpoints of the intervals defined by `breaks=`, *not* the midpoints of the actual follow-up intervals. This is because the variable to be used in modelling must be independent of the censoring and mortality pattern — it should only depend on the chosen grouping of the timescale.

## 5.3 Splitting time at a specific date

If we have a recording of the date of a specific event as for example recovery or relapse, we may classify follow-up time as being before of after this intermediate event. This is achieved with the function `cutLexis`, which takes three arguments: the time point, the timescale, and the value of the (new) state following the date.

Now we define the age for the nickel vorkers where the cumulative exposure exceeds 50 exposure years:

```
> subset( nicL, id %in% 8:10 )
> nicL$agehi <- nicL$age1st + 50 / nicL$exposure
> nicC <- cutLexis( nicL, nicL$agehi, "age", 2 )
> subset( nicC, id %in% 8:10 )
```

Note that individual 6 has had his follow-up split at age 25 where 50 exposure-years were attained. This could also have been achieved in the split dataset, try:

```
> subset( nicS2, id %in% 8:10 )
> nicS2$agehi <- nicS2$age1st + 50 / nicS2$exposure
> nicS2C <- cutLexis( nicS2, nicS2$agehi, "age", 2 )
> subset( nicS2C, id %in% 8:10 )
```

Note that follow-up subsequent to the event is classified as being in state 2, but that the final transition to state (death from lung cancer) is preserved.

If the intermediate event is to be used as a time-dependent variable in a Cox-model, then `lex.Cst` should be used as the time-dependent variable, and `lex.Xst==1` as the event.

## 5.4 Competing risks — multiple types of events

If we want to consider death from lung cancer and death from other causes as separate events we can code these as for example 1 and 2.

```
> data( nickel )
> nicL <- Lexis( entry = list( per=agein+dob,
+                             age=agein,
+                             tfh=agein-age1st ),
+               exit = list( age=ageout ),
+               exit.status = ( icd > 0 ) + ( icd %in% c(162,163) ),
+               data = nickel )
> str( nicL )
> head( nicL )
> subset( nicL, id %in% 8:10 )
```

If we want to label the states, we can enter the names of these in the `states` parameter, try for example:

```
> nicL <- Lexis( entry = list( per=agein+dob,
+                             age=agein,
+                             tfh=agein-age1st ),
+               exit = list( age=ageout ),
+               exit.status = ( icd > 0 ) + ( icd %in% c(162,163) ),
+               data = nickel,
+               states = c("Alive", "D.oth", "D.lung") )
> str( nicL )
```

You can get an overview of the number of records by state and transitions between states as well as the person-years in each state by using `tab.Lexis()`, try:

```
> tab.Lexis( nicL )
```

When we cut at a date as in the case the date where cumulative exposure exceeds 50 exposure-years, we get the follow-up *after* the date classified as being in the new state if it was in a state we

```
> nicL$agehi <- nicL$age1st + 50 / nicL$exposure
> nicC <- cutLexis( nicL, nicL$agehi, "age", new.state="HiExp", cens="Alive" )
> subset( nicC, id %in% 8:10 )
> tab.Lexis( nicC )
```

Note that the person-years is the same, but that the number of events has changed. This is because events are now defined as any transition from alive, including the transitions to `HiExp`.

## Chapter 6

# R command sheet

This R Reference Card is written by Tom Short, EPRI PEAC, [tshort@epri-peac.com](mailto:tshort@epri-peac.com), 2004-10-21 and granted to the public domain. See [www.Rpad.org](http://www.Rpad.org) for the source and latest version. Includes material from *R for Beginners* by Emmanuel Paradis (with permission).

It is also available separately as a 4-page landscape document from the R-hompage [www.r-project.org](http://www.r-project.org), Manuals → contributed documentation.

### Getting help

Most R functions have online documentation.

- `help(topic)` documentation on `topic`
- `?topic` — the same.
- `help.search("topic")` search the help system
- `apropos("topic")` the names of all objects in the search list matching the regular expression "topic"
- `help.start()` start the HTML version of help
- `str(a)` display the internal \*str\*ucture of an R object
- `summary(a)` gives a "summary" of `a`, usually a statistical summary but it is *generic* meaning it has different operations for different classes of `a`
- `ls()` show objects in the search path; specify `pat="pat"` to search on a pattern
- `ls.str()` `str()` for each variable in the search path
- `dir()` show files in the current directory
- `methods(a)` shows S3 methods of `a`
- `methods(class=class(a))` lists all the methods to handle objects of class `a`.

### Input and output

- `load()` load the datasets written with `save`
- `data(x)` loads specified data sets
- `library(x)` load add-on packages
- `read.table(file)` reads a file in table format and creates a data frame from it; the default separator `sep=""` is any whitespace; use `header=TRUE` to read the first line as a header of column names; use `as.is=TRUE` to

- prevent character vectors from being converted to factors; use `comment.char=""` to prevent `"#"` from being interpreted as a comment; use `skip=n` to skip `n` lines before reading data; see the help for options on row naming, NA treatment, and others
- `read.csv("filename",header=TRUE)` id. but with defaults set for reading comma-delimited files
- `read.delim("filename",header=TRUE)` id. but with defaults set for reading tab-delimited files
- `read.fwf(file,widths,header=FALSE,sep=" ",as.is=FALSE)` read a table of *fixed width formatted* data into a 'data.frame'; `widths` is an integer vector, giving the widths of the fixed-width fields
- `save(file,...)` saves the specified objects (...) in the XDR platform-independent binary format
- `save.image(file)` saves all objects
- `cat(..., file="", sep=" ")` prints the arguments after coercing to character; `sep` is the character separator between arguments
- `print(a, ...)` prints its arguments; generic, meaning it can have different methods for different objects
- `format(x,...)` format an R object for pretty printing
- `write.table(x,file="",row.names=TRUE,col.names=TRUE, sep=" ")` prints `x` after converting to a data frame; if `quote` is `TRUE`, character or factor columns are surrounded by quotes (""); `sep` is the field separator; `eol` is the end-of-line separator; `na` is the string for missing values;

use `col.names=NA` to add a blank column header to get the column headers aligned correctly for spreadsheet input  
`sink(file)` output to `file`, until `sink()`  
 Most of the I/O functions have a `file` argument. This can often be a character string naming a file or a connection. `file=""` means the standard input or output. Connections can include files, pipes, zipped files, and R variables.

On windows, the file connection can also be used with `description = "clipboard"`. To read a table copied from Excel, use

```
x <- read.delim("clipboard")
```

To write a table to the clipboard for Excel, use

```
write.table(x,"clipboard",sep="\t",col.names=NA)
```

For database interaction, see packages `RODBC`, `DBI`, `RMySQL`, `RPgSQL`, and `ROracle`. See packages `XML`, `hdf5`, `netCDF` for reading other file formats.

## Data creation

`c(...)` generic function to combine arguments with the default forming a vector; with `recursive=TRUE` descends through lists combining all elements into one vector  
`from:to` generates a sequence; “:” has operator priority; `1:4 + 1` is “2,3,4,5”

`seq(from,to)` generates a sequence `by=` specifies increment; `length=` specifies desired length  
`seq(along=x)` generates 1, 2, ..., `length(along)`; useful for `for` loops

`rep(x,times)` replicate `x` `times`; use `each=` to repeat “each” element of `x` `each` times;  
`rep(c(1,2,3),2)` is 1 2 3 1 2 3;  
`rep(c(1,2,3),each=2)` is 1 1 2 2 3 3

`data.frame(...)` create a data frame of the named or unnamed arguments;  
`data.frame(v=1:4,ch=c("a","B","c","d"),n=10)`;  
 shorter vectors are recycled to the length of the longest

`list(...)` create a list of the named or unnamed arguments;  
`list(a=c(1,2),b="hi",c=3i)`;

`array(x,dim=)` array with data `x`; specify dimensions like `dim=c(3,4,2)`; elements of `x` recycle if `x` is not long enough

`matrix(x,nrow=,ncol=)` matrix; elements of `x` recycle

`factor(x,levels=)` encodes a vector `x` as a factor

`gl(n,k,length=n*k,labels=1:n)` generate levels (factors) by specifying the pattern of their levels; `k` is the number of levels, and `n` is the number of replications

`expand.grid()` a data frame from all combinations of the supplied vectors or factors

`rbind(...)` combine arguments by rows for matrices, data frames, and others  
`cbind(...)` id. by columns

## Slicing and extracting data

Indexing vectors

<code>x[n]</code>	$n^{th}$ element
<code>x[-n]</code>	all <i>but</i> the $n^{th}$ element
<code>x[1:n]</code>	first <code>n</code> elements
<code>x[-(1:n)]</code>	elements from <code>n+1</code> to the end
<code>x[c(1,4,2)]</code>	specific elements
<code>x["name"]</code>	element named " <code>name</code> "
<code>x[x &gt; 3]</code>	all elements greater than 3
<code>x[x &gt; 3 &amp; x &lt; 5]</code>	all elements between 3 and 5
<code>x[x %in% c("a","and","the")]</code>	elements in the given set

Indexing lists

<code>x[n]</code>	list with elements <code>n</code>
<code>x[[n]]</code>	$n^{th}$ element of the list
<code>x[["name"]]</code>	element of the list named " <code>name</code> "
<code>x\$name</code>	id.

Indexing matrices

<code>x[i,j]</code>	element at row <code>i</code> , column <code>j</code>
<code>x[i,]</code>	row <code>i</code>
<code>x[,j]</code>	column <code>j</code>
<code>x[,c(1,3)]</code>	columns 1 and 3
<code>x["name",]</code>	row named " <code>name</code> "

Indexing data frames (matrix indexing plus the following)

<code>x[["name"]]</code>	column named " <code>name</code> "
<code>x\$name</code>	id.

## Variable conversion

```
as.array(x), as.data.frame(x),
as.numeric(x), as.logical(x),
as.complex(x), as.character(x), ...
convert type; for a complete list, use
methods(as)
```

## Variable information

```
is.na(x), is.null(x), is.array(x),
is.data.frame(x), is.numeric(x),
is.complex(x), is.character(x), ...
test for type; for a complete list, use
methods(is)
length(x) number of elements in x
dim(x) Retrieve or set the dimension of an
object; dim(x) <- c(3,2)
dimnames(x) Retrieve or set the dimension
names of an object
nrow(x) number of rows; NROW(x) is the same
but treats a vector as a one-row matrix
ncol(x) and NCOL(x) id. for columns
class(x) get or set the class of x; class(x) <-
"myclass"
unclass(x) remove the class attribute of x
```

`attr(x,which)` get or set the attribute `which` of `x`  
`attributes(obj)` get or set the list of attributes of `obj`

## Data selection and manipulation

`which.max(x)` returns the index of the greatest element of `x`  
`which.min(x)` returns the index of the smallest element of `x`  
`rev(x)` reverses the elements of `x`  
`sort(x)` sorts the elements of `x` in increasing order; to sort in decreasing order: `rev(sort(x))`  
`cut(x,breaks)` divides `x` into intervals (factors); `breaks` is the number of cut intervals or a vector of cut points  
`match(x, y)` returns a vector of the same length than `x` with the elements of `x` which are in `y` (NA otherwise)  
`which(x == a)` returns a vector of the indices of `x` if the comparison operation is true (TRUE), in this example the values of `i` for which `x[i] == a` (the argument of this function must be a variable of mode logical)  
`choose(n, k)` computes the combinations of `k` events among `n` repetitions =  $n! / [(n - k)!k!]$   
`na.omit(x)` suppresses the observations with missing data (NA) (suppresses the corresponding line if `x` is a matrix or a data frame)  
`na.fail(x)` returns an error message if `x` contains at least one NA  
`unique(x)` if `x` is a vector or a data frame, returns a similar object but with the duplicate elements suppressed  
`table(x)` returns a table with the numbers of the different values of `x` (typically for integers or factors)  
`subset(x, ...)` returns a selection of `x` with respect to criteria (...), typically comparisons: `x$V1 < 10`; if `x` is a data frame, the option `select` gives the variables to be kept or dropped using a minus sign  
`sample(x, size)` resample randomly and without replacement `size` elements in the vector `x`, the option `replace = TRUE` allows to resample with replacement  
`prop.table(x,margin=)` table entries as fraction of marginal table

## Math

`sin, cos, tan, asin, acos, atan, atan2, log, log10, exp`  
`max(x)` maximum of the elements of `x`  
`min(x)` minimum of the elements of `x`  
`range(x)` id. then `c(min(x), max(x))`

`sum(x)` sum of the elements of `x`  
`diff(x)` lagged and iterated differences of vector `x`  
`prod(x)` product of the elements of `x`  
`mean(x)` mean of the elements of `x`  
`median(x)` median of the elements of `x`  
`quantile(x,probs=)` sample quantiles corresponding to the given probabilities (defaults to 0,.25,.5,.75,1)  
`weighted.mean(x, w)` mean of `x` with weights `w`  
`rank(x)` ranks of the elements of `x`  
`var(x)` or `cov(x)` variance of the elements of `x` (calculated on  $n - 1$ ); if `x` is a matrix or a data frame, the variance-covariance matrix is calculated  
`sd(x)` standard deviation of `x`  
`cor(x)` correlation matrix of `x` if it is a matrix or a data frame (1 if `x` is a vector)  
`var(x, y)` or `cov(x, y)` covariance between `x` and `y`, or between the columns of `x` and those of `y` if they are matrices or data frames  
`cor(x, y)` linear correlation between `x` and `y`, or correlation matrix if they are matrices or data frames  
`round(x, n)` rounds the elements of `x` to `n` decimals  
`log(x, base)` computes the logarithm of `x` with base `base`  
`scale(x)` if `x` is a matrix, centers and reduces the data; to center only use the option `center=FALSE`, to reduce only `scale=FALSE` (by default `center=TRUE, scale=TRUE`)  
`pmin(x,y,...)` a vector which `i`th element is the minimum of `x[i]`, `y[i]`, ...  
`pmax(x,y,...)` id. for the maximum  
`cumsum(x)` a vector which `i`th element is the sum from `x[1]` to `x[i]`  
`cumprod(x)` id. for the product  
`cummin(x)` id. for the minimum  
`cummax(x)` id. for the maximum  
`union(x,y), intersect(x,y), setdiff(x,y), setequal(x,y), is.element(el,set)` "set" functions  
`Re(x)` real part of a complex number  
`Im(x)` imaginary part  
`Mod(x)` modulus; `abs(x)` is the same  
`Arg(x)` angle in radians of the complex number  
`Conj(x)` complex conjugate  
`convolve(x,y)` compute the several kinds of convolutions of two sequences  
`fft(x)` Fast Fourier Transform of an array  
`mvfft(x)` FFT of each column of a matrix  
`filter(x,filter)` applies linear filtering to a univariate time series or to each series separately of a multivariate time series  
Many math functions have a logical parameter `na.rm=FALSE` to specify missing data (NA) removal.



## Matrices

`t(x)` transpose  
`diag(x)` diagonal  
`%%` matrix multiplication  
`solve(a,b)` solves  $a \%*\% x = b$  for  $x$   
`solve(a)` matrix inverse of  $a$   
`rowsum(x)` sum of rows for a matrix-like object;  
     `rowSums(x)` is a faster version  
`colsum(x)`, `colSums(x)` id. for columns  
`rowMeans(x)` fast version of row means  
`colMeans(x)` id. for columns

## Advanced data processing

`apply(X, INDEX, FUN=)` a vector or array or list  
     of values obtained by applying a function  
     `FUN` to margins (`INDEX`) of  $X$   
`lapply(X, FUN)` apply `FUN` to each element of  
     the list  $X$   
`tapply(X, INDEX, FUN=)` apply `FUN` to each cell  
     of a ragged array given by  $X$  with indexes  
     `INDEX`  
`by(data, INDEX, FUN)` apply `FUN` to data frame  
     data subsetted by `INDEX`  
`merge(a,b)` merge two data frames by common  
     columns or row names  
`xtabs(a b, data=x)` a contingency table from  
     cross-classifying factors  
`aggregate(x, by, FUN)` splits the data frame  $x$   
     into subsets, computes summary statistics  
     for each, and returns the result in a  
     convenient form; `by` is a list of grouping  
     elements, each as long as the variables in  $x$   
`stack(x, ...)` transform data available as  
     separate columns in a data frame or list into  
     a single column  
`unstack(x, ...)` inverse of `stack()`  
`reshape(x, ...)` reshapes a data frame  
     between 'wide' format with repeated  
     measurements in separate columns of the  
     same record and 'long' format with the  
     repeated measurements in separate records;  
     use (`direction="wide"`) or  
     (`direction="long"`)

## Strings

`paste(...)` concatenate vectors after  
     converting to character; `sep=` is the string to  
     separate terms (a single space is the  
     default); `collapse=` is an optional string to  
     separate "collapsed" results  
`substr(x, start, stop)` substrings in a  
     character vector; can also assign, as  
     `substr(x, start, stop) <- value`

`strsplit(x, split)` split  $x$  according to the  
     substring `split`  
`grep(pattern, x)` searches for matches to  
     `pattern` within  $x$ ; see `?regex`  
`gsub(pattern, replacement, x)` replacement of  
     matches determined by regular expression  
     matching `sub()` is the same but only  
     replaces the first occurrence.  
`tolower(x)` convert to lowercase  
`toupper(x)` convert to uppercase  
`match(x, table)` a vector of the positions of first  
     matches for the elements of  $x$  among `table`  
`x %in% table` id. but returns a logical vector  
`pmatch(x, table)` partial matches for the  
     elements of  $x$  among `table`  
`nchar(x)` number of characters

## Dates and Times

The class `Date` has dates without times.  
`POSIXct` has dates and times, including time  
     zones. Comparisons (e.g. `>`), `seq()`, and  
`difftime()` are useful. `Date` also allows `+` and `-`.  
`?DateTimeClasses` gives more information. See  
     also package `chron`. `as.Date(s)` and  
`as.POSIXct(s)` convert to the respective class.  
`format(dt)` converts to a string representation.  
     The default string format is "2001-02-21". These  
     accept a second argument to specify a format for  
     conversion. Some common formats are:

`%a`, `%A` Abbreviated and full weekday name.  
`%b`, `%B` Abbreviated and full month name.  
`%d` Day of the month (01–31).  
`%H` Hours (00–23).  
`%I` Hours (01–12).  
`%j` Day of year (001–366).  
`%m` Month (01–12).  
`%M` Minute (00–59).  
`%p` AM/PM indicator.  
`%S` Second as decimal number (00–61).  
`%U` Week (00–53); the first Sunday as day 1 of  
     week 1.  
`%w` Weekday (0–6, Sunday is 0).  
`%W` Week (00–53); the first Monday as day 1 of  
     week 1.  
`%y` Year without century (00–99). Don't use.  
`%Y` Year with century.  
`%z` (output only.) Offset from Greenwich; -0800  
     is 8 hours west of.  
`%Z` (output only.) Time zone as a character string  
     (empty if not available).

Where leading zeros are shown they will be used  
     on output but are optional on input. See  
`?strftime`.

## Plotting

**plot(x)** plot of the values of **x** (on the *y*-axis) ordered on the *x*-axis

**plot(x, y)** bivariate plot of **x** (on the *x*-axis) and **y** (on the *y*-axis)

**hist(x)** histogram of the frequencies of **x**

**barplot(x)** histogram of the values of **x**; use **horiz=FALSE** for horizontal bars

**dotplot(x)** if **x** is a data frame, plots a Cleveland dot plot (stacked plots line-by-line and column-by-column)

**piechart(x)** circular pie-chart

**boxplot(x)** “box-and-whiskers” plot

**sunflowerplot(x, y)** id. than **plot()** but the points with similar coordinates are drawn as flowers which petal number represents the number of points

**stripplot(x)** plot of the values of **x** on a line (an alternative to **boxplot()** for small sample sizes)

**coplot(x~y | z)** bivariate plot of **x** and **y** for each value or interval of values of **z**

**interaction.plot(f1, f2, y)** if **f1** and **f2** are factors, plots the means of **y** (on the *y*-axis) with respect to the values of **f1** (on the *x*-axis) and of **f2** (different curves); the option **fun** allows to choose the summary statistic of **y** (by default **fun=mean**)

**matplot(x,y)** bivariate plot of the first column of **x** *vs.* the first one of **y**, the second one of **x** *vs.* the second one of **y**, etc.

**fourfoldplot(x)** visualizes, with quarters of circles, the association between two dichotomous variables for different populations (**x** must be an array with **dim=c(2, 2, k)**, or a matrix with **dim=c(2, 2)** if **k = 1**)

**assocplot(x)** Cohen–Friendly graph showing the deviations from independence of rows and columns in a two dimensional contingency table

**mosaicplot(x)** ‘mosaic’ graph of the residuals from a log-linear regression of a contingency table. Also useful for graphical display of contingency tables.

**pairs(x)** if **x** is a matrix or a data frame, draws all possible bivariate plots between the columns of **x**

**plot.ts(x)** if **x** is an object of class “**ts**”, plot of **x** with respect to time, **x** may be multivariate but the series must have the same frequency and dates

**ts.plot(x)** id. but if **x** is multivariate the series may have different dates and must have the same frequency

**qqnorm(x)** quantiles of **x** with respect to the values expected under a normal law

**qqplot(x, y)** quantiles of **y** with respect to the quantiles of **x**

**contour(x, y, z)** contour plot (data are interpolated to draw the curves), **x** and **y** must be vectors and **z** must be a matrix so that **dim(z)=c(length(x), length(y))** (**x** and **y** may be omitted)

**filled.contour(x, y, z)** id. but the areas between the contours are coloured, and a legend of the colours is drawn as well

**image(x, y, z)** id. but with colours (actual data are plotted)

**persp(x, y, z)** id. but in perspective (actual data are plotted)

**stars(x)** if **x** is a matrix or a data frame, draws a graph with segments or a star where each row of **x** is represented by a star and the columns are the lengths of the segments

**symbols(x, y, ...)** draws, at the coordinates given by **x** and **y**, symbols (circles, squares, rectangles, stars, thermometres or “boxplots”) which sizes, colours ... are specified by supplementary arguments

**termplot(mod.obj)** plot of the (partial) effects of a regression model (**mod.obj**)

The following parameters are common to many plotting functions:

**add=FALSE** if **TRUE** superposes the plot on the previous one (if it exists)

**axes=TRUE** if **FALSE** does not draw the axes and the box

**type="p"** specifies the type of plot, “**p**”: points, “**l**”: lines, “**b**”: points connected by lines, “**o**”: id. but the lines are over the points, “**h**”: vertical lines, “**s**”: steps, the data are represented by the top of the vertical lines, “**S**”: id. but the data are represented by the bottom of the vertical lines

**xlim=, ylim=** specifies the lower and upper limits of the axes, for example with **xlim=c(1, 10)** or **xlim=range(x)**

**xlab=, ylab=** annotates the axes, must be variables of mode character

**main=** main title, must be a variable of mode character

**sub=** sub-title (written in a smaller font)

## Low-level plotting commands

**points(x, y)** adds points (the option **type=** can be used)

**lines(x, y)** id. but with lines

**text(x, y, labels, ...)** adds text given by **labels** at coordinates (**x,y**); a typical use is: **plot(x, y, type="n"); text(x, y, names)**

**mtext(text, side=3, line=0, ...)** adds text given by **text** in the margin specified by **side** (see **axis()** below); **line** specifies the line from the plotting area

**segments**(x0, y0, x1, y1) draws lines from points (x0,y0) to points (x1,y1)

**arrows**(x0, y0, x1, y1, angle= 30, code=2) id. with arrows at points (x0,y0) if code=2, at points (x1,y1) if code=1, or both if code=3; angle controls the angle from the shaft of the arrow to the edge of the arrow head

**abline**(a,b) draws a line of slope b and intercept a

**abline**(h=y) draws a horizontal line at ordinate y

**abline**(v=x) draws a vertical line at abscissa x

**abline**(lm.obj) draws the regression line given by lm.obj

**rect**(x1, y1, x2, y2) draws a rectangle which left, right, bottom, and top limits are x1, x2, y1, and y2, respectively

**polygon**(x, y) draws a polygon linking the points with coordinates given by x and y

**legend**(x, y, legend) adds the legend at the point (x,y) with the symbols given by legend

**title**() adds a title and optionally a sub-title

**axis**(side, vect) adds an axis at the bottom (side=1), on the left (2), at the top (3), or on the right (4); vect (optional) gives the abscissa (or ordinates) where tick-marks are drawn

**rug**(x) draws the data x on the x-axis as small vertical lines

**locator**(n, type="n", ...) returns the coordinates (x,y) after the user has clicked n times on the plot with the mouse; also draws symbols (type="p") or lines (type="l") with respect to optional graphic parameters (...); by default nothing is drawn (type="n")

## Graphical parameters

These can be set globally with **par**(...); many can be passed as parameters to plotting commands.

**adj** controls text justification (0 left-justified, 0.5 centred, 1 right-justified)

**bg** specifies the colour of the background (ex. : **bg**="red", **bg**="blue", ... the list of the 657 available colours is displayed with **colors**())

**bty** controls the type of box drawn around the plot, allowed values are: "o", "l", "7", "c", "u" ou "]" (the box looks like the corresponding character); if **bty**="n" the box is not drawn

**cex** a value controlling the size of texts and symbols with respect to the default; the following parameters have the same control for numbers on the axes, **cex.axis**, the axis labels, **cex.lab**, the title, **cex.main**, and the sub-title, **cex.sub**

**col** controls the color of symbols and lines; use color names: "red", "blue" see **colors**() or as "#RRGGBB"; see **rgb**(), **hsv**(), **gray**(), and **rainbow**(); as for **cex** there are: **col.axis**, **col.lab**, **col.main**, **col.sub**

**font** an integer which controls the style of text (1: normal, 2: italics, 3: bold, 4: bold italics); as for **cex** there are: **font.axis**, **font.lab**, **font.main**, **font.sub**

**las** an integer which controls the orientation of the axis labels (0: parallel to the axes, 1: horizontal, 2: perpendicular to the axes, 3: vertical)

**lty** controls the type of lines, can be an integer or string (1: "solid", 2: "dashed", 3: "dotted", 4: "dotdash", 5: "longdash", 6: "twodash", or a string of up to eight characters (between "0" and "9") which specifies alternatively the length, in points or pixels, of the drawn elements and the blanks, for example **lty**="44" will have the same effect than **lty**=2

**lwd** a numeric which controls the width of lines, default 1

**mar** a vector of 4 numeric values which control the space between the axes and the border of the graph of the form **c**(bottom, left, top, right), the default values are **c**(5.1, 4.1, 4.1, 2.1)

**mfc** a vector of the form **c**(nr,nc) which partitions the graphic window as a matrix of nr lines and nc columns, the plots are then drawn in columns

**mfrow** id. but the plots are drawn by row

**pch** controls the type of symbol, either an integer between 1 and 25, or a single character in "":

1: ○ 2: △ 3: + 4: × 5: ◇ 6: ▽ 7: ☒ 8: ✱ 9: ⬠  
 10: ⊕ 11: ⊗ 12: ⊞ 13: ⊠ 14: ⊡ 15: ■ 16: ● 17: ▲ 18: ◆  
 19: ● 20: ● 21: ○ 22: □ 23: ◇ 24: △ 25: ▽ \*: \* ∴ ∴

**ps** an integer which controls the size in points of texts and symbols

**pty** a character which specifies the type of the plotting region, "s": square, "m": maximal

**tck** a value which specifies the length of tick-marks on the axes as a fraction of the smallest of the width or height of the plot; if **tck**=1 a grid is drawn

**tcl** a value which specifies the length of tick-marks on the axes as a fraction of the height of a line of text (by default **tcl**=-0.5)

**xaxt** if **xaxt="n"** the *x*-axis is set but not drawn (useful in conjunction with **axis(side=1, ...)**)  
**yaxt** if **yaxt="n"** the *y*-axis is set but not drawn (useful in conjunction with **axis(side=2, ...)**)

## Lattice (Trellis) graphics

**barchart(y~x)** histogram of the values of *y* with respect to those of *x*  
**bwplot(y~x)** “box-and-whiskers” plot  
**densityplot(~x)** density functions plot  
**dotplot(y~x)** Cleveland dot plot (stacked plots line-by-line and column-by-column)  
**histogram(~x)** histogram of the frequencies of *x*  
**qqmath(~x)** quantiles of *x* with respect to the values expected under a theoretical distribution  
**stripplot(y~x)** single dimension plot, *x* must be numeric, *y* may be a factor  
**qq(y~x)** quantiles to compare two distributions, *x* must be numeric, *y* may be numeric, character, or factor but must have two ‘levels’  
**xyplot(y~x)** bivariate plots (with many functionalities)  
**levelplot(z~x\*y)** coloured plot of the values of *z* at the coordinates given by *x* and *y* (*x*, *y* and *z* are all of the same length)  
**splom(~x)** matrix of bivariate plots  
**parallel(~x)** parallel coordinates plot

## Optimization and model fitting

**optim(par, fn, method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"))**  
 general-purpose optimization; **par** is initial values, **fn** is function to optimize (normally minimize)  
**nlm(f,p)** minimize function *f* using a Newton-type algorithm with starting values *p*  
**lm(formula)** fit linear models; **formula** is typically of the form **response termA + termB + ...**; use **I(x\*y) + I(x^2)** for terms made of nonlinear components  
**glm(formula,family=)** fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution; **family** is a description of the error distribution and link function to be used in the model; see **?family**  
**nlm(formula)** nonlinear least-squares estimates of the nonlinear model parameters

**approx(x,y=)** linearly interpolate given data points; *x* can be an *xy* plotting structure  
**spline(x,y=)** cubic spline interpolation  
**loess(formula)** fit a polynomial surface using local fitting

Many of the formula-based modeling functions have several common arguments: **data=** the data frame for the formula variables, **subset=** a subset of variables used in the fit, **na.action=** action for missing values: **"na.fail"**, **"na.omit"**, or a function. The following generics often apply to model fitting functions:

**predict(fit,...)** predictions from **fit** based on input data  
**df.residual(fit)** returns the number of residual degrees of freedom  
**coef(fit)** returns the estimated coefficients (sometimes with their standard-errors)  
**residuals(fit)** returns the residuals  
**deviance(fit)** returns the deviance  
**fitted(fit)** returns the fitted values  
**logLik(fit)** computes the logarithm of the likelihood and the number of parameters  
**AIC(fit)** computes the Akaike information criterion or AIC

## Statistics

**aov(formula)** analysis of variance model  
**anova(fit,...)** analysis of variance (or deviance) tables for one or more fitted model objects  
**density(x)** kernel density estimates of *x*  
**binom.test()**, **pairwise.t.test()**, **power.t.test()**, **prop.test()**, **t.test()**, ...  
 use **help.search("test")**

## Distributions

**rnorm(n, mean=0, sd=1)** Gaussian (normal)  
**rexp(n, rate=1)** exponential  
**rgamma(n, shape, scale=1)** gamma  
**rpois(n, lambda)** Poisson  
**rweibull(n, shape, scale=1)** Weibull  
**rcauchy(n, location=0, scale=1)** Cauchy  
**rbeta(n, shape1, shape2)** beta  
**rt(n, df)** ‘Student’ (*t*)  
**rf(n, df1, df2)** Fisher–Snedecor (*F*) ( $\chi^2$ )  
**rchisq(n, df)** Pearson  
**rbinom(n, size, prob)** binomial  
**rgeom(n, prob)** geometric  
**rhyper(nn, m, n, k)** hypergeometric  
**rlogis(n, location=0, scale=1)** logistic  
**rlnorm(n, meanlog=0, sdlog=1)** lognormal  
**rnbinom(n, size, prob)** negative binomial  
**runif(n, min=0, max=1)** uniform  
**rwilcox(nn, m, n), rsignrank(nn, n)**  
 Wilcoxon’s statistics

All these functions can be used by replacing the letter **r** with **d**, **p** or **q** to get, respectively, the probability density (**dfunc**(**x**, ...)), the cumulative probability density (**pfunc**(**x**, ...)), and the value of quantile (**qfunc**(**p**, ...), with  $0 < p < 1$ ).

## Programming

```
function( arglist ) expr function definition
return(value)
if(cond) expr
if(cond) cons.expr else alt.expr
for(var in seq) expr
while(cond) expr
repeat expr
break
next
Use braces {} around statements
ifelse(test, yes, no) a value with the same
  shape as test filled with elements from
  either yes or no
do.call(funname, args) executes a function
  call from the name of the function and a list
  of arguments to be passed to it.
```

## The Epi package

```
Lexis.diagram() Draw a Lexis diagram,
  optionally with life lines.
Lexis.lines() Add lines to a Lexis diagram.
rateplot(rates,...) Make plots of rates from
  an age by period table.
cal.yr(x,format) Convert x to fractional
  calendar year.
stat.table(index,contents,...) Make
  tables, classified by index, of sums, ratios
  etc. given in contents.
ci.lin(obj,ctr.mat,subset,diffs,Exp)
  Extract parameters and linear functions of
  them from model objects.
plotEst(ests,...) Make a plot of parameter
  estimates.
twoby2(exposure,outcome,...) Analysis of a
   $2 \times 2$  table. Input can be either two binary
  variables or a matrix of counts.
```