Programming in S

Statistics 135

Autumn 2005

Writing Functions in S

As we have seen during the term, **S** is a full featured, object-oriented programming language. Previously we have discussed writing scripts and running them within the language. Similarly it is easy to write your own functions in **S**.

The general structure of a function is

```
function.name <- function(args) {

  command 1
    . . .
  command n

  function.output
}
```

The last object mentioned in the function is what is returned by the function. However don't have it as part of an assignment.

Function Arguments

As with any high level language, you need to give arguments to your function. Well sort of.

The argument list will usually be of the form

```
arg1, arg2, arg3 = default, arg4 = default, ...
```

As we've seen in the past, it is possible to give some arguments default values, as has been done with `arg3` and `arg4`. The first two arguments, `arg1` and `arg2`, since they do not have default settings, they **must** be given.

In addition it is possible to pass in extra arguments that do not need to be specified ahead of time with '`...`'.

The order that the arguments are listed in is the order expected when the function is called. For example

```
testfun <- function(x, y)
  { x / y }

> testfun(1, 2)
[1] 0.5

> testfun(2, 1)
[1] 2

> testfun(y=2, x=1)
[1] 0.5

> testfun(1)
Error in testfun(1) : argument "y" is missing, with no default
```

```
testfun2 <- function(x, y = 1, z)
   { x * y + z}

> testfun2(2, 2, 4)
[1] 8

> testfun2(2, z=4)   # y takes default value of 1
[1] 6

> testfun2(2, , 4)   # not recommended
[1] 6
```

Local vs Global Variables

Most of the time, variables inside functions are treated locally. That is, assignments made inside the function do not affect what is stored in your workspace.

```
succ <- function(n) {
  n <- n + 1
  n
  }

> n
[1] 93

> succ(n)
[1] 94

> n
[1] 93
```

```
succ2 <- function(n) {n <- n+1}

> n
[1] 93

> succ2(n)       # no output is produced
> n
[1] 93
```

When a function is called and it comes across something that hasn't be given as an argument or defined earlier in the function, it will go through the search path until it finds the object.

```
> n <- 2

testfun3 <- function(x, y = 1, z)
   { x * y + z * n}

> testfun3(2, z=4)
[1] 10
```

While it does have its uses, doing this can be dangerous and it is usually not recommended. Passing the values in as arguments is usually the way to go.

In addition, it is possible for assignments not to be local to the function, but global. You can reassign values in your workspace from within a function as follows

```
succ3 <- function(n) {
   x <- n + 1
   n <<- x
   x  }
```

```
> n
[1] 2
```

```
> succ3(n)
[1] 3
> n
[1] 3
```

This can be very dangerous.

DO NOT DO THIS !!!!!!

Especially with any functions you might pass onto somebody else. You might end up trashing some object you need without realizing it.

Control Structures (`for`, `while`, etc)

The standard control structures in most high level languages are available in **S**. These include `if` statements, `for` loops, `while` loops, etc.

- `if`: The basic structure is

```
if (condition)
  { true branch commands }
else
  { false branch commands }
```

  For example

```
fact2 <- function(n) {
  if (n != trunc(n))
    { stop("n is not an integer") }
  else
    { fact <- prod(1:n) }
  fact
}

> fact2(3)
[1] 6

> fact2(3.5)
Error in fact2(3.5) : n is not an integer
```

In the `if` statement, the condition should be a single logical value. If you are dealing with vectors, you may not get what you expect. An alternative in this case is `ifelse`. For example

```
> y <- -1:4

> ylogy <- ifelse(y <= 0, 0, y*log(y))
Warning message: NaNs produced in: log(x)

> ylogy
[1] 0.000000 0.000000 0.000000 1.386294 3.295837 5.545177
```

- `switch`:

  When there are more than 2 conditions that you need to deal with, as in

```
centre <- function(x, type) {
  if (type == "mean") result <- mean(x)
  else {
    if (type == "median") result <- median(x)
      else result <- mean(x, trim = 0.1)
  }
  result
}
```

  it may be easier to deal with as

```
centre <- function(x, type) {
   switch(type,
     mean = mean(x),
     median = median(x),
     trimmed = mean(x, trim = .1))
 }

>       x <- rcauchy(10)

>       centre(x, "mean")
[1] 0.6201826

>       centre(x, "median")
[1] 0.5793802

>       centre(x, "trimmed")
[1] 0.5566397
```

- `for`: The basic structure is

```
for ( variable in sequence) commands
```

For example,

```
fact1 <- function(n) {
  fact <- 1
  for (i in 1:n)
    fact <- fact * i
  fact
}

> fact1(10)
[1] 3628800
```

Note that the `sequence` doesn't need to be a numeric vector. It could be a vector of strings or more interestingly a list or data frame. For example

```
> for (i in crabs)
+   print(summary(i))
  B   O
100 100
  F   M
100 100
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0    13.0    25.5    25.5    38.0    50.0
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   7.20   12.90   15.55   15.58   18.05   23.10
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6.50   11.00   12.80   12.74   14.30   20.20
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.70   27.28   32.10   32.11   37.23   47.60
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  17.10   31.50   36.80   36.41   42.00   54.60
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
   6.10   11.40   13.90   14.03   16.60   21.60
> names(crabs)
[1] "sp"    "sex"   "index" "FL"    "RW"    "CL"    "CW"    "BD"
> summary(crabs)
 sp      sex          index             FL               RW               CL
 B:100   F:100   Min.   : 1.0   Min.   : 7.20   Min.   : 6.50   Min.   :14.70
 O:100   M:100   1st Qu.:13.0   1st Qu.:12.90   1st Qu.:11.00   1st Qu.:27.27
                 Median :25.5   Median :15.55   Median :12.80   Median :32.10
                 Mean   :25.5   Mean   :15.58   Mean   :12.74   Mean   :32.11
                 3rd Qu.:38.0   3rd Qu.:18.05   3rd Qu.:14.30   3rd Qu.:37.23
                 Max.   :50.0   Max.   :23.10   Max.   :20.20   Max.   :47.60
       CW              BD
 Min.   :17.10   Min.   : 6.10
 1st Qu.:31.50   1st Qu.:11.40
 Median :36.80   Median :13.90
 Mean   :36.41   Mean   :14.03
 3rd Qu.:42.00   3rd Qu.:16.60
 Max.   :54.60   Max.   :21.60
```

- `while`: The basic structure is

```
while (condition) commands
```

The `while` structure will keep repeating commands until a certain condition is met.

For example

```
fact3 <- function(n) {
  fact <- 1; i <- 1
  while ( i != n ) {
    i <- i + 1
    fact <- fact * i
  }
  fact
}
```

```
> fact3(5)
[1] 120

> fact3(5.5)
```

If I hadn't stopped the command in the second example, it would still be running. You need to be careful in setting your conditions in `while` statements or otherwise you could create an infinite loop.

- `repeat`:

  This is similar to `while` but you kick out of the loop in a slightly different way.

  The basic structure is

```
repeat commands
```

  However to break out of the loop, you need to have a `break` command. For example

```
fact4 <- function(n) {
  fact <- 1; i <- 1
  repeat {
    i <- i + 1
    fact <- fact * i
    if (i == n) break
  }
  fact  }
```

```
> fact4(6)
[1] 720

> fact4(6.5)
```

In this example, the loop keeps repeating until `i == n`. A similar problems occurs with this function if a non integer in input

If you are going to use equality constraints to stop loops you need to be careful. Consider the following example

```
> exp(3)/exp(1) == exp(2)
[1] TRUE

> exp(102)/exp(100) == exp(2)
[1] FALSE

> exp(2) - exp(3)/exp(1)
[1] 0
```

```
> exp(2) - exp(102)/exp(100)
[1] 8.881784e-16
```

All numbers in **S** are stored as double precision numbers, which gives 16 or 17 significant digits. So as calculations procedure, small errors can kick in.

```
> print((sqrt(2))^2, digits=17)
[1] 2.0000000000000004
```

```
> print(sqrt(2^2), digits=17)
[1] 2
```

So checking to see if two numbers are the same, while if you had infinite precision you would be fine, you can make mistakes in **S** (or C, Fortran, Pascal, etc).

Usually you want do see if two numbers differ from a small amount, usually based on the machine precision.