Part 3: Data Import/Export

140.776 Statistical Computing

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.

Data Export

The function `cat` underlies the functions for exporting data. It takes a `file` argument, and the `append` argument allows a text file to be written via successive calls to `cat`.

```
> cat("Good morning!","\n")
Good morning!

> maxit <- 10
> for (j in 1:maxit){
+ if (j==1) cat("Iteration: ")
+ cat(j,".. ")
+ if (j==maxit) cat("Done!","\n")}
Iteration: 1 .. 2 .. 3 .. 4 .. 5 .. 6 .. 7 .. 8 .. 9 .. 10 .. Done!

> ff <- tempfile()
> cat(file=ff, "123456", "987654", sep="\n")
> ff
[1] "/tmp/Rtmp1808/file4db127f8"
```

Data Export

`print` prints its argument. It is a generic function which means that new printing methods can be easily added for new classes.

It is possible to use `sink` to divert the standard R output to a file, and thereby capture the output of (possibly implicit) `print` statements.

The commonest task is to write a matrix or data frame to file as a rectangular grid of numbers, possibly with row and column labels. This can be done by the functions `write.table` and `write`. Function `write` just writes out a matrix or vector in a specified number of columns (and transposes a matrix). Function `write.table` is more convenient, and writes out a data frame (or an object that can be coerced to a data frame) with row and column labels.

Exporting data frames

There are a number of issues that need to be considered in writing out a data frame to a text file:

- Precision
- Header line
- Separator
- Missing values
- Quoting strings

Note that `write.table` is often not the best way to write out very large matrices, for which it can use excessively large amounts of memory. Function `write.matrix` in package MASS provides specialized interface for writing matrices, with the option of writing them in blocks and thereby reducing memory usage.

write.table

```
>?write.table
```

```
write.table                package:base                R Documentation

Data Output

Description:

     ‘write.table’ prints its required argument ‘x’ (after converting
     it to a data frame if it is not one already) to ‘file’.  The
     entries in each line (row) are separated by the value of ‘sep’.

Usage:

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

write.table

To write a CSV file for input to Excel one might use

```
write.table(x, file = "foo.csv", sep = ",", col.names = NA)
```

and to read this file back into R one needs

```
read.table("file.csv", header = TRUE, sep = ",", row.names=1)
```

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!

`paste`

```
paste                   package:base                  R Documentation

Concatenate Strings

Description:
     Concatenate vectors after converting to character.

Usage:
     paste(..., sep = " ", collapse = NULL)

Arguments:
     ...: one or more R objects, to be coerced to character vectors.
     sep: a character string to separate the terms.
collapse: an optional character string to separate the results.
```

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

```
source                package:base                R Documentation

Read R Code from a File or a Connection

Description:
     ‘source’ causes R to accept its input from the named file (the
     name must be quoted). Input is read from that file until the end
     of the file is reached.   ‘parse’ is used to scan the expressions
     in, they are then evaluated sequentially in the chosen
     environment.

Usage:
     source(file, local = FALSE, echo = verbose, print.eval = echo,
            verbose = getOption("verbose"),  prompt.echo = getOption("
            max.deparse.length = 150, chdir = FALSE)

Arguments:
    file: a connection or a character string giving the name of the
          file or URL to read from.
```

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.

sink

```
sink                    package:base                    R Documentation

Send R Output to a File

Description:

     'sink' diverts R output to a connection.

     'sink.number()' reports how many diversions are in use.

     'sink.number(type = "message")' reports the number of the
     connection currently being used for error messages.

Usage:

     sink(file = NULL, append = FALSE, type = c("output", "message"))
     sink.number(type = c("output", "message"))
```

sink

```
make.tex <- function(xx,fl="output.tex",nms=FALSE,r.nms=FALSE){
  file <- fl
  sink(file)
  n1 <- dim(xx)[1];n2 <- dim(xx)[2]
  if(nms){
    if(r.nms) cat(" & ")
    for(k in 1:n2){
      cat(names(xx)[k])
      if(k<n2) cat(" & ")
        else cat(" \\\\\n")}}
  for(j in 1:n1){
    if(r.nms){
      cat(row.names(xx)[j])
      cat(" & ")}
    for(k in 1:n2){
      cat(xx[j,k])
      if(k<n2) cat(" & ")
        else cat(" \\\\\n")}}
  sink()
  invisible()}
```

sink

```
> source("make.tex.R")
> data(iris)
> make.tex(iris,nms=T,r.nms=T)
```

This creates a file output.tex that looks like this:

```
 & Sepal.Length & Sepal.Width & Petal.Length &
Petal.Width & Species \\
1 & 5.1 & 3.5 & 1.4 & 0.2 & 1 \\
2 & 4.9 & 3 & 1.4 & 0.2 & 1 \\
3 & 4.7 & 3.2 & 1.3 & 0.2 & 1 \\
4 & 4.6 & 3.1 & 1.5 & 0.2 & 1 \\
5 & 5 & 3.6 & 1.4 & 0.2 & 1 \\
6 & 5.4 & 3.9 & 1.7 & 0.4 & 1 \\
...
```

which can be included in the tabular environment of a LaTeX document using:

```
\input{output}
```

Storing data

Every R object can be stored into and restored from a file with the commands `save` and `load`. This uses the external data representation (XDR) standard of Sun Microsystems and others, and is portable between MS-Windows, Unix, Mac.

```
> x <- 1:4
> save(x,file="x.Rdata")
> rm(x)
> x
Error: Object "x" not found
> load("x.Rdata")
> x
[1] 1 2 3 4
```

Moreover, the command

```
load("/home/ingo/.RData")
```

loads all R objects stored in my home directory.

Importing from other statistical systems

The package foreign provides import facilities for files produced by

- Minitab `read.mtp`,
- S-PLUS `read.S`,
- SAS `read.xport`, `read.ssd`,
- SPSS `read.spss`,

and export and import facilities for

- Stata `read.dta`, `write.dta`.

Additional information

- Several recent packages allow functionality developed in languages such as Java, Perl and Python to be directly integrated with R code (see packages `Java`, `RSPerl`, `RSPython`).
- There are several packages available on CRAN to help R communicate with Database Management Systems (see packages `DBI`, `RMySQL`, `RPgSQL`, `RmSQL`, `RODBC`).
- Some limited facilities are available to exchange data across network connections.

See CRAN webpage http://cran.r-project.org/ and the page of the Omegahat project http://www.omegahat.org/ for details.