Programming in S

Functions

Vectorized calculations vs loops

As we've seen S is a full featured, object-oriented programming language. Previously I've shown how you can write scripts and running them from within S-Plus and R. Similarly, it is easy to write your own functions in S-Plus/R

General structure of a function

```
function.name <- function(args) {
  commands
  function.output
}
```

Function arguments

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

The argument list will usually be of the form

```
arg1, arg2, arg3 = default3, arg4 = default4 , …
```

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 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(2,1,4)

[1] 6

> testfun2(2,z=4)

[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

```
succ <- function(n){
  n <- n+1
  n
}
> n
[1] 2
> succ(n)
[1] 3
> n
[1] 2
```

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

```
testfun3 <- function(x, y = 1, z)
 { x * y + z * n}
> testfun3(2,z=4)
[1] 10
```

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

In addition it possible for assignments not to be local, but global. You can reassign values in the first level of the search path from within a function. For example

```
succ2 <- function(n){
  x <- n+1
  n <<- x
  x
}
> n
[1] 4
> succ2(n)
[1] 5
> n
[1] 5
```

This is 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, if, while, etc)

The standard control structures in most high level languages are available in S-Plus/R. The 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(x) {
  if(x != trunc(x)) {stop("x is not an integer")}
  else {
    fact<-1
    for(i in 1:x)
      fact<- fact * i
  }
  fact
}
```

In the if statement, the condition should be a single logical value. If you are dealing with vectors, you may not get what you want. 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, such as with

```
result <- if (test == “Levene”) levene(y, f)

  else

    if (test == “Cochran”) cochran(y, f)

      else bartlett(y, f)
```

it may be easier to deal with as

```
result <- switch(test,

    Levene = levene(y, f)

    Cochran = cochran(y, f)

    Bartlett = Bartlett(y, f))
```

`for`:

The basic structure is

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

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

Note that the sequence doesn't have to be a vector. It could be a list or a data frame, as with

```
> for (i in lcrabs)
+  print(summary(i))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.974   2.557   2.744   2.720   2.893   3.140
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.872   2.398   2.549   2.523   2.660   3.006
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.688   3.306   3.469   3.443   3.617   3.863
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.839   3.450   3.605   3.570   3.738   4.000
```

`while`:

while (*`condition`*) *`commands`*

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

`repeat`:

`repeat` *`commands`*

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

The commands in the repeat loop will continue until a break is seen which will hop you out of the loop.

Error checking

When writing functions, it is usually a good idea to check to make sure the input arguments are value. For example, with the factorial functions shown before as all based on a single, integer being input.

```
facte <- function(x) {

  if (length(x)>1) warning("x should be of
length 1, only first component used")

  if (x[1] <= 0) stop("x must be positive")

  prod(1:x[1])

}

> facte(1:4)

[1] 1

Warning message:

x should be of length 1, only first component
used in: facte(1:4)

> facte(-2)

Error in facte(-2) : x must be positive
```

`warning` will allow the function to continue and return output. `stop`, however will terminate the function, yielding no output.

Printing output in functions

Sometimes it is useful to print results of calculations from within a function, such as during debugging.

For example

```
testprint1 <- function(x) {
  for (i in 1:x)
    c(i, fact(i))
}
> testprint1(4)
>
```

You must explicitly print the objects as with

```
testprint2 <- function(x) {
  for (i in 1:x)
    print(c(i,fact(i)))
}
> testprint2(4)
[1] 1 1
[1] 2 2
[1] 3 6
[1]  4 24
```

The functions `cat` and `format` is also useful, particularly with formatted output.

```
testprint3 <- function(x) {
  for (i in 1:x)
    cat("x = ",i,", ",i,"! = ",fact(i),"\n",
        sep="")
}
> testprint3(4)
x = 1, 1! = 1
x = 2, 2! = 2
x = 3, 3! = 6
x = 4, 4! = 24
```

Recursive functions

Another approach that can be useful is that of recursive functions. For example, the factorial function can be written as

$$x! = x \times (x - 1)!$$

This can be written as

```
factr <- function(x) {
  if(x != trunc(x))
    stop("x is not an integer")
  if (x == 1) factr <- 1
  else factr <- factr(x-1) * x
  factr
}
```

Note that this approach is usually slow and memory-intensive. Each time the function is called, a copy of the important information is made and passed onto the new call. The function factr will only handle x up to 82. With $x \geq 83$,

```
> factr(82)
[1] 4.753643e+122
> factr(83)
Error in factr(x - 1) : evaluation is nested too
deeply: infinite recursion?
```

However this approach does have its uses with intrinsically recursive problems. For example, listing all possible subsets of size *r* from *n* objects.

Example code

```
subsets <- function(n, r, v = 1:n) {
  if(r <= 0) NULL else
  if(r >= n) v[1:n] else
  rbind(cbind(v[1], subsets(n - 1, r - 1, v[-1])),
                    subsets(n - 1, r    , v[-1]))
}
```

The idea behind this function is that if $n = r$, there is only one possible subset, the whole vector. Otherwise pick one element from the set. Then you need to look at all subsets with that element combined with subsets of size $r - 1$ from the remaining $n - 1$ elements plus the subsets of size *r* taken from the other $n - 1$ elements.

Note that this is not the best way to write a recursive function as it you change the name, the function will break (subsets won't exist any more). See pages 49-50 in S Programming by Venables and Ripley. A better approach uses the Recall function.

Vectorized functions

The standard S functions, such as sin, log, dnorm, etc, have the useful property that if the first argument is a vector, the result is a vector of the same size. Note that a similar result will also happen with matrices and higher level arrays.

For example in S-Plus try `sin(iris)` or in R try `data(iris3); sin(iris3)`. You'll see that the result is a 3 dimensional array, the same as `iris` (or `iris3`).

When designing your own functions, you should strive to do the same thing. Often it is easy to do, as where possible you should base your own functions on the built in vectorized functions.

For example, probably the best version of the factorial function you could write is

```
fact <- function(x) gamma(x+1)
```

Since it uses the built in function gamma, all of its built-in error checking will be there. Also it is automatically vectorized. However an example of how you could write a vectorized version of the factorial function is

```
fact.vec <- function(x) {
  size <- dim(x)
  fact <- NULL
  for(i in x) {
    temp <- 1
    for(j in 1:i)
      temp <- temp * j
    fact <- c(fact,temp)
  }
  array(fact, dim=size)
}
> fact.vec(mat)
     [,1]  [,2]      [,3]
[1,]    1   120    362880
[2,]    2   720   3628800
[3,]    6  5040  39916800
[4,]   24 40320 479001600
> fact(mat)
     [,1]  [,2]      [,3]
[1,]    1   120    362880
[2,]    2   720   3628800
[3,]    6  5040  39916800
[4,]   24 40320 479001600
```

Loops vs vectorized calculations

Where possible, you generally want to avoid using loops, particularly in S-Plus. (The situation isn't quite as bad in R). The reason is similar to why you don't want to write recursive functions.

Lets look at the Fisher Iris data set, getting summary statistics for the different species and measurement.

```
> meanmat<-matrix(0,ncol=3,nrow=4,dimnames
= list(c("Sepal L", "Sepal W", "Petal L",
"Petal W"), c("Setosa", "Versicolor",
"Virginica")))

> for (i in 1:4)

+ for (j in 1:3)

+   meanmat[i,j] <-mean(iris3[,i,j])

> meanmat

        Setosa Versicolor Virginica

Sepal L  5.006      5.936     6.588

Sepal W  3.428      2.770     2.974

Petal L  1.462      4.260     5.552

Petal W  0.246      1.326     2.026
```

This can be done much easier with `apply`.

```
> apply(iris3,c(2,3),mean)
         Setosa Versicolor Virginica
Sepal L.  5.006      5.936     6.588
Sepal W.  3.428      2.770     2.974
Petal L.  1.462      4.260     5.552
Petal W.  0.246      1.326     2.026
```

The general form of apply is

```
apply(array, margins, function, …)
```

margins are the dimensions that you want to apply the function over. For the iris dataset, the first dimension corresponds to observation number, the second corresponds to variable, and the third is the species. The example says to fix each combination of variable and species and average the observations for each combination.

If you just want to average for each variable (averaging over observations and species) use the following

```
> apply(iris3,2,mean)
Sepal L. Sepal W. Petal L. Petal W.
5.843333 3.057333 3.758000 1.199333
```

Note additional arguments can be passed onto the function. For example

```
> apply(iris3,2,mean,trim=0.1)

Sepal L. Sepal W. Petal L. Petal W.

5.808333 3.043333 3.760000 1.184167
```

which will calculate the 10% trimmed mean for each variable.

Note that for linear computations, such as computing the mean, using matrix multiplication can be even more efficient. For example, instead of `apply(iris3,c(2,3),mean)`, the following could also have been used

```
> matrix(rep(1/50,50) %*% matrix(iris3,nrow=50),
+   nrow=4, dimnames = dimnames(iris3)[-1])

          Setosa Versicolor Virginica

Sepal L.  5.006      5.936     6.588

Sepal W.  3.428      2.770     2.974

Petal L.  1.462      4.260     5.552

Petal W.  0.246      1.326     2.026
```

While more efficient, I'll often use apply, since its more readable. Also it may take longer to figure out how to do it more efficiently that the improvement in calculation time.

In addition to `apply` there are 3 similar function for different data structures, `tapply`, `lapply`, `sapply`.

`tapply` when you have a vector and a labeling factor or factors. For example the get the variances of log(CL) in the crabs data, the following can be used

```
> tapply(lcrabs[,3],sex,var)

    Female       Male

0.05199732 0.05906848

> tapply(lcrabs[,3],list(sex,sp),var)

             Blue    Orange

Female 0.04914712 0.0319532

Male   0.06104549 0.0568927
```

The general form of the function is

```
tapply(vector, label, function, …)
```

It is possible to use your own functions with apply, tapply, etc, with the function even defined in the call.

```
> tapply(lcrabs[,3],list(sex,sp),
  function(x) sqrt(var(x)/length(x)))

             Blue     Orange

Female 0.03135191 0.02527972

Male   0.03494152 0.03373209
```

The last two, lapply and sapply are used with lists (& dataframes)

For example

```
> lapply(cars93,function(x) class(x))

$Manu

[1] "factor"

$Model

[1] "factor"

$Type

[1] "factor"

$MinPrice

[1] "numeric"

$MidPrice

[1] "numeric"

and so on
```

The function lapply will always return a list, whereas, sapply, which does the same calculations, will try to return a vector if possible. For example

```
> sapply(cars93,mean)
        Manu        Model         Type     MinPrice     MidPrice
          NA           NA           NA   17.1258065   19.5096774
     CityMPG      HighMPG      AirBags     DriveTra     Cylinder
  22.3655914   29.0860215    0.8064516    0.9354839           NA
       Horse          RPM     EngRevMi       Manual     FuelTank
 143.8279570 5280.6451613 2332.2043011    0.6559140   16.6645161
> lapply(cars93,mean)
$Manu
[1] NA
$Model
[1] NA
$Type
[1] NA
$MinPrice
[1] 17.12581
$MidPrice
[1] 19.50968
$MaxPrice
[1] 21.89892
```

Note while it is desirable to deal with vectorized calculations, it is not always possible and loops must be used. An example of this is the Gibbs sampler, which is an iterative approach for generating samples from complicated joint distributions. Suppose you wanted to generate samples from the joint density f(x,y,z), but f is complicated. A scheme that will generate (dependent) samples (asymptotically) is

```
initialize x, y, and z as $x^{(0)}$, $y^{(0)}$, and $z^{(0)}$
for i = 1 to n {
  draw $x^{(i)}$ from $f(x|y^{(i-1)}, z^{(i-1)})$
  draw $y^{(i)}$ from $f(y|x^{(i)}, z^{(i-1)})$
  draw $z^{(i)}$ from $f(x|x^{(i)}, y^{(i)})$
}
```

The realizations $(x^{(i)}, y^{(i)}, z^{(i)})$ form a Markov Chain with a stationary distribution with density f(x,y,z).

This approach, and its extensions which are known generally as Markov Chain Monte Carlo (MCMC), has opened many areas of statistics in the last 20 years, in particular Bayesian analysis. The techniques actually go back to the Manhattan project during WW2.
N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller J. Chem. Phys 21 (1953) 1087.