S Basics

Statistics 135

Autumn 2005

S Basics

When discussing the **S** environment, I will, or at least try to, make the following distinctions.

- **S** will be used when what is being discussed will work with **R** or **S-Plus**.
- **R** will be used when discussing **R** specific features
- **S-Plus** will be used when discussing **S-Plus** specific features

Starting R

- Windows
  - Select **R** from the Start menu
  - or double-click `.Rdata` file in desired directory
- Macintosh
  - Select **R** from the Start menu
  - or double-click `.Rdata` file in desired directory
- Unix
  - Change to the desired directory
  - Type `R <return>`
  - For on-screen graphics, you must be running under an Xwindows setup. Logging into `ice, stat`, etc with Secure CRT will not allow graphics to be displayed on the screen. Graphics can be saved in various file formats (which available depends on platform).

```
R : Copyright 2005, The R Foundation for Statistical Computing
Version 2.1.1  (2005-06-20), ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY. You
are welcome to redistribute it under certain conditions. Type
'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors. Type
'contributors()' for more information and 'citation()' on how
to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help. Type 'q()'
to quit R.

[Previously saved workspace restored]
```

Quitting R

In any platform, type `q()`. You will be asked if you wish to save your workspace. In unix/linux, it looks like

```
> q()
Save workspace image? [y/n/c]:
```

If you say yes, the final version of any objects will be saved. If you say no, any changes made during the session will **not** be saved. Cancel will stop quitting the program and return you to your session.

Note that the brackets in `q()` are important.

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

If they are left out, you see the code for the function `q`.

With Windows and Mac, you can use the standard menu approaches to quitting **R** as well.

Similar approaches to quitting in **S-Plus** work, however there is one big difference, you will not be asked whether you want to save your changes.

The reason for this difference will be discussed later today.

Example S Session

Arithmetic:

```
> 4 + 5 / 3           # addition & division
[1] 5.666667
> (4 + 5) / 3         # using parentheses
[1] 3
> log(10)             # natural logarithm
[1] 2.302585
```

S uses the standard order of operation rules (BODMAS).

Output from an *expression* can be *assigned* to an object for later use. Assignments are done with the <- operator.

```
> xbar <- (4 + 6 + 8 + 10)/4   # assigns avg to xbar
> xbar                         # display the object xbar
[1] 7
> s <- sqrt(((-3)^2 + (-1)^2 + 1^2 + 3^2)/3)
```

```
> s
[1] 2.581989
> z <- (xbar - 10)/s  # test statistic for normal z test
> z
[1] -1.161895
```

In this example, the value of expressions were assigned to objects and used one of the *functions* (`sqrt`) available in **S**.

The # is used as the comment character. Anything after this character on a line will not be processed as part of a command.

Alternative assignment operators (recommended that they not be used)

- **R**: =
- **S-Plus**: _ (underscore character)

To see (list) the objects in a directory, use the `ls()` command.

```
> ls()
[1] "crabs.grp" "lcrabs.pc" "s"          "sex"        "sp"
[6] "super.sym" "xbar"      "z"
```

To remove objects use the `rm()` function

```
> rm(s,z)
> ls()
[1] "crabs.grp" "lcrabs.pc" "sex"        "sp"         "super.sym"
[6] "xbar"
```

Objects in S

There are 3 basic types of objects in **S**: *arrays*, *lists*, and *functions*

- Arrays (numeric, logical, or character strings)

  These include vectors, matrices, and higher dimensional (up to 8) arrays.

  Lets start by looking at vectors

```
> data <- c(4,6,8,10)    # concatenating
> data                   # print data
[1]  4  6  8 10
> length(data)           # length of vector data
[1] 4
> data[3]                # 3rd element of vector data
[1] 8
> ind <- 2:3             # sequence
> ind
```

```
[1] 2 3
> data[ind]                 # subset vector
[1] 6 8
> data + 100
[1] 104 106 108 110
> sqrt(data)                # square root of components of vector
[1] 2.000000 2.449490 2.828427 3.162278
> sum(data)                 # sum of elements of vector
[1] 28
```

Using vectors makes the earlier example much clearer

```
> xbar <- mean(data)
> s <- sqrt(sum((data - xbar)^2) / (length(data)-1))
> s.easier <- sqrt(var(data))
> s
[1] 2.581989
> s.easier
[1] 2.581989
```

```
> z <- (xbar - 10)/s
> z
[1] -1.161895
```

Matrices work in a similar fashion to vectors

```
> mat <- cbind(id=1:5, iq=rnorm(5, mean=100, sd=15))
> mat
     id        iq
[1,]  1 133.24186
[2,]  2 118.64908
[3,]  3  94.64872
[4,]  4  94.49125
[5,]  5 107.53032

> dim(mat)            # dimensions of matrix mat
[1] 5 2
```

```
> dimnames(mat)       # row and column names
[[1]] NULL

[[2]] [1] "id" "iq"

> mat[1:2,]           # the first two rows of mat
     id       iq
[1,]  1 133.2419
[2,]  2 118.6491

> mat[,2]       # 2nd column of mat, indicated by number
[1] 133.24186 118.64908  94.64872  94.49125 107.53032

> mat[,'iq']    # 2nd column of mat, indicated by column name
[1] 133.24186 118.64908  94.64872  94.49125 107.53032
```

```
> mat.2 <- mat[-5,]  # remove last line
> mat.2
     id        iq
[1,]  1 133.24186
[2,]  2 118.64908
[3,]  3  94.64872
[4,]  4  94.49125

> ind <- mat[,'iq'] > 115   # IQ's > 115
> ind                       # vector of logicals (True/False)
[1]  TRUE  TRUE FALSE FALSE FALSE

> mat.3 <- mat[ind,]        # part of mat with IQ's > 115
> mat.3
     id       iq
[1,]  1 133.2419
[2,]  2 118.6491
```

Other logical operators for direct comparison are < (less than), == (equal to), <= (less or equal), >= (greater or equal), != (not equal). Also can use & (AND), | (OR), and ! (NOT).

```
> mat[!((mat[,2]>85) & (mat[,2] < 115)),]
     id       iq
[1,]  1 133.2419
[2,]  2 118.6491
```

- Lists

  Lists can store different different types of data in a single object

```
> example.l
$id [1] 1 2 3 4

$data [1]  4  6  8 10

$xbar [1] 7

$stdev [1] 2.581989

> names(example.l)     # the names of the objects in the list
[1] "id"    "data"  "xbar"  "stdev"
> example.l$data       # item data of the list
[1]  4  6  8 10
> example.l$data[2:3]  # elements 2 & 3 of data
[1] 6 8
```

Fitting into the object oriented paradigm, there are lists with special structures. One example is the output of the linear model (`lm`) function. It returns an object of class `lm`, which is a list with a special structure.

```
> class(cityfuel.lm)
[1] "lm"

> is.list(cityfuel.lm)
[1] TRUE

> names(cityfuel.lm)
 [1] "coefficients"  "residuals"  "effects"    "rank"
 [5] "fitted.values" "assign"     "qr"         "df.residual"
 [9] "xlevels"       "call"       "terms"      "model"

> names(cityfuel.lm$qr)
[1] "qr"    "qraux" "pivot" "tol"   "rank"
```

- Functions

  There are many *functions* built into **S**. A few examples are

  - Arithmetical: `abs`, `log`, `sqrt`, `sin`, `asin`
  - Matrix: `t` (matrix transpose), `%*%` (matrix multiply), `solve`, `qr` (qr decomposition)
  - Statistical: `mean`, `median`, `var`, `quantile`, `summary`, `lm` (linear model)
  - Distributions: `dnorm`, `pnorm`, `qnorm`, and `rnorm` are the density, cdf, quantile and random number functions for the normal distribution. Also available for other common distributions, e.g, `dt` (t), `df` (F), `dchisq` (χ^2), `dgamma` (Gamma), `dexp` (Exponential), `dbinom` (Binomial), etc

You can also write your own functions in **S**.

Here is an example for calculating the t statistic for a one-sample t-test.

```
> my.ttest <- function(x, null = 0) {
+   n <- length(x)
+   xbar <- mean(x)
+   sd <- sqrt(var(x))
+   se <- sd/sqrt(n)
+   tstat <- (xbar - null) / se
+   tstat
+ }
> my.ttest(data)
[1] 5.422177
```

Note that this function isn't needed as one already exists.

```
> t.test(data)

        One Sample t-test

data:  data
t = 5.4222, df = 3, p-value = 0.01231
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
  2.891479 11.108521
sample estimates:
mean of x
        7
```

To see what a function does, type the name of the function without the brackets

```
> var
function (x, y = NULL, na.rm = FALSE, use) {
    if (missing(use))
        use <- if (na.rm)
            "complete.obs"
        else "all.obs"
    na.method <- pmatch(use, c("all.obs", "complete.obs",
                               "pairwise.complete.obs"))
    if (is.data.frame(x))
        x <- as.matrix(x)
    else stopifnot(is.atomic(x))
    if (is.data.frame(y))
        y <- as.matrix(y)
    else stopifnot(is.atomic(y))
    .Internal(cov(x, y, na.method, FALSE))
}
<environment: namespace:stats>
```

This may not always be informative however as can be seen with

```
> t.test
function (x, ...)
UseMethod("t.test")
<environment: namespace:stats>

> methods(t.test)
[1] t.test.default* t.test.formula*

   Non-visible functions are asterisked
```

To figure out what is going on here, you probably have to get into the **C** source code, which can be downloaded.

Getting Help in S

Getting help in **S** is easy with the built-in help system. Help for any function can be gotten by `help(command)` or `?command`

```
> ?mean

mean                    package:base                    R Documentation

Arithmetic Mean

Description:

     Generic function for the (trimmed) arithmetic mean.

Usage:

     mean(x, ...)
```

```
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:

      x: An R object.  Currently there are methods for numeric
         data frames, numeric vectors and dates.  A complex
         vector is allowed for 'trim = 0', only.

   trim: the fraction (0 to 0.5) of observations to be trimmed
         from each end of 'x' before the mean is computed.

  na.rm: a logical value indicating whether 'NA' values should
         be stripped before the computation proceeds.

    ...: further arguments passed to or from other methods.

Value:
```

```
     For a data frame, a named vector with the appropriate
     method being applied column by column.

     If 'trim' is zero (the default), the arithmetic mean of the
     values in 'x' is computed.

     If 'trim' is non-zero, a symmetrically trimmed mean is
     computed with a fraction of 'trim' observations deleted
     from each end before the mean is computed.

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The
     New S Language_. Wadsworth & Brooks/Cole.

See Also:

     'weighted.mean', 'mean.POSIXct'
```

```
Examples:

      x <- c(0:10, 50)
      xm <- mean(x)
      c(xm, mean(x, trim = 0.10))

      mean(USArrests, trim = 0.2)
```

If you wish to see just the possible arguments to a function, use the `args` function.

```
> args(var)
function (x, y = NULL, na.rm = FALSE, use)
NULL
```

Due to the object oriented structure, you may need to do a bit of work to get the `args` function to give you what you want

```
> args(mean)
function (x, ...)
NULL

> mean
function (x, ...)
UseMethod("mean")
<environment: namespace:base>

> args(mean.default)
function (x, trim = 0, na.rm = FALSE, ...)
NULL

> args(mean.Date)
function (x, ...)
NULL
```

Another option for help is the web based help system, which can be started with `help.start()`.

In the Windows (and Mac ?) versions, the Help menu has many resources.

Differences between R and S-Plus

See Section 15.3 of Krause and Olson for a further discussion

- Source code

  It is possible to get the **R** source code, at least for the main program. This is **not** available for **S-Plus**.

- How objects are stored

  – In **R**, all objects are stored in a file with the name `.Rdata`
  – In **S-Plus**, all objects are stored in a directory `.Data` as separate files

- Memory handling

  In **R**, data are kept in memory until explicitly written to file (using function `save.image`). In **S-Plus**, objects are stored after successful completion of a command. So if **R** crashes, you can lose you work (so save workspace every so often).

- Evaluation (scoping) rules

  – **R** uses lexical scoping
  – **S-Plus** use static scoping

  This has to do with how each program deals with objects inside functions

- Function options

  Functions may take different options in the 2 packages. For example, `pnorm`, the normal CDF function is different.

  – **R**: `pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)`
  – **S-Plus**: `pnorm(q, mean=0, sd=1)`

  The **R** version can deal with lower and upper (sometimes called the survivor function) tails probabilities and log probabilities.

  This has implications if you wish to distribute your code.

- Mathematical notation in graphs

  - Relatively easy in **R**.
  - Difficult in **S-Plus**. There is a package written by Alan Zaslavsky (Health Care Policy) which helps.

- Data Import and Export

  S-Plus can handle more formats