Introduction
Computing at Hopkins Biostatistics

Ingo Ruczinski

Thanks to Thomas Lumley and Robert Gentleman of the R-core group (http://www.r-project.org/) for providing some tex files that appear in part in the following slides.

Some R facts

- R is an environment for data analysis and visualization.
- R is an open source implementation of the S language (S-Plus is a commercial implementation of the S language).
- The current version of R (September 2004) is 1.9.1.
- The R Core group consists of Doug Bates, John Chambers, Peter Dalgaard, Robert Gentleman, Kurt Hornik, Stefano Iacus, Ross Ihaka, Friedrich Leisch, Thomas Lumley, Martin Maechler, Guido Masarotto, Paul Murrell, Brian Ripley, Duncan Temple Lang, and Luke Tierney.
- If you use R extensively, be a good citizen and join the R Foundation for Statistical Computing http://www.r-project.org/ .

The R license

- R is both open source and open development.
- You can look at the source code and you can propose changes.
- R is not in the public domain.
- You are given a license to run the software (currently GPL).

The R software

- R is mainly written in C.
- R is available for many platforms:
  - Unix of many flavors, including Linux, Solaris, FreeBSD, AIX.
  - Windows 95 and later.
  - MacOS X.
- Binaries and source code are available from www.r-project.org .
- R "talks" to data bases, programming languages, and other statistical packages.
- R should be source code compatible with most of the Splus code written.

How do I get R?

- The informational web site http://www.r-project.org/ .
- CRAN - the Comprehensive R Archive Network:
  - The primary site is http://cran.r-project.org/ .
  - Mirror sites are available for many countries. For example: http://cran.us.r-project.org/ .
- CRAN sites have source code andbinary distributions for Windows 95, 98, ME, NT4, 2000 and XP on Intel and compatible processor, for the Macintosh (System 8.6 to 9.1 and MacOS X), and for several Linux distributions.
- New releases occur frequently, at least twice a year. Be prepared to re-install frequently.

Installing R

- Windows:
  Download and run the `SetupR.exe` installer.
- Macintosh:
  Download `R.dmg`, and double-click on the `R-1.9.1.pkg` icon on the `R.dmg` disk image.
- Linux:
  RPM files are available for RedHat, SuSE, and Mandrake. Deb files are available for Debian.
- Unix/Linux:
  Download and expand the compressed tar file of the sources. Run `./configure`, and then `make`, `make check`, and `make install`.

Installing R with a script

Create the directory `R-lr` in your root directory (for example, mine is `/home/iruczins/`). Edit the following script and copy it to your `bin/` directory.

```
rsync -rC rsync.r-project.org::r-release /home/iruczins/R-lr
cd /home/iruczins/R-lr
./configure --prefix=/home/iruczins/R-lr
make
make install
echo Done!
```

Make sure you can execute it (`chmod 755 [filename]` will do). When you run it, `rsync` will automatically grab the latest released R version for you.

Create an alias in your `.tcshrc` file, or whatever shell you use:

```
alias R '/home/iruczins/R-lr/bin/R'
```

Sources of information about R

- The web site http://www.r-project.org/ and CRAN.
- Kurt Hornik's page http://www.ci.tuwien.ac.at/~hornik/R/R-FAQ.html mirrored at http://cran.r-project.org/doc/FAQ/R-FAQ.html . Most of the contents of these slides (and many other topics) are covered in this FAQ site.
- The manuals at http://cran.r-project.org/manuals.html . In particular, check out `R-intro.pdf` and `R-data.pdf`.
- Karl's R page at http://www.biostat.jhsph.edu/~kbroman/Rintro/
- More detailled notes are on the Statistical Computing class page at http://www.biostat.jhsph.edu/~bcaffo/statcomp/

A sample session

```
> log(64)
[1] 4.158883

> log2(64)
[1] 6

> sqrt(2)
[1] 1.414214

> sqrt(-1)
[1] NaN
Warning message:
NaNs produced in: sqrt(-1)

> sqrt(-1+0i)
[1] 0+1i
```

A sample session

```
> x <- 5
> x
[1] 5

> x = 5
> x
[1] 5

> x <- c(1,2,3,4)
> x
[1] 1 2 3 4

> x <- 1:4
> x
[1] 1 2 3 4

> x <- seq(1,4)
> x
[1] 1 2 3 4
```

A sample session

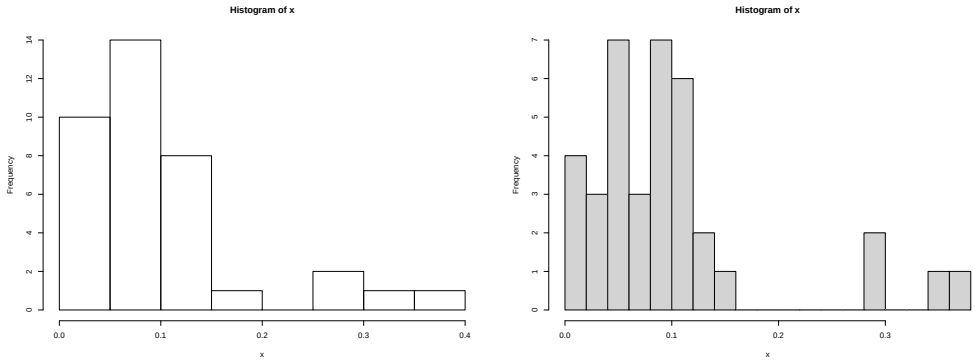
```
> x <- c(0.008, 0.018, 0.056, 0.055, 0.135,
+        0.052, 0.077, 0.026, 0.044, 0.300,
+        0.025, 0.036, 0.043, 0.100, 0.120,
+        0.110, 0.100, 0.350, 0.100, 0.300,
+        0.011, 0.060, 0.070, 0.050, 0.080,
+        0.110, 0.110, 0.120, 0.133, 0.100,
+        0.100, 0.155, 0.370, 0.019, 0.100,
+        0.100, 0.116)

> x
 [1] 0.008 0.018 0.056 0.055 0.135 0.052 0.077 0.026 0.044
[10] 0.300 0.025 0.036 0.043 0.100 0.120 0.110 0.100 0.350
[19] 0.100 0.300 0.011 0.060 0.070 0.050 0.080 0.110 0.110
[28] 0.120 0.133 0.100 0.100 0.155 0.370 0.019 0.100 0.100
[37] 0.116

> class(x)
[1] "numeric"

> length(x)
[1] 37
```

A sample session

```
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0080  0.0500  0.1000  0.1043  0.1160  0.3700

> quantile(x, c(0.1, 0.95))
   10%    95%
0.0226 0.3100

> mean(x);median(x);sd(x);var(x)
[1] 0.1042973
[1] 0.1
[1] 0.08895344
[1] 0.007912715

> round(c(mean(x),median(x),sd(x),var(x)),3)
[1] 0.104 0.100 0.089 0.008
```

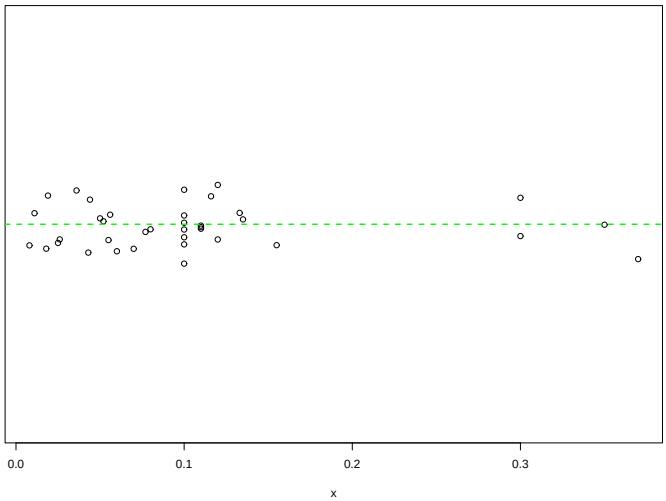
A sample session

```
> exp(mean(log(x)))
[1] 0.07463571

> 1/mean(1/x)
[1] 0.04841714

> hist(x)
> hist(x,breaks=15,col="lightgrey")
```



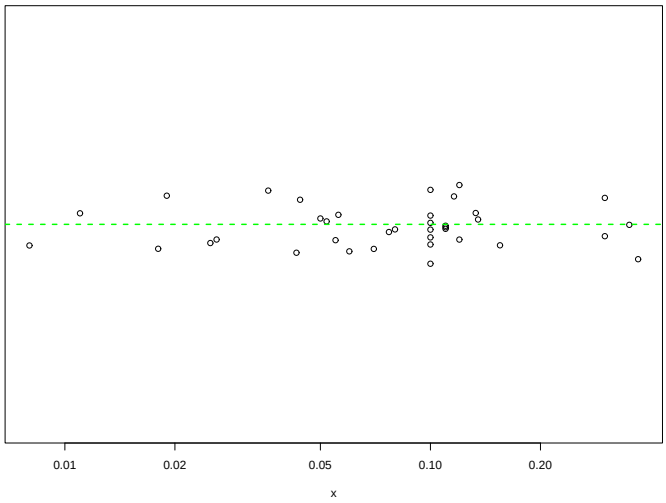
A sample session

```
> y <- runif(length(x))
> plot(x, y, ylim=c(-2,3), yaxt="n", ylab="")
> abline(h=0.5, lty=2, col="green",lwd=2)
```



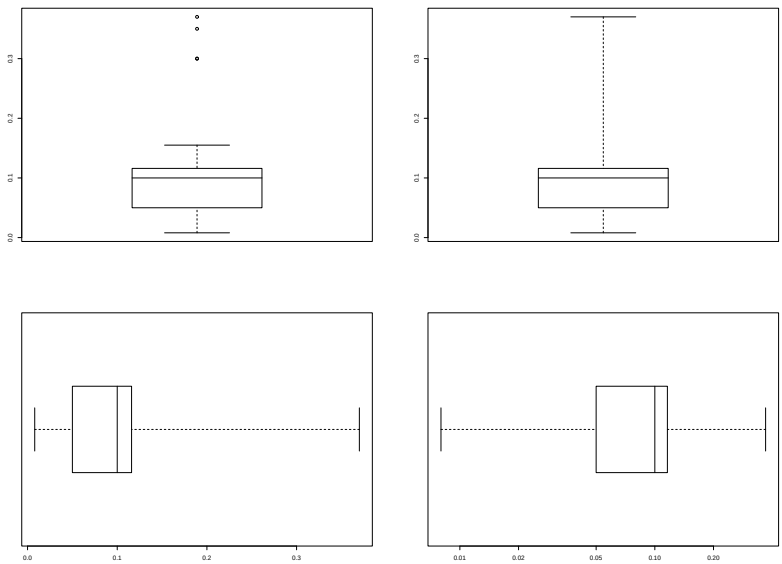
A sample session

```
> plot(x, y, ylim=c(-2,3), yaxt="n", ylab="", log="x")
> abline(h=0.5, lty=2, col="green",lwd=2)
```



A sample session

```
> boxplot(x)
> boxplot(x, range=0)
> boxplot(x, range=0, horizontal=TRUE)
> boxplot(x, range=0, horizontal=TRUE, log="x")
```

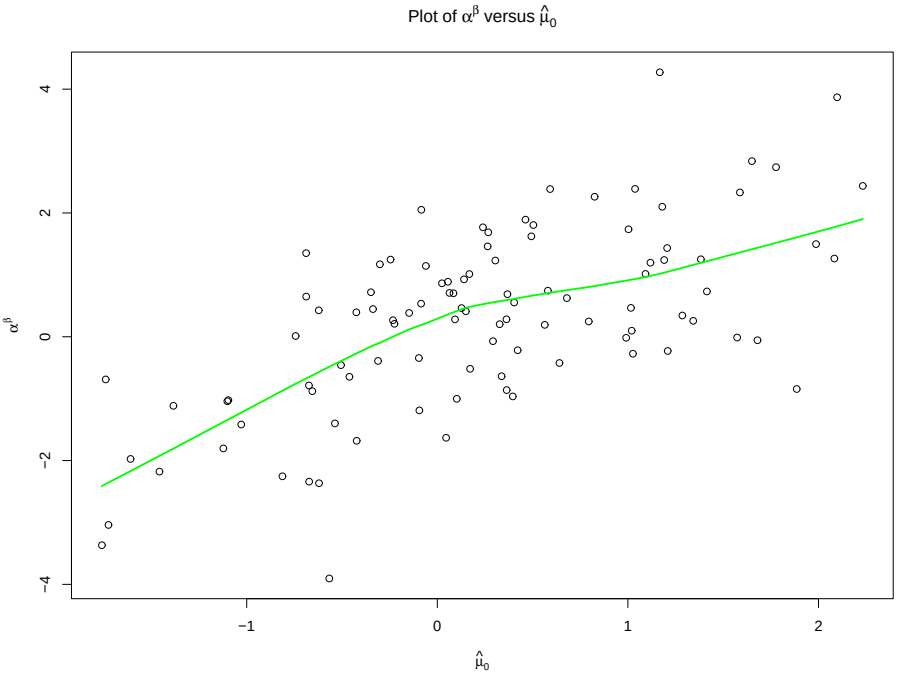


A sample session

One of the nice additions to R (compared to Splus) is the easy inclusion of mathematical expressions in plots using the function `expression`. Take a look at `help(plotmath)` to see a big list of what you can do; also look at the examples in the help file for the function legend.

```
x <- rnorm(100)
y <- x+rnorm(100)
plot(x,y,
     xlab=expression(hat(mu)[0]),
     ylab=expression(alphaˆbeta),
     main=expression(paste(
     "Plot of ",alphaˆbeta," versus ",hat(mu)[0])))
lines(lowess(x,y),col="green",lwd=2)
```

A sample session



A sample session

```
> x <- 1:4
> x
[1] 1 2 3 4

> x*x
[1]  1  4  9 16

> z <- x %*% x
> z
     [,1]
[1,]   30

> drop(z)
[1] 30
```

A sample session

```
> y <- diag(x)
> y
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4

> solve(y)
     [,1] [,2]      [,3] [,4]
[1,]    1  0.0 0.0000000 0.00
[2,]    0  0.5 0.0000000 0.00
[3,]    0  0.0 0.3333333 0.00
[4,]    0  0.0 0.0000000 0.25

> det(y)
[1] 24
```

A sample session

```
> z <- matrix(sample(1:12), ncol = 3, nrow = 4)
> z
     [,1] [,2] [,3]
[1,]    7   12    8
[2,]    4    5    9
[3,]    2    1    6
[4,]   10   11    3

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

A sample session

```
> y %*% z
     [,1] [,2] [,3]
[1,]    7   12    8
[2,]    8   10   18
[3,]    6    3   18
[4,]   40   44   12

> y %*% x
     [,1]
[1,]    1
[2,]    4
[3,]    9
[4,]   16

> x %*% z
     [,1] [,2] [,3]
[1,]   61   69   56
```

A sample session

```
> solve(t(z)%*%z) %*% (t(z)%*%x)

             [,1]
[1,]  1.1210197
[2,] -0.6962714
[3,]  0.1637485

> A <- t(z)%*%z
> b <- t(z)%*%x
> solve(A,b)
           [,1]
[1,]  1.1210197
[2,] -0.6962714
[3,]  0.1637485
```

Getting help

- Check the help files and manuals.
  - For example, to check out how the function weighted.mean() works, type ?weighted.mean or help(weighted.mean) at the R prompt.
  - If you do not know the exact name of the function, type for example help.search("mean") or use the html search engine, which you can start by typing help.start().
- Ask some R wizard.
- Use the mailing list archives and search facilities.
- Post your question to the mailing list.

9 basic R functions you should know

Typing a function name and hitting the <RET> button simply displays the function. To invoke the function you must include an argument list in `()`, even if the list is empty. That is, use `q()` <RET> and not just `q` <RET>.

- `q` Quit the session
- `help` Get help on a function or object
- `help.start` Allow the use of a web browser for reading help
- `example` Run the example from the help page for an object
- `data` List the available data sets or import a data set
- `library` List available packages or attach a package
- `objects` List the objects in the workspace
- `summary` Summarize an object
- `str` Show the low-level structure of an object

The R package system

- Packages are self-contained units of code with documentation.
- The packages are simple to obtain, understand, and update. Try commands like `install.packages()`, `example()`, and `update.packages()`.
- You can write your own packages!
- All functions must have examples and the examples must run.
- There are automatic testing features built in.

Downloading packages and bundles

```
> install.packages("rpart")
> library(rpart)
> data(kyphosis)

> kyphosis
   Kyphosis Age Number Start
1    absent  71      3     5
2    absent 158      3    14
3   present 128      4     5
4    absent   2      5     1
5    absent   1      4    15
6    absent   1      2    16
7    absent  61      2    17
8    absent  37      3    16
9    absent 113      2    16
10  present  59      6    12
11  present  82      5    14
12   absent 148      3    16
13   absent  18      5     2
...
```

`read.table`

The function `read.table` is the most convenient way to read in a rectangular grid of data. Some of the issues to consider are:

- Header line
- Separator
- Quoting
- Missing values
- Unfilled lines
- White space in character fields
- Blank lines
- Classes for the variables
- Comments

read.table

```
>?read.table

read.table               package:base               R Documentation

Data Input

Description:

     Reads a file in table format and creates a data frame from it,
     with cases corresponding to lines and variables to fields in the
     file.

Usage:

     read.table(file, header = FALSE, sep = "", quote = "\"'", dec = ".",
                row.names, col.names, as.is = FALSE, na.strings = "NA",
                colClasses = NA, nrows = -1,
                skip = 0, check.names = TRUE, fill = !blank.lines.skip,
                strip.white = FALSE, blank.lines.skip = TRUE,
                comment.char = "#")
```

Good to know

Because of the many possibilities, there are several other functions that call `read.table` but change a group of default arguments:

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, ...)

read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
          fill = TRUE, ...)

read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
           fill = TRUE, ...)

read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
            fill = TRUE, ...)
```

The function `count.fields` can be useful when you get an error message!

Fixed-width-format files

Sometimes data files have no field delimiters but have fields in pre-specified columns. The function `read.fwf` provides a simple way to read such files, specifying a vector of field widths.

```
> ff <- tempfile()
> cat(file=ff, "123456", "987654", sep="\n")
> read.fwf(ff, width=c(1,2,3))
  V1 V2  V3
1  1 23 456
2  9 87 654
> unlink(ff)

> cat(file=ff, "123", "987654", sep="\n")
> read.fwf(ff, width=c(1,0,2,3))
  V1 V2 V3  V4
1  1 NA 23  NA
2  9 NA 87 654
> unlink(ff)
```

scan

Both `read.table` and `read.fwf` use `scan` to read the file, and then process the results of `scan`. They are very convenient, but sometimes it is better to use `scan` directly.

`scan` has many arguments in common with `read.table`. One additional argument is `what`, which specifies a list of modes of variables to be read from the file. If the list is named, the names are used for the components of the returned list. Modes can be numeric, character or complex, and are usually specified by an example, e.g. `0`, `""` or `0i`.

scan

```
>?scan

scan                    package:base                    R Documentation

Read Data Values

Description:

     Read data into a vector or list from the console or file.

Usage:

     scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
          quote = if (sep=="\n") "" else "'\"", dec = ".",
          skip = 0, nlines = 0, na.strings = "NA",
          flush = FALSE, fill = FALSE, strip.white = FALSE, quiet = FALSE,
          blank.lines.skip = TRUE, multi.line = TRUE, comment.char = "")
```

scan

```
> x <- scan()
1: 4
2: 7 3
4:
Read 3 items
> x
[1] 4 7 3

> cat("2 3 5 7", "11 13 17 19", file="ex.dat", sep="\n")
> scan(file="ex.dat")
Read 8 items
[1]  2  3  5  7 11 13 17 19
> scan(file="ex.dat", what=list(x=0, y="", z=0), flush=TRUE)
Read 2 records
$x
[1]  2 11
$y
[1] "3"  "13"
$z
[1]  5 17
```

source

You can use `source` to read in R code:

```
source("somecode.R")
```

The commands in `somecode.R` will be executed, and the objects specified will be created in your current `.RData` file. For example, this is quite convenient when you want to create a fairly fancy plot, and still need to tinker with the layout. You can also use `source` to read in functions you wrote:

```
myfunction <- source("myfunction.R")
```

The file you read in can contain more than one function statement. For example, if your main function calls some subfunctions, they can all be included in `myfunction.R`, and read in at the same time.

Customizing your R environment

Creating a `.Renviron` file in your root directory is a good idea. For example, when you use the `postscript` command to create a figure, the default for the paper format is 'A4'. You can change it to format 'letter' by typing

```
postscript("test.ps",paper="letter")
```

but you had to do this every time you want to create a postscript file. However, if you put the line

```
R_PAPERSIZE=letter
```

into your `.Renviron` file, R will use the 'letter' format as default. And it works in all subdirectories!

Customizing your R environment

Creating a `.Rprofile` file in your root directory is an equally good idea. For example, the commands

```
load("/home/iruczins/code/R/functions/.RData")
options(width=200,defaultPackages=c(getOption("defaultPackages"),
"nlme","rpart","survival"))
```

ensure that my own R functions are available when I start an R session, that the width of the screen output is 200 characters wide, and that the default libraries plus the libraries `nlme`, `rpart`, and `survival` are loaded. Works in all subdirectories!

Type `?Startup` to see what exactly R is doing.

Types of data in R

- The basic data object is a vector of elements of type:

  numeric : numbers, either floating point or integer.
  character : each element is a character string.
  logical : each element is `TRUE` or `FALSE`.
  list : elements can be any type of object, including other lists.
- Components of the S language, such as functions, are also vectors.
- Any vector can include the missing data marker `NA` as an element.
- All vectors have a `length` and a `mode`. The functions `length` and `mode` return this information as does the `str` function.
- A structure consists of a data object plus additional information. Arrays and time series are examples of structures.

Variables

```
> x <- 5
> mode(x)
[1] "numeric"

> x <- "I like chocolate ice cream"
> sub("chocolate","strawberry",x)
[1] "I like strawberry ice cream"
> mode(x)
[1] "character"

> x <- LETTERS[1:5]
> x
[1] "A" "B" "C" "D" "E"
> mode(x)
[1] "character"

> x <- (1+2==4)
> x
[1] FALSE
> mode(x)
[1] "logical"
```

Generating simple vectors

- The assignment operator in R is the two-character sequence '`<-`'. An alternative is available, but its use is discouraged.
- Any type of vector can be created explicitly with the c (concatenation) function.
- Numeric vectors can be generated with the `seq` function or the sequence operator '`:`'.
- The function `rep` generates a new vector by repeating a vector of any mode (including a list) a specified number of times.
- Pseudo-random samples from various distributions can be created. The function names have the pattern `r<distname>`, such as `runif` for a uniform distribution or `rnorm` for a normal distribution.

Numeric vectors

```
> rn <- rnorm(100)
> str(rn)
 num [1:100] -0.696 -0.158 -2.449 -0.383  0.665 ...
> stem(rn)

  The decimal point is at the |

  -2 | 4
  -1 | 98754322111
  -0 | 8887776644333332222211111100
   0 | 000011112222333333334445555666777777788
   1 | 000001111223567899
   2 | 046

> rn <- rpois(100,lambda=3)
> table(rn)
rn
 0  1  2  3  4  5  6  7  8
 3 16 22 20 20 11  4  3  1
```

Characters

```
> rep(c("A","B"),4)
[1] "A" "B" "A" "B" "A" "B" "A" "B"

> rep(c("A","B"),rep(4,2))
[1] "A" "A" "A" "A" "B" "B" "B" "B"

> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
[15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

> LETTERS[c(1,20,7,3)]
[1] "A" "T" "G" "C"

> x <- c("A","T","G","C")
> x
[1] "A" "T" "G" "C"
```

Characters

```
> expand.grid(x,x,x)
   Var1 Var2 Var3
1     A    A    A
2     T    A    A
3     G    A    A
4     C    A    A
5     A    T    A
6     T    T    A
7     G    T    A
8     C    T    A
9     A    G    A
10    T    G    A
...
57    A    G    C
58    T    G    C
59    G    G    C
60    C    G    C
61    A    C    C
62    T    C    C
63    G    C    C
64    C    C    C
```

Character and logical vectors

```
> fabfive <- c("Karl","Rafa","Roger","Ingo","Brian")
> str(fabfive)
 chr [1:5] "Karl" "Rafa" "Roger" "Ingo" "Brian"

> fabfive < "K"
[1] FALSE FALSE FALSE  TRUE  TRUE

> grep("a",fabfive)
[1] 1 2 5

> grep("a",fabfive,value=T)
[1] "Karl"  "Rafa"  "Brian"

> grep("[a-e]",fabfive,value=T)
[1] "Karl"  "Rafa"  "Roger" "Brian"

> gsub("[a-e]","X",fabfive)
[1] "KXrl"  "RXfX"  "RogXr" "Ingo"  "XriXn"
```

Character and logical vectors

A list is an ordered collection of data of arbitrary types.

```
> krrib=list(name=c("Karl","Rafa","Roger","Ingo","Brian"),
+            age=c(17,20,18,19,5),dad=c(F,T,F,F,F))

> krrib
$name
[1] "Karl"  "Rafa"  "Roger" "Ingo"  "Brian"

$age
[1] 17 20 18 19  5

$dad
[1] FALSE  TRUE FALSE FALSE FALSE

> krrib$name
[1] "Karl"  "Rafa"  "Roger" "Ingo"  "Brian"
```

Disclaimer: ages are rough estimates only . . .

Factors

Qualitative data that can assume only a discrete set of values are represented by a `factor`.

```
> trt <- factor(rep(c("Control","Treated"),c(3,4)))
> str(trt)
 Factor w/ 2 levels "Control","Treated": 1 1 1 2 2 2 2

> summary(trt)
Control Treated
      3       4
```

If the levels of a factor are numeric (e.g. the treatments are labelled “1”, “2”, and “3”) it is important to ensure that the data are actually stored as a factor and not as numeric data. Always check this by using `summary`.

Factors

If you have numeric data that should be a factor, use `factor` or `as.factor` to convert it to a factor.

```
> x <- c(0,1,1,02,1,0,2,1,2)
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.000   1.000   1.000   1.111   2.000   2.000

> x <- as.factor(x)
> summary(x)
0 1 2
2 4 3
```

Ordered Factors

An ordered factor is, not surprisingly, a special type of factor in which the levels have an ordering.

```
> pain <- ordered(c("Moderate","None","Severe","Severe","None"),
+                 levels = c("None", "Moderate", "Severe"))
> str(pain)
 Ord.factor w/ 3 levels "None"<"Moderate"<..: 2 1 3 3 1

> pain
[1] Moderate None     Severe   Severe   None
Levels: None < Moderate < Severe

> summary(pain)
    None Moderate   Severe
       2        1        2
```

Ordered Factors

```
> class(pain)
[1] "ordered" "factor"

> mode(pain)
[1] "numeric"

> typeof(pain)
[1] "integer"
```

You do not want the following:

```
> pain <- ordered(c("Moderate", "None", "Severe", "Severe", "None"))
> pain
[1] Moderate None     Severe   Severe   None
Levels: Moderate < None < Severe
```

Data frames

A `data.frame` is the basic S structure for a data set that can be represented as a set of observations (rows) on several variables (columns). Most of the data sets you see listed in the output of `data()` are data frames.

```
> data(Formaldehyde)
> str(Formaldehyde)
'data.frame':   6 obs. of  2 variables:
 $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
> summary(Formaldehyde)
      carb           optden
 Min.   :0.1000   Min.   :0.0860
 1st Qu.:0.3500   1st Qu.:0.3132
 Median :0.5500   Median :0.4920
 Mean   :0.5167   Mean   :0.4578
 3rd Qu.:0.6750   3rd Qu.:0.6040
 Max.   :0.9000   Max.   :0.7820
```

Data frames

Columns in data frames are usually numeric variables or factors. (Other possibilities exist but are rare.) Always check a data frame using `summary` to ensure that variables that should be factors are factors. Factors are summarized by frequency tables.

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

The general subset operator

The '`[`' operator is the general extraction operator. It creates an object of the same mode as the object to which it is applied. In the expression `x[i]` several forms of indices `i` can be used:

positive integers:
indicate the positions of the elements to extract. The first position is numbered 1.

negative integers:
indicate all elements except those at indices numbered `-i`. That is, `x[-1]` means "drop the first element of `x`".

logical vectors:
if `i` is a logical vector of the same length as `x` then the elements of `x` corresponding to `TRUE` in `i` are returned.

character variables:
are matched against the names of elements of `x`.

Examples of the general subset

```
> rn <- rnorm(100)
> rn[1:3]
[1] -1.52057659 -0.29059035 -0.08113082
> rn[3:1]
[1] -0.08113082 -0.29059035 -1.52057659

> rn[98:100]
[1] 0.2454289 0.1317154 1.1490626
> rn[-(1:97)]
[1] 0.2454289 0.1317154 1.1490626

> str(rn[rn>0])
 num [1:53] 0.254 0.776 1.323 0.239 0.510 ...

> con <- c(e=exp(1),pi=pi,twopi=2*pi)
> con[c("e","pi")]
       e       pi
2.718282 3.141593
```

Extracting single elements

The '[' operator returns an object of the same mode as the object to which it is applied. The '[[' and '$' operators extract single elements in their native mode. The distinction is like that between "an element of a set" (what '[[' produces) and "a subset of size 1 from a set", (what '[' produces).

```
> li <- list(pi=pi,e=exp(1))
> mode(li[1])
[1] "list"
> mode(li[[1]])
[1] "numeric"
> li[1]
$pi
[1] 3.141593

> sqrt(li[1])
Error in sqrt(li[1]) : Non-numeric argument to mathematical function
> sqrt(li[[1]])
[1] 1.772454
```

Subsets applied to data frames

Data frames are most naturally regarded as a rectangular structure. We can use '[' to extract subsets of rows or subsets of columns or both. For this, two indexing expressions are used. Omitting an indexing expression for the rows (or columns) means to use all the rows (or columns).

```
> dim(iris)
[1] 150   5
> summary(iris[,c(1,2,5)])
  Sepal.Length    Sepal.Width          Species
 Min.   :4.300   Min.   :2.000   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   versicolor:50
 Median :5.800   Median :3.000   virginica :50
 Mean   :5.843   Mean   :3.057
 3rd Qu.:6.400   3rd Qu.:3.300
 Max.   :7.900   Max.   :4.400
> dim(iris[iris$Species=="setosa",])
[1] 50  5
```

Subsets that are larger than the original

The extraction operator '[' is more general than a subset operator. By repeating indices we can produce "subsets" that are larger than the original.

```
> c("Yes","No")
[1] "Yes" "No"

> rep(c("Yes","No"),3)
[1] "Yes" "No"  "Yes" "No"  "Yes" "No"

> rep(1:2,3)
[1] 1 2 1 2 1 2

> c("Yes","No")[rep(1:2,3)]
[1] "Yes" "No"  "Yes" "No"  "Yes" "No"

> LETTERS[c(11,18,18,9,2)]
[1] "K" "R" "R" "I" "B"
```

NA - the missing data marker

The codes NA (not available) and NaN (not a number) indicates a missing data value in a vector or other data structure. Both are called NA's. An NA can be part of the original data, or it can be the result of operations on other data where the result is undefined, or it can be assigned.

```
> rrn <- rnorm(100)
> lrn <- log(rrn)
Warning message:
NaNs produced in: log(x)
> str(rrn)
 num [1:100]  0.198  0.261  1.647  1.679 -2.463 ...
> str(lrn)
 num [1:100] -1.617 -1.344  0.499  0.518    NaN ...

> NA & TRUE
[1] NA
> NA | TRUE
[1] TRUE
```

Use is.na to check for missing data

Note that we check for missing data with `is.na`. This is the only way to detect missing data. A common mistake is trying to check for missing data with expressions like `x == NA`. This doesn't work as expected. Missing values propagate in operations, including comparison operations. Comparing another value to NA always produces an NA.

```
> str(1 + lrn)
 num [1:100] -0.617 -0.344  1.499  1.518    NaN ...

> lrn.msng <- lrn == NA
> str(lrn.msng)
 logi [1:100] NA NA NA NA NA NA NA NA NA NA NA NA ...

> lrn.msng <- is.na(lrn)
> str(lrn.msng)
 logi [1:100] FALSE FALSE FALSE FALSE  TRUE  TRUE ...
```

Summaries of data that have NA's

Applying a summary function, such as `mean`, `median`, or `var` to data with any `NA`'s (or `NaN`'s) will return `NA` (or `NaN`).

If you want the value of the summary function after excluding the NA's, you must exclude the NA's then do the summary. Several summary functions allow an argument `na.rm = TRUE` that causes this to be done automatically.

```
> mean(lrn)
[1] NaN

> mean(lrn[!is.na(lrn)])
[1] -0.5095632

> mean(lrn,na.rm=TRUE)
[1] -0.5095632
```

Other special numeric values

NA's are allowed in all types of data. Numeric data also allows `NaN`, as shown previously, `Inf` (∞) and `-Inf` ($-\infty$).

```
> log(0:2)
[1]      -Inf 0.0000000 0.6931472
> exp(log(0:2))
[1] 0 1 2
```

There are ways of detecting `NaN` and infinite values.

```
> x=rnorm(5)
> x
[1]  1.0847354 -0.2244801 -0.3103911 -0.6022185  0.5310318

> y=log(x)
Warning message:
NaNs produced in: log(x)

> is.nan(y)
[1] FALSE  TRUE  TRUE  TRUE FALSE
```