AN INTRODUCTION TO R

DEEPAYAN SARKAR

Introduction and examples

What is R?.

> R provides an environment in which you can perform statistical analysis and produce graphics. It is actually a complete programming language, although that is only marginally described in this book.
>
> —Peter Dalgaard, *"Introductory Statistics with R"*, 2002

R can be viewed as a programming language that happens to come with a large library of pre-defined functions that can be used to perform various tasks. A major focus of these pre-defined functions is statistical data analysis, and these allow R to be used purely as a toolbox for standard statistical techniques. However, some knowledge of R programming is essential to use it well at any level. For advanced users in particular, the main appeal of R (as opposed to other data analysis software) is as a programming environment suited to data analysis. Our goal will be to learn R as a statistics toolbox, but with a fairly strong emphasis on its programming language aspects.

In this tutorial, we will do some elementary statistics, learn to use the documentation system, and learn about common data structures and programming features in R. Some follow-up tutorials are available for self-study at http://www.isid.ac.in/~deepayan/R-tutorials/.

For more resources, see the R Project homepage http://www.r-project.org, which links to various manuals and other user-contributed documentation. A list of books related to R is available at http://www.r-project.org/doc/bib/R-jabref.html. An excellent introductory book is Peter Dalgaard, *"Introductory Statistics with R"*, Springer (2002).

Interacting with R. Unlike languages like C, Fortran, or Java, R is an *interactive* programming langauge. This means that R works interactively, using a question-and-answer model:

- Start R
- Type a command and press `Enter`
- R executes this command (often printing the result)
- R then waits for more input
- Type `q()` to exit

Here are some simple examples:

```
> 2 + 2
[1] 4
> exp(-2) ## exponential function
[1] 0.1353353
> log(100, base = 10)
[1] 2
> runif(10)
```

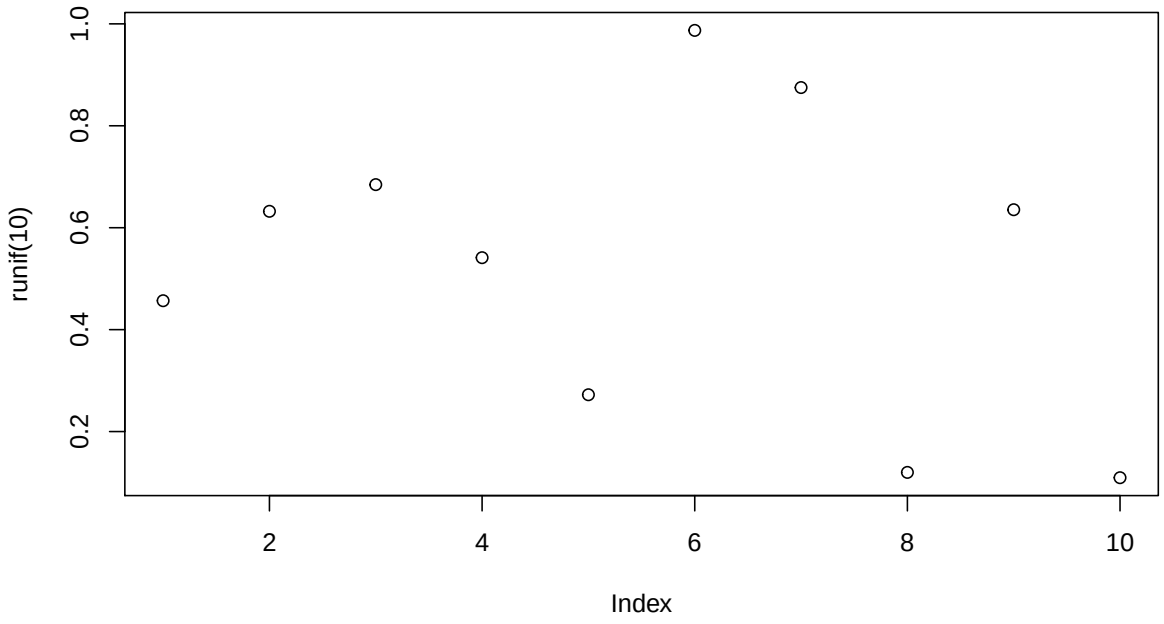
Date: December 2013.

```
 [1] 0.6188782 0.2595751 0.5500394 0.3630359 0.5308452 0.6490918 0.8871383
 [8] 0.8335538 0.4570793 0.5657994
```

The last command generates ten $U(0,1)$ random variables; the result (which is printed) is a vector of 10 numbers. `exp()`, `log()`, and `runif()` are *functions*.

R features very strong graphics capabilities. The simplest way to produce plots is to use the `plot()` function. We will learn more about graphics in a different tutorial.

```
> plot(runif(10))
```



Variables. As in other programming languages, R has *symbolic variables* which can be assigned values. Assignment is done using the `'<-'` operator. The more `C`-like `'='` also works (with some exceptions).

```
> x <- 2
> x + x
[1] 4
> yVar2 = x + 3
> yVar2
[1] 5
> s <- "this is a character string"
> s
[1] "this is a character string"
```

Variable names can be almost anything, but they should not start with a digit, and should not contain spaces. Names are case-sensitive. Some common names are already used by R (`c, q, t, C, D, F, I, T`) and should be avoided.

Vectorized arithmetic. The elementary data types in R are all vectors; even the scalar variables we defined above are stored as vectors of length one. The `c(...)` construct can be used to create vectors:

```
> weight <- c(60, 72, 57, 90, 95, 72)
> weight
[1] 60 72 57 90 95 72
```

To generate a vector of regularly spaced numbers, use

```
> seq(0, 1, length = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

The `c()` function can be used to combine vectors as well as scalars, e.g.,

```
> x <- seq(0, 1, length = 6)
> c(x, 1:10, 100)
 [1]   0.0   0.2   0.4   0.6   0.8   1.0   1.0   2.0   3.0   4.0   5.0   6.0
[13]   7.0   8.0   9.0  10.0 100.0
```

Common arithmetic operations (including `+, -, *, /, ^`) and mathematical functions (e.g. `sin()`, `cos()`, `log()`) work *element-wise* on vectors, and produce another vector:

```
> height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
> height^2
[1] 3.0625 3.2400 2.7225 3.6100 3.0276 3.6481
> bmi <- weight / height^2
> bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
> log(bmi)
[1] 2.975113 3.101093 3.041501 3.216102 3.446107 2.982460
```

When two vectors are not of equal length, the shorter one is *recycled*. For example, The following adds 0 to all the odd elements and 2 to all the even elements of `1:10`:

```
> 1:10 + c(0, 2)
 [1]  1  4  3  6  5  8  7 10  9 12
```

These simple vectorized arithmetic rules are among the most powerful design features of the R language.

Scalars from Vectors. Many functions summarize a data vector by producing a scalar from a vector, e.g.,

```
> sum(weight)
[1] 446
> length(weight)
[1] 6
> avg.weight <- mean(weight)
> avg.weight
[1] 74.33333
```

Simple descriptive statistics. Simple summary statistics (mean, median, s.d., variance) can be computed from numeric vectors using appropriately named functions:

```
> x <- rnorm(100)
> mean(x)
[1] -0.05483579
> sd(x)
```

```
[1] 1.00529
> var(x)
[1] 1.010607
> median(x)
[1] -0.03847274
```

Quantiles can be computed using the `quantile()` function. `IQR()` computes the inter-quartile range.

```
> xquants <- quantile(x)
> xquants
         0%         25%         50%         75%        100%
-2.40368508 -0.64198753 -0.03847274  0.58305784  2.76789106
> xquants[4] - xquants[2]
     75%
1.225045
> IQR(x)
[1] 1.225045
> quantile(x, probs = c(0.2, 0.4, 0.6, 0.8))
       20%        40%        60%        80%
-0.7865878 -0.2426701  0.1273250  0.6996438
```

The so-called five-number summary (minimum, maximum, and quartiles) is given by `fivenum()`. A slightly extended summary is given by `summary()`.

```
> fivenum(x)
[1] -2.40368508 -0.64850725 -0.03847274  0.59636251  2.76789106
> summary(x)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.40400 -0.64200 -0.03847 -0.05484  0.58310  2.76800
```

Object-oriented progamming: classes and methods. `summary()` produces a nice summary for numeric vectors. However, not all data is numeric in nature, and we may want a summary of non-numeric variables as well, for which this kind of output would not be appropriate. To illustrate how R handles this problem, we will make use of a real dataset. Data can be read into R from a file (we will soon see how), but we can also use one of the many datasets built into R. One such dataset is the well-known Iris data. The dataset contains measurements on 150 flowers, 50 each from 3 species: *Iris setosa*, *versicolor* and *virginica*. It is typically used to illustrate the problem of *classification*— given the four measurements for a new flower, can we predict its Species?

Like most datasets, iris is not a simple vector, but a composite "data frame" object made up of several component vectors. We can think of a data frame as a matrix-like object, with each row representing an observational unit (in this case, a flower), and columns representing multiple measurements made on the unit. The `head()` function extracts the first few rows, and the `$` operator extracts individual components.

```
> head(iris) # The first few rows
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
> iris$Sepal.Length
```

```
  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
 [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
 [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
 [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
 [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
 [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
[109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
[127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
[145] 6.7 6.7 6.3 6.5 6.2 5.9
```

A more concise description is given by the `str()` function (short for "structure").

```
> str(iris)
'data.frame':        150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

As we can see, the first four components of iris are numeric vectors, but the last is a "factor". These are how R represents categorical variables.

Let us now see the effect of calling `summary()` for different types of objects.

```
> summary(iris$Sepal.Length)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.300   5.100   5.800   5.843   6.400   7.900
> summary(iris$Species)
    setosa versicolor  virginica
        50         50         50
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

Note the different formats of the output. `Species` is summarized differently (by the frequency distribution of its values) because it is a categorical variable, for which mean or quantiles are meaningless. The entire data frame iris is summarized by combining the summaries of all its components.

R achieves this kind of object-specific customized output through a fairly simple object-oriented paradigm. Each R object has a *class* ("numeric", "factor", etc.). `summary()` is what is referred to as a *generic* function, with class-specific methods that handle objects of various classes. When the generic `summary()` is called, R figures out the appropriate method and calls it.

The details of what happens in the example above is revealed by the following calls.

```
> class(iris$Sepal.Length)
```

```
[1] "numeric"
> class(iris$Species)
[1] "factor"
> class(iris)
[1] "data.frame"
> methods(summary)
 [1] summary.aov             summary.aovlist*        summary.aspell*
 [4] summary.connection      summary.data.frame      summary.Date
 [7] summary.default         summary.ecdf*           summary.factor
[10] summary.glm             summary.infl*           summary.lm
[13] summary.loess*          summary.manova          summary.matrix
[16] summary.mlm*            summary.nls*            summary.packageStatus*
[19] summary.PDF_Dictionary* summary.PDF_Stream*     summary.POSIXct
[22] summary.POSIXlt         summary.ppr*            summary.prcomp*
[25] summary.princomp*       summary.proc_time       summary.srcfile
[28] summary.srcref          summary.stepfun         summary.stl*
[31] summary.table           summary.tukeysmooth*

   Non-visible functions are asterisked
```

The rules are fairly intuitive. The last call gives the list of all available methods. Objects of class "factor" are handled by `summary.factor()`, "data.frame"-s are handled by `summary.data.frame()`. There is no `summary.numeric()`, so numeric vectors are handled by `summary.default()`. We will learn more about object-oriented programming later, but we introduce the concept now because finding appropriate documentation may be dificult otherwise.

Getting help. R has too many tools for anyone to remember them all, so it is very important to know how to find relevant information using the help system. There are three important functions every R user should know about.

- `help.start()` starts a browser window with an HTML help interface. This is one of the best ways to get started; it links to a very detailed manual for beginners called *"An Introduction to R"*, as well as topic-wise listings.
- `help(topic)` displays the help page for a particular topic. Every R function has a help page.
- `help.search("search string")` performs a subject/keyword search.

The `help()` function provides topic-wise help. When you know which function you are interested in, this is usually the best way to learn how to use it. There's also a short-cut for this; use a question mark (`?`) followed by the topic. The following are equivalent.

```
> help(plot)
> ?plot
```

The help pages can be opened in a browser for easier navigation (through links to other help pages).

```
> help(plot, help_type = "html")
```

When you want to know about a subject, but don't know which particular help page has the information, the `help.search()` function (shortcut: `??`) can be useful. For example, try

```
> help.search("logarithm")
> ??logarithm
```

The help pages are usually detailed (but terse). Among other things, they often contain a description of what the function returns, a 'See Also' section that lists related help pages, and an 'Examples' section, with actual code illutrating how to use the documented functions. These examples can be run directly using the `example()` function. e.g., try

```
> example(plot)
```

Another useful tool is the `apropos()` function, which lists all functions (or other variables) whose name matches a specified character string.

```
> apropos("plot")
 [1] "assocplot"           "barplot"             "barplot.default"
 [4] "biplot"              "boxplot"             "boxplot.default"
 [7] "boxplot.matrix"      "boxplot.stats"       "cdplot"
[10] "coplot"              ".__C__recordedplot"  "fourfoldplot"
[13] "interaction.plot"    "lag.plot"            "matplot"
[16] "monthplot"           "mosaicplot"          "plot"
[19] "plot.default"        "plot.design"         "plot.ecdf"
[22] "plot.function"       "plot.new"            "plot.spec.coherency"
[25] "plot.spec.phase"     "plot.stepfun"        "plot.ts"
[28] "plot.window"         "plot.xy"             "preplot"
[31] "qqplot"              "recordPlot"          "replayPlot"
[34] "savePlot"            "screeplot"           "spineplot"
[37] "sunflowerplot"       "termplot"            "ts.plot"
```

Getting help for generic functions and methods. Getting help is not as obvious in the case of generic functions and methods. In principle, the generic function and specific methods are all considered separate topics, and may have separate help pages, although in practice a single page often documents multiple topics. For example, `summary()` and `summary.default()` are documented together, but `plot()` and `plot.default()` are documented separately. To use `plot()` effectively, it is essential to read the `?plot.default` help page. Note that even though `plot.default()` has its own help page, it should never be called directly.

Further reading. Even the "standard" functions in R are too numerous to be covered exhaustively. Thus, you should get in the habit of studying the documentation and examples for any new function you encounter. A useful initial list can be found in a one-page "R Reference Card" available at http://cran.fhcrc.org/doc/contrib/refcard.pdf. It may also be worthwhile to browse through the index of help pages in specific packages, produced by

```
> library(help = base)
> library(help = graphics)
```

etc., and read the topics that seem interesting.

Importing data. External data is typically read in into R using the `read.table()` function or one of its close relatives. For a comprehensive discussion and more advanced options, consult the *"R Data Import/Export"* Manual (type `help.start()` for a link).

Exercise 1. *Read `?read.table` and read in data from the file `"Gcsemv.txt"`. A description of the data available at*
http://www.cmm.bristol.ac.uk/learning-training/multilevel-m-software/gcsemv.shtml is reproduced here:

> *The data contain GCSE exam scores on a science subject. Two components of the exam were chosen as outcome variables: written paper and course work. There are 1,905 students from 73 schools in England. Five fields are as follows.*
>
> *Missing values are coded as -1.*
>
> 1. *School ID*
> 2. *Student ID*
> 3. *Gender of student*

0 = *boy*
1 = *girl*

4. *Total score of written paper*
5. *Total score of coursework paper*

Read in the data and give appropriate names to the columns. How should you handle the missing values? Some of the variables read in as numeric are actually categorical variables. How would you convert them to factors? Missing values and factors are discussed later in this tutorial.

Exercise 2. *The file* `"iris.xls"` *contains a copy of the* iris *data with one observation modified. Read in the data after first exporting it as a CSV file (comma-separated values) from a spreadsheet software such as Microsoft Excel, OpenOffice Calc, or Gnumeric. Determine which value has been modified.*

Packages. R makes use of a system of *packages* that enables the addition of new functionality. Each package is a collection of functions (and data) with a common theme; the core of R itself is a package called base. A collection of packages is called a *library*. Some packages are already attached when R starts up. Other packages need be attached using the `library()` function. It is fairly easy for anyone to write new R packages. This is one of the attractions of R over other statistical software.

Several packages come pre-installed with R.

```
> ip <- installed.packages()
> rownames(ip)[ip[, "Priority"] %in% c("base", "recommended")]
 [1] "base"       "boot"       "class"      "cluster"    "codetools"
 [6] "compiler"   "datasets"   "foreign"    "graphics"   "grDevices"
[11] "grid"       "KernSmooth" "lattice"    "MASS"       "Matrix"
[16] "methods"    "mgcv"       "nlme"       "nnet"       "parallel"
[21] "rpart"      "spatial"    "splines"    "stats"      "stats4"
[26] "survival"   "tcltk"      "tools"      "utils"
```

There are also many (more than 5000) other packages contributed by various users of R available online, from the Comprehensive R Archive Network (*CRAN*) at http://cran.fhcrc.org/web/packages/. The *Bioconductor* project provides an extensive collection of R packages specifically for bioinformatics at http://www.bioconductor.org/packages/release/Software.html

Some packages are already attached when R starts up. At any point, The list of currently loaded packages can be listed by the `search()` function:

```
> search()
 [1] ".GlobalEnv"        "package:tools"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

Other packages can be attached by the user. For example, the ISwR package contains datasets used in Peter Dalgaard's *"Introductory Statistics with R"*. This can be loaded by:

```
> library(ISwR)
```

New packages can be downloaded and installed using the `install.packages()` function. For example, to install the ISwR package (if it's not already installed), one can use

```
> install.packages("ISwR")
> library(help = ISwR)
```

The last call gives a list of all help pages in the package.

Session management and serialization. R has the ability to save objects, to be loaded again later. Whenever exiting, R asks to save all the variables created by the user, and restores them when starting up the next time (in the same directory).

This is actually a special case of a very powerful feature of R called *serialization*. All R objects, however complex, can be saved as a file on disk, and re-read in a later session. This allows the results of resource-intensive operations to be retained across sessions (or shared with others), avoiding the need to recompute them. See `?save` and `?load` for details.

Language Overview I

R is often referred to as a *dialect* of the S language. S was developed at the AT&T Bell Laboratories by John Chambers and his colleagues doing research in statistical computing, beginning in the late 1970's. The original S implementation is used in the commercially available software S-PLUS. R is an open source implementation developed independently (starting in the early 1990's) which is mostly similar, but with some important differences. We will only discuss the S language as implemented in R (and often refer to it as the R language).

Expressions and Objects. R works by evaluating *expressions* typed at the command prompt. Expressions involve variable references, operators, function calls, etc. Most expressions, when evaluated, produce a value, which can be either assigned to a variable (e.g. `x <- 2 + 2`), or is printed in the R session. Some expressions are useful for their "side-effects" (e.g., `plot()` produces graphics output). Evaluated expression values can be quite large, and often need to be re-used, so it is good practice to assign them to variables rather than print them directly.

Objects are anything that can be assigned to a variable. In the following example, `c(1, 2, 3, 4, 5)` is an **expression** that produces an **object**, whether or not the result is stored in a variable:

```
> sum(c(1, 2, 3, 4, 5))
[1] 15
> x <- c(1, 2, 3, 4, 5)
> sum(x)
[1] 15
```

R has several important kinds of objects; for example: *functions*, *vectors* (numeric, character, logical), *matrices*, *lists*, and *data frames*.

Functions. Most useful things in R are done by function calls. Function calls look like a name followed by some *arguments* in parentheses.

```
> plot(height, weight)
```

Apart from a special argument called `...`, all arguments have a *formal name*. When a function is evaluated, it needs to know what value has been assigned to each of its arguments. There are several ways to specify arguments:

By position: The first two arguments of the `plot()` function are `x` and `y`. So,

```
> plot(height, weight)
```

is equivalent to

```
> plot(x = height, y = weight)
```

By name: This is the safest way to match arguments, by specifying the argument names explicitly. This overrides positional matching, so it is equivalent to say

```
> plot(y = weight, x = height)
```

Formal argument names can be matched partially (we will see examples later).

With default values: Arguments will often have default values. If they are not specified in the call, these default values will be used.

```
> plot(height)
```

Functions are just like other objects in R; they can be return values in other functions, they can be assigned to variables, and they can be used as arguments in other function calls. This is often sumarized by the statement that "functions are first-class citizens" in R.

New function objects are defined using the construct `function( arglist ) expression`. Here is a simple function assigned to the variable `myfun`.

```
> myfun <- function(a = 1, b = 2, c)
      return(list(sum = a + b + c, product = a * b * c))
> myfun(6, 7, 8) # positional matching
$sum
[1] 21

$product
[1] 336
> myfun(10, c = 3) # defaults and named argument
$sum
[1] 15

$product
[1] 60
```

The arguments that a particular function accepts (along with their default values) can be listed by the `args()` function:

```
> args(myfun)
function (a = 1, b = 2, c)
NULL
> args(plot.default)
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
    ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
    panel.last = NULL, asp = NA, ...)
NULL
```

The triple-dot (...) argument indicates that the function can accept any number of further arguments. What happens to those arguments is determined by the function.

The full definition of a function can be shown by printing it (as opposed to calling it, by omitting the parentheses after the function's name); for example:

```
> myfun
function(a = 1, b = 2, c)
    return(list(sum = a + b + c, product = a * b * c))
> summary
function (object, ...)
UseMethod("summary")
<bytecode: 0x2ac2cb8>
<environment: namespace:base>
```

Exercise 3. *From the output above,* `summary()` *seems to simply call another function* `UseMethod()`*. Read the help page for* `UseMethod()` *to learn what it does (skip the technical parts). Print the definition of the function that is used when* `summary()` *is called on a "data.frame" object.*

Exercise 4. *In the output of* `methods(summary)`*, some functions were marked with an asterisk, with a note stating that "Non-visible functions are asterisked". One example of this is* `summary.ecdf`*. Find and print the definition of* `summary.ecdf`*.*

Some functions are "hidden" in this way because they are inside *namespaces*. A namespace usually consists of several functions that are all visible to each other, but only the ones that are *exported* are visible outside that namespace. Namespaces roughly correspond to packages in R. The main use for namespaces is to prevent potential confusion caused by multiple functions of the same name in different packages.

Vectors. The basic data types in R are all vectors. The simplest varieties are *numeric*, *character*, and *logical* (`TRUE` or `FALSE`):

```
> c(1, 2, 3, 4, 5)
[1] 1 2 3 4 5
> c("Spring", "Summer", "Autumn", "Winter")
[1] "Spring" "Summer" "Autumn" "Winter"
> c(TRUE, TRUE, FALSE, TRUE)
[1]  TRUE  TRUE FALSE  TRUE
> c(1, 2, 3, 4, 5) > 3
[1] FALSE FALSE FALSE  TRUE  TRUE
```

The length of any vector can be determined by the `length()` function:

```
> gt.3 <- c(1, 2, 3, 4, 5) > 3
> gt.3
[1] FALSE FALSE FALSE  TRUE  TRUE
> length(gt.3)
[1] 5
> sum(gt.3)
[1] 2
```

This happens because of *coercion* from logical to numeric.

Functions that create vectors. `seq()` creates a sequence of equidistant numbers (See `?seq`)

```
> seq(4, 10, 0.5)
 [1]  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
> seq(length = 10)
 [1]  1  2  3  4  5  6  7  8  9 10
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> args(seq.default)
function (from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
NULL
```

Partial matching. Note that the named argument `length` of the call to `seq()` actually matches the argument `length.out`.

`c()` *concatenates* one or more vectors.

```
> c(1:5, seq(10, 20, length = 6))
 [1]  1  2  3  4  5 10 12 14 16 18 20
```

`rep()` replicates a vector.

```
> rep(1:5, 2)
 [1] 1 2 3 4 5 1 2 3 4 5
> rep(1:5, length = 12)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2
> rep(c('one', 'two'), c(6, 3))
[1] "one" "one" "one" "one" "one" "one" "two" "two" "two"
```

Special values. R has several special values: `NA` Denotes a 'missing value', `NaN` denotes 'Not a Number', e.g., 0/0, `-Inf, Inf` positive and negative infinities, e.g. 1/0. Finally, `NULL` is the 'Null object', which is mostly used for programming convenience.

Exercise 5. *Here are two simple numeric vectors containing* `NA` *and* `Inf` *values.*

```
> x <- c(1:5, NA, 7:10, NULL)
> y <- c(1:5, Inf, 7:10)
```

Use R to find the mean and median of these vectors. How can you make R ignore the `NA` *values? What is the length of the* `NULL` *object?*

Exercise 6. *Find the mean of each numeric variable in the* `airquality` *dataset.*

Matrices and Arrays. Matrices (and more generally arrays of any dimension) are stored in R as a vector with a dimension attribute:

```
> x <- 1:12
> dim(x) <- c(3, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> nrow(x)
[1] 3
> ncol(x)
[1] 4
```

The fact that the left hand side of an assignment can look like a function applied to an object (rather than a variable) is a very interesting and useful feature. These are called *replacement* functions.

The same vector can be used to create a 3-dimensional array

```
> dim(x) <- c(2, 2, 3)
> x
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

Matrices can also be created conveniently by the `matrix` function. Their row and column names can be set.

```
> x <- matrix(1:12, nrow = 3, byrow = TRUE)
> rownames(x) <- LETTERS[1:3]
> x
  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
```

Matrices can be transposed by the `t()` function. General array permutation is done by `aperm()`.

```
> t(x)
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
```

Matrices do not need to be numeric, there can be character or logical matrices as well:

```
> matrix(month.name, nrow = 6)
     [,1]       [,2]
[1,] "January"  "July"
[2,] "February" "August"
[3,] "March"    "September"
[4,] "April"    "October"
[5,] "May"      "November"
[6,] "June"     "December"
```

Matrix multiplication. The multiplication operator (`*`) works element-wise, as with vectors. The matrix multiplication operator is `%*%`:

```
> x
  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
```

```
> x * x
  [,1] [,2] [,3] [,4]
A    1    4    9   16
B   25   36   49   64
C   81  100  121  144
> x %*% t(x)
    A   B   C
A  30  70 110
B  70 174 278
C 110 278 446
```

Creating matrices from vectors. The `cbind` (*column bind*) and `rbind` (*row bind*) functions can create matrices from smaller matrices or vectors.

```
> y <- cbind(A = 1:4, B = 5:8, C = 9:12)
> y
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
> rbind(y, 0)
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
[5,] 0 0  0
```

Note that the short vector (`0`) is replicated.

Factors. Factors are how R handles *categorical data* (e.g., eye color, gender, pain level). Such data are often available as numeric codes, but should be converted to factors for proper analysis. Without such conversion, R would not be able to distinguish between numeric and categorical data.

```
> pain <- c(0, 3, 2, 2, 1)
> fpain <- factor(pain, levels = 0:3)
> fpain
[1] 0 3 2 2 1
Levels: 0 1 2 3
> levels(fpain) <- c("none", "mild", "medium", "severe")
> fpain
[1] none   severe medium medium mild
Levels: none mild medium severe
> as.numeric(fpain)
[1] 1 4 3 3 2
```

Factors can be thought of as vectors with a dual nature: as descriptive character strings as well as integer codes. The last function extracts the internal representation of factors, as integer codes starting from 1. Factors can also be created from character vectors.

```
> text.pain <- c("none", "severe", "medium", "medium", "mild")
> factor(text.pain)
```

```
[1] none   severe medium medium mild
Levels: medium mild none severe
```

Note that the levels are sorted alphabetically by default, which may not be what you really want. It is often a good idea to specify the levels explicitly when creating a factor.

Exercise 7. *Use the `gl()` function to create a factor with 5 levels, and then change the levels so that two of the existing levels now have the same name. For example:*

```
> a <- gl(5, 2)
> levels(a)
[1] "1" "2" "3" "4" "5"
> levels(a) <- c("1", "2", "2", "4", "5")
```

How does the factor change? (Hint: look at the numeric codes of the result.) What happens if you add a level that does not exist?

Lists. *Lists* are very flexible data structures used extensively in R. A list is a vector, but the elements of a list do not need to be of the same type. Each element of a list can be *any* R object, including another list. lists can be created using the `list()` function. List elements are usually extracted by name (using the `$` operator).

```
> x <- list(fun = seq, len = 10)
> x$fun
function (...)
UseMethod("seq")
<bytecode: 0x2463718>
<environment: namespace:base>
> x$len
[1] 10
> x$fun(length = x$len)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Functions are R objects too. In this case, the `fun` element of `x` is the already familiar `seq()` function, and can be called like any other function.

Lists give us the ability to create *composite objects* that contain several related, simpler objects. Many useful R functions return a list rather than a simple vector. Here is a more natural example, where three sets of observations on ten patients at an asylum are collected together. The observations recorded are the average increase in the hours of sleep given three sleep-inducing drugs, compared to a control average where no medication was given.

```
> extra.hyoscyamine <- c(0.7, -1.6, -0.2, -1.2, -0.1, 3.4, 3.7, 0.8, 0.0, 2.0)
> extra.laevorotatory <- c(1.9, 0.8, 1.1, 0.1, -0.1, 4.4, 5.5, 1.6, 4.6, 3.4)
> extra.racemic <- c(1.5, 1.4, 0.0, -0.7, 0.5, 5.1, 5.7, 1.5, 4.7, 3.5)
> extra.sleep <- list(hyoscyamine = extra.hyoscyamine,
                      laevorotatory = extra.laevorotatory,
                      racemic = extra.racemic)
> extra.sleep
$hyoscyamine
 [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0

$laevorotatory
```

```
 [1]  1.9  0.8  1.1  0.1 -0.1  4.4  5.5  1.6  4.6  3.4

$racemic
 [1]  1.5  1.4  0.0 -0.7  0.5  5.1  5.7  1.5  4.7  3.5
> extra.sleep$racemic
 [1]  1.5  1.4  0.0 -0.7  0.5  5.1  5.7  1.5  4.7  3.5
> extra.sleep[[2]]
 [1]  1.9  0.8  1.1  0.1 -0.1  4.4  5.5  1.6  4.6  3.4
```

List elements can be extracted by name as well as position.

Data Frames. Lists like the one above are so commonly used in statistical applications that R has a special data type for them. *Data frames* are R objects that represent (rectangular) data sets. They are essentially lists with some additional structure, but they conceptually represent multiple columns of observations taken over multiple observational units represented by rows.

Each column of a data frame has to be a either a factor or a numeric, character, or logical vector. Each of these must have the same length. They are similar to matrices because they have the same *rectangular array* structure; the only difference is that different columns of a data frame can be of different data types.

Data frames are created by the `data.frame()` function.

```
> d <- data.frame(extra.hyoscyamine, extra.laevorotatory, extra.racemic)
> d
   extra.hyoscyamine extra.laevorotatory extra.racemic
1                0.7                 1.9           1.5
2               -1.6                 0.8           1.4
3               -0.2                 1.1           0.0
4               -1.2                 0.1          -0.7
5               -0.1                -0.1           0.5
6                3.4                 4.4           5.1
7                3.7                 5.5           5.7
8                0.8                 1.6           1.5
9                0.0                 4.6           4.7
10               2.0                 3.4           3.5
> d$extra.racemic
 [1]  1.5  1.4  0.0 -0.7  0.5  5.1  5.7  1.5  4.7  3.5
```

The list-like `$` operator can be used to extract columns.

Indexing. Extracting one or more elements from a vector is done by *indexing*. Several kinds of indexing are possible in R, among them

- Indexing by a vector of positive integers
- Indexing by a vector of negative integers
- Indexing by a logical vector
- Indexing by a vector of names

In each case, the extraction is done by following the vector by a pair of brackets (`[...]`). The type of indexing depends on the object inside the brackets.
Here are some examples of indexing by positive integers.

```
> extra.hyoscyamine
 [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0
> extra.hyoscyamine[5]
[1] -0.1
```

```
> extra.hyoscyamine[c(3,5,7)]
[1] -0.2 -0.1  3.7
> ind <- c(3,5,7)
> extra.hyoscyamine[ind]
[1] -0.2 -0.1  3.7
> extra.hyoscyamine[8:12]
[1] 0.8 0.0 2.0  NA  NA
> extra.hyoscyamine[c(1, 2, 1, 2)]
[1]  0.7 -1.6  0.7 -1.6
```

This works more or less as expected. Using an index bigger than the length of the vector produces `NA`'s. Indices can be repeated, resulting in the same element being chosen more than once. This feature is often very useful.

Using negative integers as indices leaves out the specified elements.

```
> extra.hyoscyamine
 [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0
> extra.hyoscyamine[-5]
[1]  0.7 -1.6 -0.2 -1.2  3.4  3.7  0.8  0.0  2.0
> ind <- -c(3,5,7)
> ind
[1] -3 -5 -7
> extra.hyoscyamine[ind]
[1]  0.7 -1.6 -1.2  3.4  0.8  0.0  2.0
```

Negative indices cannot be mixed with positive indices.

For indexing by a logical vector, the logical vector being used as the index should be exactly as long as the vector being indexed. If it is shorter, it is replicated to be as long as necessary. Only the elements that correspond to `TRUE` are retained.

```
> extra.hyoscyamine
 [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0
> ind <- rep(c(TRUE, FALSE), length = length(extra.hyoscyamine))
> ind
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
> extra.hyoscyamine[ind]
[1]  0.7 -0.2 -0.1  3.7  0.0
> extra.hyoscyamine[c(T, F)] ## T = TRUE, F = FALSE
[1]  0.7 -0.2 -0.1  3.7  0.0
```

Indexing by names works only for vectors that have names.

```
> names(extra.hyoscyamine) <- LETTERS[1:10]
> extra.hyoscyamine
   A    B    C    D    E    F    G    H    I    J
 0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0
> extra.hyoscyamine[c('A', 'B', 'C', 'K')]
   A    B    C <NA>
 0.7 -1.6 -0.2   NA
> names(extra.hyoscyamine) <- NULL ## revert
```

All these types of indexing works for matrices and arrays as well, as we shall see later.

Exercise 8. *What happens if you use a factor as an indexing vector? For example, consider*

```
my.f <- gl(5, 1, 50)
x <- c("a", "b", "c", "d", "e")
```

What do you think `x[my.f]` *will look like? What general rule can you infer from this?*

Logical comparisons. All the usual logical comparisons are possible:

less than	<	less than or equal to	<=
greater than	>	greater than or equal to	>=
equals	==	does not equal	!=

Each of these operate on two vectors element-wise (the shorter one is replicated if necessary).

```
> extra.hyoscyamine
 [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0
> extra.hyoscyamine > 0
 [1]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
> extra.hyoscyamine < extra.racemic
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Element-wise boolean operations are also possible.

AND	&
OR	|
NOT	!

```
> extra.hyoscyamine
 [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0
> extra.hyoscyamine > 0
 [1]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
> extra.hyoscyamine < 1
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
> extra.hyoscyamine > 0 & extra.hyoscyamine < 1
 [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

Conditional Selection. Logical comparisons and indexing by logical vectors together allow subsetting a vector based on the properties of other (or perhaps the same) vectors.

```
> extra.racemic
 [1]  1.5  1.4  0.0 -0.7  0.5  5.1  5.7  1.5  4.7  3.5
> extra.racemic[extra.hyoscyamine > 0]
[1] 1.5 5.1 5.7 1.5 3.5
> extra.racemic[extra.hyoscyamine > 0 & extra.hyoscyamine < 2]
[1] 1.5 1.5
> month.name[month.name > "N"]
[1] "September" "October"   "November"
```

For character vectors, sorting is determined by alphabetical order.

Matrix and Data frame indexing. Indexing for matrices and data frames are similar: they also use brackets, but need two indices. If one (or both) of the indices are unspecified, all the corresponding rows and columns are selected.

```
> x <- matrix(1:12, 3, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> x[1:2, 1:2]
     [,1] [,2]
[1,]    1    4
[2,]    2    5
> x[1:2, ]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
```

If only one row or column is selected, the result is converted to a vector. This can be suppressed by adding a `drop = FALSE`.

```
> x[1,]
[1]  1  4  7 10
> x[1,,drop = FALSE]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
```

Data frames behave similarly.

```
> d[1:3,]
  extra.hyoscyamine extra.laevorotatory extra.racemic
1               0.7                 1.9           1.5
2              -1.6                 0.8           1.4
3              -0.2                 1.1           0.0
> d[1:3, "extra.hyoscyamine"]
[1]  0.7 -1.6 -0.2
> d[d$extra.racemic < 1.5, 1, drop = FALSE]
  extra.hyoscyamine
2              -1.6
3              -0.2
4              -1.2
5              -0.1
```

Modifying objects. It is usually possible to modify R objects by assigning a value to a subset or function of that object. For the most part, anything that makes sense, works. This will become clearer with more experience.

```
> x <- runif(10, min = -1, max = 1)
> x
 [1]  0.626695178  0.644059978 -0.003825108 -0.698547989 -0.314868690
 [6]  0.123496425 -0.990746668  0.519975554 -0.742774245  0.225232550
> x < 0
 [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
> x[x < 0] <- 0
> x
 [1] 0.6266952 0.6440600 0.0000000 0.0000000 0.0000000 0.1234964 0.0000000
 [8] 0.5199756 0.0000000 0.2252325
```

This is a very powerful feature of R that is not particularly common in other languages.

Manipulating data frames. New columns can be added to data frame, by assigning to a currently non-existent column name (this works for lists too):

```
> d$difference.HR
NULL
> d$difference.HR <- d$extra.racemic - d$extra.hyoscyamine
> d
   extra.hyoscyamine extra.laevorotatory extra.racemic difference.HR
1                0.7                 1.9           1.5           0.8
2               -1.6                 0.8           1.4           3.0
3               -0.2                 1.1           0.0           0.2
4               -1.2                 0.1          -0.7           0.5
5               -0.1                -0.1           0.5           0.6
6                3.4                 4.4           5.1           1.7
7                3.7                 5.5           5.7           2.0
8                0.8                 1.6           1.5           0.7
9                0.0                 4.6           4.7           4.7
10               2.0                 3.4           3.5           1.5
```

But working with data frames this way can become a bit cumbersome because we always need to prefix the name of the data frame to every column. There are several functions to make this easier. For example, `subset()` can be used to select rows of a data frame.

```
> library(ISwR)
> data(thuesen)
> str(thuesen)
'data.frame':        24 obs. of  2 variables:
 $ blood.glucose : num  15.3 10.8 8.1 19.5 7.2 5.3 9.3 11.1 7.5 12.2 ...
 $ short.velocity: num  1.76 1.34 1.27 1.47 1.27 1.49 1.31 1.09 1.18 1.22 ...
> thue2 <- subset(thuesen, blood.glucose < 7)
> thue2
   blood.glucose short.velocity
6            5.3           1.49
11           6.7           1.25
12           5.2           1.19
15           6.7           1.52
17           4.2           1.12
22           4.9           1.03
```

Similarly, the `transform()` function can be used to add new variables to a data frame using the old ones.

```
> thue3 <- transform(thue2, log.gluc = log(blood.glucose))
> thue3
   blood.glucose short.velocity log.gluc
6            5.3           1.49 1.667707
11           6.7           1.25 1.902108
12           5.2           1.19 1.648659
15           6.7           1.52 1.902108
```

```
17           4.2           1.12 1.435085
22           4.9           1.03 1.589235
```

Another similar and very useful function is `with()`, which can be used to evaluate arbitrary expressions using variables in a data frame:

```
> with(thuesen, log(blood.glucose))
 [1] 2.727853 2.379546 2.091864 2.970414 1.974081 1.667707 2.230014 2.406945
 [9] 2.014903 2.501436 1.902108 1.648659 2.944439 2.714695 1.902108 2.151762
[17] 1.435085 2.332144 2.525729 2.778819 2.587764 1.589235 2.174752 2.251292
```

Grouped data. Grouped data have one or more numerical variables, and one or more categorical factors (also called groups) that indicate the category for each observation. The most natural way to store such data is as data frames with different columns for the numerical and categorical variables.

```
> data(energy)
> str(energy)
'data.frame':        22 obs. of  2 variables:
 $ expend : num  9.21 7.53 7.48 8.08 8.09 ...
 $ stature: Factor w/ 2 levels "lean","obese": 2 1 1 1 1 1 1 1 1 1 ...
> summary(energy)
     expend        stature
 Min.   : 6.130   lean :13
 1st Qu.: 7.660   obese: 9
 Median : 8.595
 Mean   : 8.979
 3rd Qu.: 9.900
 Max.   :12.790
```

We often want to extract information by group. We can do so using what we have already learnt.

```
> exp.lean <- energy$expend[energy$stature == "lean"]
> exp.obese <- with(energy, expend[stature == "obese"])
> exp.lean
 [1]  7.53  7.48  8.08  8.09 10.15  8.40 10.88  6.13  7.90  7.05  7.48  7.58
[13]  8.11
> exp.obese
[1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69  9.68  9.19
```

A more sophisticated way to do this is

```
> l <- with(energy, split(x = expend, f = stature))
> l
$lean
 [1]  7.53  7.48  8.08  8.09 10.15  8.40 10.88  6.13  7.90  7.05  7.48  7.58
[13]  8.11

$obese
[1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69  9.68  9.19
```

More generally, arbitrary functions can be applied to data frames split by a group using the `by()` function:

```
> by(data = energy, INDICES = energy$stature, FUN = summary)
energy$stature: lean
     expend        stature
 Min.   : 6.130   lean :13
 1st Qu.: 7.480   obese: 0
```

```
 Median : 7.900
 Mean   : 8.066
 3rd Qu.: 8.110
 Max.   :10.880
------------------------------------------------------------
energy$stature: obese
     expend       stature
 Min.   : 8.79   lean :0
 1st Qu.: 9.21   obese:9
 Median : 9.69
 Mean   :10.30
 3rd Qu.:11.51
 Max.   :12.79
```

Sorting. Vectors can be sorted by `sort()`.

```
> sort(extra.racemic)

 [1] -0.7  0.0  0.5  1.4  1.5  1.5  3.5  4.7  5.1  5.7
```

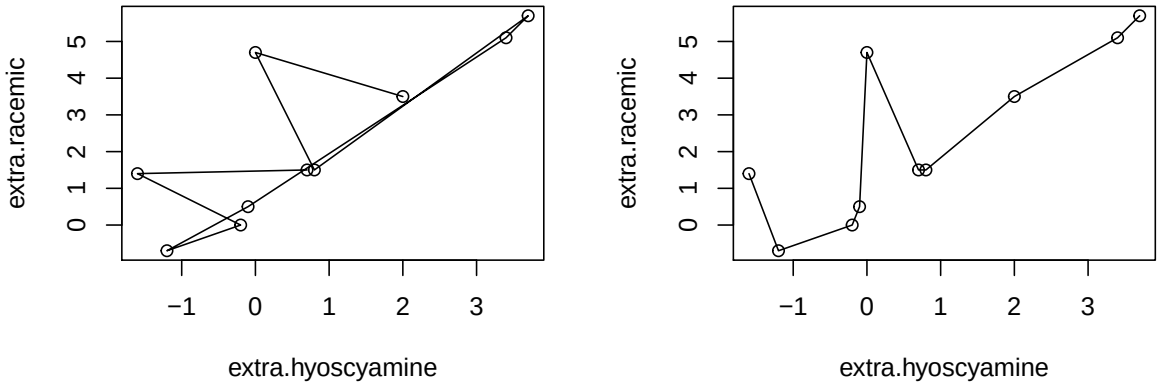
But it is usually more useful to work with the *sort order*, using the `order()` function, which returns an integer indexing vector that can be used get the sorted vectors. This can be useful to re-order the rows of a data frame by one or more columns.

```
> ord <- order(extra.racemic)
> ord

 [1]  4  3  5  2  1  8 10  9  6  7

> extra.racemic[ord]

 [1] -0.7  0.0  0.5  1.4  1.5  1.5  3.5  4.7  5.1  5.7

> d[ord, ]

   extra.hyoscyamine extra.laevorotatory extra.racemic difference.HR
4               -1.2                 0.1          -0.7           0.5
3               -0.2                 1.1           0.0           0.2
5               -0.1                -0.1           0.5           0.6
2               -1.6                 0.8           1.4           3.0
1                0.7                 1.9           1.5           0.8
8                0.8                 1.6           1.5           0.7
10               2.0                 3.4           3.5           1.5
9                0.0                 4.6           4.7           4.7
6                3.4                 4.4           5.1           1.7
7                3.7                 5.5           5.7           2.0
```

Exercise 9. *The plot on the left below is produced by*

```
> with(d, plot(extra.hyoscyamine, extra.racemic, type = "o"))
```

The `type="o"` *argument in* `plot()` *causes the points to be joined by line segments, in the order in which the data are provided (in the* `x` *and* `y` *arguments). Modify the call so that points are joined in the order of increasing x-value, as in the plot on the right.*


extra.racemic
0 1 2 3 4 5
−1 0 1 2 3
extra.hyoscyamine
extra.racemic
0 1 2 3 4 5
−1 0 1 2 3
extra.hyoscyamine