Interpreting vs. compiling

In order to tell a computer what to do, we have to communicate with it. Usually we first write a list of commands (a computer program) in a high-level programming language (such as C or S), and then use a compiler or interpreter to translate the commands into machine code.

A **compiler** translates a complete program in one go and produces a machine code program which we can run and re-run any time we want.

An **interpreter**, on the other hand, reads, translates, and immediately executes the commands one by one. If a command is encountered multiple times (for example in a loop), it has to be translated each time. Interpreted programs run therefore generally much slower than compiled programs.

Programs written in C are compiled and programs written in S are interpreted. R is a free interpreter for S and S-PLUS is a commercial interpreter for S.

R can be downloaded from **www.r-project.org**.

After the installation of R, it can be started by double-clicking on the R icon.

Prospective R users must prepare for typing commands instead of clicking on menus and dialog boxes.

Using R as a calculator

To evaluate the expression $\sin(\frac{\pi}{2}) - 2^3\left|1-\sqrt{9}\right|$ type

sin(pi/2)-2^3*abs(1-sqrt(9))

directly into the **R Console** at the prompt **>** and press **Enter**.

R immediately executes the command, prints

[1] -15,[1]

and waits for the next command.

[1] The result -15 is preceded by the counter [1]. The inclusion of counters will turn out to be helpful when longer output is printed

Assignments

If you want to save the value of an expression evaluated by R, you need to store it into a variable.

The assignment

x <- pi/4

first evaluates the expression $\frac{\pi}{4}$ and then passes the result to the variable **x**. It does not print the result. If you want to see the value stored in the variable **x**, just type its name.

Choose variable names with care. If you choose short names such as **x**, **y**, or **z**, nobody will be able to guess what they are used for. On the other hand, really long names such as **pi.divided.by.4** , **pi_divided_by_4**, or **piDividedBy4** are much harder to read and also much harder to write.

Using the variable **x** created by the assignment

x <- pi/4

we can evaluate the expression

$$\sin^2(\tfrac{\pi}{4}) + \cos^2(\tfrac{\pi}{4})$$

by entering the command

(sin(x))^2+(cos(x))^2.

Note: Putting one space before and after <- is not necessary, but it improves readability.

Vectors

Vectors can be constructed with the concatenate function, **c**.

```
> c(0.1,-2.2,5,8.8)
[1] 0.1 -2.2 5.0 8.8
```

More regular vectors can also be obtained with the sequence function, **seq**.

```
> seq(2,10,by=2)
[1] 2 4 6 8 10

> seq(2,-3.5,by=-0.5)
[1] 2.0 1.5 1.0 0.5 0.0 -0.5 -1.0 -1.5 -2.0 -2.5 -3.0 -3.5

> seq(1,10,by=1)
[1] 1 2 3 4 5 6 7 8 9 10
```

If the increment is 1 or -1, a shorthand for **seq** is to use the colon, **:**.

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

A subvector is obtained by giving the name of the vector followed by an index vector in square brackets.

```
> v <- c(-2,12,0,7,-1,867)
> v[c(1,3,5,6)]
[1] -2 0 -1 867

> v[1:3]+100*c(0,1,2)
[1]  -2 112 200
```

Matrices

Vectors can be bound together into matrices, either row by row or column by column.

```
> rbind(2:4,10:12)
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]   10   11   12
```

```
> cbind(2:4,10:12,c(-7,8,3),10*c(1,2,4),3:1,(3:1)^2)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    2   10   -7   10    3    9
[2,]    3   11    8   20    2    4
[3,]    4   12    3   40    1    1
```

The indexing of matrices is analogous to that of vectors.

```
> X <- rbind(1:5,seq(10,50,10))
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]   10   20   30   40   50
```

```
> X[2,3:5]
[1] 30 40 50
```

```
> X[1,4]
[1] 4
```

```
> X[,4]
[1]  4 40
```

```
> X[1,]
[1] 1 2 3 4 5
```

Data frames and lists

The columns of a matrix must be of the same length and the same type. In contrast, the elements of a data.frame do not have to be of the same type. For example, an object with names (type: character) in the first column and values (type: numeric) in the other columns is a data.frame, but not a matrix.

```
> index <- c("DJI","FTSE 100","NIKKEI  225")
> n.cmpn <- c(30,100,225) # number of components
> cl <- c(10520.32, 5260.99, 10695.69) # close prices
> F <- data.frame(index,n.cmpn,cl)
> F
          index n.cmpn       cl
1           DJI     30 10520.32
2      FTSE 100    100  5260.99
3 NIKKEI  225     225 10695.69
```

The elements of a list can have different lengths as well as different types.

```
> day <- "May 06 2010"
> L <- list(day,F)
```

The elements of a list can be referenced using double square brackets.

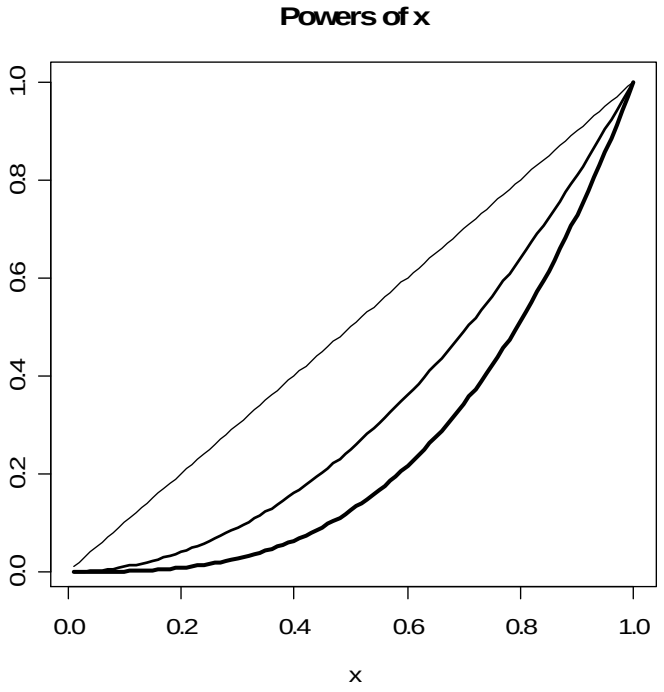
```
> L[[1]]
[1] "May 06 2010"

> L[[2]][3,2]
[1] 225
```

Plotting data

```
> x<-1:10; y<-x^2; plot(x,y) # create x,y, plot y vs. x
```

Note: (i) Commands that are typed into the same line must be separated by a semi-colon, ;.
(ii) Comments must be preceded with the number sign, #. Everything to the right of this symbol will be ignored.

```
> plot(x,y,col="red")  # plotting color = red
> plot(x,y,type="l") # plot a line instead of points
> plot(x,y,type="o") # points and line overplotted
> n <- 100
> x <- (1:n)/n # if x already exists it is overwritten
> y1 <- x; y2 <- x^2; y3 <- x^3
> plot(x,y1,type="l",ylab="") # no Y axis label
> title("Powers of x") # add a title to the plot
> lines(x,y2,lwd=2) # add a line with line width = 2
> lines(x,y3,lwd=3) # add a line with line width = 3
```



Batch processing

Commands can be typed into the **R Console** and executed immediately by pressing **Enter**. Alternatively, a sequence of commands can be assembled into a text file and then be executed in batch mode.

For example, we may first create a working directory **C:\Projects\Powers** and then use a simple text editor to store all the R commands

```
n <- 100; x <- (1:n)/n; y1 <- x; y2 <- x^2; y3 <- x^3
plot(x,y1,type="l",ylab="") # no Y axis label
title("Powers of x") # add a title to the plot
lines(x,y2,lwd=2); lines(x,y3,lwd=3)
```

on an external file, say **plot.txt**, in the working directory.

Next we start R, enter the command

```
setwd("C:/Projects/Powers") # R uses / instead of \
```

to set the R working directory to **C:\Projects\Powers**, and finally execute the stored commands with the command

```
source("plot.txt") # read, analyze (parse), evaluate
```

or simply with copy and paste.

It is often reasonable to carry out preliminary analyses interactively and to produce the final results in batch mode to guarantee full reproducibility.

Getting help and quitting

If you get stuck, you should first interrupt the current operation by pressing **Esc** and then get help. The command **help()** starts the R help facility.

If you know the name of a specific function, for example **plot**, you can get information on this function by entering **help(plot)**.

When you terminate an R session by entering the command **quit()** or via the **File** menu, you are asked whether you want to save the objects that you have created during that session. If your answer is **Yes**, R writes all objects to a workspace file called **.RData** in the current working directory. The workspace can be reloaded in later R sessions by first selecting the working directory and then entering **load(".RData")**.