# Input/Output
# Data Frames

Statistics 135

Autumn 2005

# Input/Output

## Importing text files

- Rectangular ($n$ rows, $c$ columns)

  Usually you want to use `read.table`

```
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 = "#")
```

  The arguments you normally are going to want to deal with are `file`, `header`, and `sep`.

`file`:

The name of the file to be read in

`header`:

a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: 'header' is set to 'TRUE' if and only if the first row contains one fewer field than the number of columns.

`sep`:

the field separator character. Values on each line of the file are separated by this character. If '`sep = ""`' (the default for `read.table`) the separator is "white space", that is one or more spaces, tabs, newlines or carriage returns.

An example use of this function was seen during introductory example on **S**.

```
cars93.df <- read.table("93cars.txt", header=T, row.names=NULL)
```

Other similar functions are `read.csv` or `read.csv2` (comma separated values) and `read.delim` or `read.delim` (tab delimited field) (as defaults)

Each of these functions produces a data frame, a special type of list. The resulting data frame consists of $c$ vectors, each of length $n$.

- Text, but arbitrary structure

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
       quote = if(identical(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 = "",
       allowEscapes = TRUE)
```

The important parameters are `file`, `what`, `sep`, and `flush`.

`what`:

the type of 'what' gives the type of data to be read. If 'what' is a list, it is assumed that the lines of the data file are records each containing 'length(what)' items ("fields"). The supported types are 'logical', 'integer', 'numeric', 'complex', 'character', 'raw' and 'list': 'list' values should have elements which are one of the first six types listed or 'NULL'.

`sep`:

by default, scan expects to read white-space delimited input fields. Alternatively, 'sep' can be used to specify a character which delimits fields. A field is always delimited by an end-of-line marker unless it is quoted.
If specified this should be the empty character string (the default) or 'NULL' or a character string containing just one single-byte character.

`flush`:

logical: if 'TRUE', 'scan' will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field, but precludes putting more that one record on a line.

## Importing non-text files

With the `foreign` package, files from other statistic packages. These packages include **S-Plus**, **SAS** (SAS transport files), **SPSS**, **Minitab** (Minitab portable worksheets), **Stata**, **Systat**, and **EpiInfo**.

For information on these routines, and other Imput/Output routines, see the R Data Input/Output pdf file, available under the Help menu in **R**.

## Importing Excel files

Best approach – don't do it in **R**. Convert the data you want in Excel into either a CSV file or a tab delimited file and read in with `read.table`, `read.csv`, or `read.delim`

## Exporting

Usually the best approach is to export the data you want into a text format (CVS, space, or tab delimited). This can be done with the function `write.table`, `write.csv`, or `write.csv`

```
write.table(x, file = "", append = FALSE, quote = TRUE,
            sep = " ", eol = "\n", na = "NA", dec = ".",
            row.names = TRUE, col.names = TRUE,
            qmethod = c("escape", "double"))
```

The important options are `x`, `file`, `sep`, and `quote`

`x`:

> the object to be written, preferably a matrix or data frame. If not, it is attempted to coerce 'x' to a data frame.

`quote`:

> a logical value or a numeric vector. If 'TRUE', any character or factor columns will be surrounded by double quotes. If a numeric vector, its elements are taken as the indices of the columns to quote. In both cases, row and column names are quoted if they are written. If 'FALSE', nothing is quoted.

An example is

```
write.table(cars.df,"cars.txt",sep="\t")
```

This will write the data frame `cars.df` into the file `cars.txt` (which is in the same directory as the `.Rdata` file) as a tab delimited file. The first few lines look like

```
"Manu"  "Model" "Cylinder"  "Type"  "EngSize"   "Weight"    "CityMP
"1" "Acura" "Integra"   "4" "Small" 1.8 2705    25  31  4   3.22580
"2" "Acura" "Legend"    "6" "Midsize"   3.2 3560    18  25  5.55555
"3" "Audi"  "90"    "6" "Compact"   2.8 3375    20  26  5   3.84615
"4" "Audi"  "100"   "6" "Midsize"   2.8 3405    19  26  5.26315789
"5" "BMW"   "535i"  "4" "Midsize"   3.5 3640    22  30  4.54545454
```

Personally I prefer to set `quote=F` and set `sep="\t"` as it tends to be easier to read the file into other programs like **Excel**. You may need to bit careful if character strings may contain tabs (which is the reason for `sep="\t"`).

Also if there aren't informative row names, the option `row.names=F` is usually desirable.

If you need to export data in a format that doesn't fit into an $n$ row by $c$ column format, use the `write` function instead.

```
write(x, file = "data",
            ncolumns = if(is.character(x)) 1 else 5,
            append = FALSE)
```

# Data Frames

As mentioned before, a data frame is a special type of list. When dealing with data, it is often the most useful way of storing your data. There are a number of reasons for this

- Keeps a data set together in a single object
- It is the expected input for many modeling functions
- Great flexibility. Components can be accessed by a number of different ways.

```
> test.df <- data.frame(norm=rnorm(5,0,2), chi2=rchisq(5,2),
+                 let=c("a","b","c","d","e"))

> names(test.df)
[1] "norm" "chi2" "let"
```

```
> test.df
        norm       chi2 let
1 -0.8376633 0.2120354   a
2 -1.2233754 5.3120881   b
3 -2.1004727 2.2560415   c
4  1.8781837 1.8405489   d
5 -0.8908000 0.2286858   e
```

Since a data frame is also a list, the standard approach for accessing components of a list will work

```
> test.df$norm
[1] -0.8376633 -1.2233754 -2.1004727  1.8781837 -0.8908000
```

In addition, due to the rectangular structure, components can also be accessed as they are a matrix

```
> test.df[,1]
[1] -0.8376633 -1.2233754 -2.1004727  1.8781837 -0.8908000
```

```
> test.df[,"norm"]
[1] -0.8376633 -1.2233754 -2.1004727  1.8781837 -0.8908000

> test.df[1:2,]
        norm      chi2 let
1 -0.8376633 0.2120354   a
2 -1.2233754 5.3120881   b
```

## Attaching a data frame

In a normal **R** session, there are more objects available than what is in the `.Rdata`. These include required packages loaded at startup plus optional packages and data frames. The current search path can be shown with the `search` function

```
> search()
 [1] ".GlobalEnv"        "package:RWinEdt"   "package:methods"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
```

```
 [7] "package:utils"     "package:datasets"  "Autoloads"
[10] "package:base"
```

By adding a data frame to the search path it can be easier to access the variables in the data frame. The data frame can be added with the `attach` function

```
attach(what, pos = 2, name = deparse(substitute(what)))
```

```
Arguments:

    what: "database".  This may currently be a 'data.frame' or
          'list' or a R data file created with 'save'.

     pos: integer specifying position in 'search()' where to
          attach.

    name: alternative way to specify the database to be attached.
```

Note that strictly, the data frame isn't attach, but a copy is made and

added to the search path.

```
> chi2
Error: Object "chi2" not found

> attach(test.df)

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

> chi2
[1] 0.2120354 5.3120881 2.2560415 1.8405489 0.2286858
```

```
> ls()     # objects in the first element of the search path
 [1] "cars.df"      "cars93"       "cityfuel.lm"  "cityfuelt.lm"
 [5] "citympg.lm"   "data"         "example.l"    "ind"
 [9] "last.warning"

> chi2 <- chi2 ^ 2
> ls()

 [1] "cars.df"      "cars93"       "chi2"         "cityfuel.lm"
 [5] "cityfuelt.lm" "citympg.lm"   "data"         "example.l"
 [9] "ind"          "last.warning"

> chi2
[1] 0.04495900 28.21828025  5.08972343  3.38762027  0.05229717

> test.df$chi2
[1] 0.2120354 5.3120881 2.2560415 1.8405489 0.2286858
```

Note that when assignments like the previous are done, they are put in the `.GlobalEnv` (position 1). They do not replace objects later in the search path. If you wish to replace objects later in the search path, you need to use the `assign` function or the <<- operator.

When multiple objects have the search path, the one highest in the search path is the one accessed.

If you need to access an object in a particular location of the search path use the `get` function

```
> chi2
[1] 0.04495900 28.21828025  5.08972343  3.38762027  0.05229717

> get("chi2",2)
[1] 0.2120354 5.3120881 2.2560415 1.8405489 0.2286858
```

To remove an environment from the search path, use the `detach` function

```
detach(name, pos = 2, version)
```

If the call is done by `detach(n)`, where `n` is a number, the $n$th item from the search path is deleted.

Note that if items are altered in an attached data frame, the changes are not kept during after detaching the data frame

```
> chi2 <<- chi2 ^ 2
> ls()
 [1] "a_b"           "cars.df"      "cars93"        "cityfuel.lm"
 [5] "cityfuelt.lm" "citympg.lm"   "data"          "example.l"
 [9] "ind"           "last.warning"
```

```
> chi2
[1]  0.04495900 28.21828025  5.08972343  3.38762027  0.05229717

> get("chi2",2)
[1]  0.04495900 28.21828025  5.08972343  3.38762027  0.05229717

> detach(2)
> test.df$chi2
[1] 0.2120354 5.3120881 2.2560415 1.8405489 0.2286858
```

So attaching a data frame to alter it isn't a good of doing it. (You used to be able to do it, but it now appears to be depreciated.)

To alter, or add objects, it is better to mimic the following

```
> test.df$chi2
[1] 0.2120354 5.3120881 2.2560415 1.8405489 0.2286858
> test.df$chi2 <- test.df$chi2 + 2
> test.df$new <- 1:5
> test.df
        norm     chi2 let new
1 -0.8376633 2.212035   a   1
2 -1.2233754 7.312088   b   2
3 -2.1004727 4.256042   c   3
4  1.8781837 3.840549   d   4
5 -0.8908000 2.228686   e   5
```

When a data frame is attached, it is still possible to access the original version as follows

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

> summary(test.df)
      norm               chi2        let
 Min.   :-2.1005   Min.   :2.212   a:1
 1st Qu.:-1.2234   1st Qu.:2.229   b:1
 Median :-0.8908   Median :3.841   c:1
 Mean   :-0.6348   Mean   :3.970   d:1
 3rd Qu.:-0.8377   3rd Qu.:4.256   e:1
 Max.   : 1.8782   Max.   :7.312
```