R: Programming and Looping Functions

140.776 Statistical Computing

September 29, 2011
Functions can be recursive. For example, suppose $x = 2^d$. Now given an integer $x$, we want to compute $d = \log_2(x)$. You can implement using a loop:

```r
g<-function(x) {
  d<-0
  while(x>=2) {
    x<-x/2
    d<-d+1
  }
}

> y<-g(32)
> y
[1] 5
```
But you can also implement it using recursive function calls:

```r
f <- function(x) {
    if (x > 2) {
        d <- 1 + f(x / 2)
    } else {
        d <- 1
    }
    d
}
```

> y <- f(32)
> y
[1] 5

The function `f()` calls itself within the body of the function.
> load("tree.rda")
Tree

preferred

biostatistician

most

that

agree

the

to

approach

analyzing

140.776 Statistical Computing  R: Programming and Looping Functions
scantree<-function(t) {
    leftn<-t$left
    rightn<-t$right

    if(is.list(leftn) == TRUE) {
        strl<-scantree(leftn)
    } else {
        strl<=""
    }

    if(is.list(rightn) == TRUE) {
        strr<-scantree(rightn)
    } else {
        strr<=""
    }

    str<-c(strl, t$key, strr)
}

myst<-scantree(mytree)
print(paste(myst, collapse= " "))
The for, while loops can often be replaced by looping functions:

- **lapply**: loop over a list and evaluate a function on each element
- **sapply**: same as lapply but try to simplify the result
- **apply**: apply a function over the margins of an array
- **tapply**: apply a function over subsets of a vector
- **mapply**: multivariate version of lapply
lapply(X, FUN, ...)

lapply takes three arguments: a list X, a function FUN, and other arguments ... If X is not a list, it will be converted to a list using as.list().

It returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.
Examples:

```r
> u<-list(x=1:10,y=rnorm(100))
> lapply(u,mean)
$y
[1] -0.06615078

> v<-c("bio","stat","comput")
> lapply(v,nchar)
[[1]]
[1] 3
[[2]]
[1] 4
[[3]]
[1] 6
```
> lapply(1:4, runif)

[[1]]
[1] 0.7082681

[[2]]
[1] 0.1966707 0.3025155

[[3]]
[1] 0.1872824 0.1093319 0.1641600

[[4]]
[1] 0.29028286 0.74228311 0.03167801 0.44816463
lapply makes heavy use of *anonymous functions*.

```r
> x<-list(a=matrix(1:4,2,2),b=matrix(1:6,3,2))
> x
$a
 [,1] [,2]
[1,] 1 3
[2,] 2 4

$b
 [,1] [,2]
[1,] 1 4
[2,] 2 5
[3,] 3 6
```
An anonymous function for extracting the first column of each matrix.

> lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
sapply is a user-friendly version of lapply.

- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length (>1), a matrix is returned.
- If it cannot figure things out, a list is returned.
Examples:

```r
> v<-c("bio","stat","comput")
> lapply(v,nchar)
[[1]]
[1] 3

[[2]]
[1] 4

[[3]]
[1] 6

> sapply(v,nchar)
       bio   stat  comput
[1,]    3     4     6
```
apply(X, MARGIN, FUN, ...)  

Applying a function to margins of an array X.

- If X is not an array but has a dimension attribute, apply attempts to coerce it to an array via `as.matrix` if it is two-dimensional (e.g., data frames) or via `as.array`.
- MARGIN is an integer vector indicating which margins should be “retained”.
- FUN is a function to be applied.
- ... is for arguments to be passed to FUN.
If each call to FUN returns a vector of length n, then apply returns an array of dimension c(n, dim(X)[MARGIN]) if n > 1. If n equals 1, apply returns a vector if MARGIN has length 1 and an array of dimension dim(X)[MARGIN] otherwise.

If the calls to FUN return vectors of different lengths, apply returns a list of length prod(dim(X)[MARGIN]) with dim set to MARGIN if this has length greater than one.
> x <- matrix(1:6, 3, 2)
> x

[,1] [,2]
[1,] 1  4
[2,] 2  5
[3,] 3  6

> apply(x, 1, function(u) sum(u))
[1]  5  7  9

> apply(x, 1, mean)
[1] 2.5 3.5 4.5
Quantiles of the rows of a matrix.

```r
> x <- matrix(rnorm(100), 5, 20)

> apply(x, 1, quantile, probs = c(0.25, 0.75))

25% -0.7463242 -0.1705673 -0.6468663 -0.7671974
75% 0.4318116  0.9564692  0.4955044  0.4381032

[,5]
25% -0.9612553
75% 1.4259372
```
For sums and means of matrix dimensions, we have some shortcuts.

- `rowSums = apply(x,1,sum)`
- `rowMeans = apply(x,1,mean)`
- `colSums = apply(x,2,sum)`
- `colMeans = apply(x,2,mean)`

The shortcut functions are much faster, but you won’t notice unless you’re using a large matrix.
col/row sums and means

```r
> x <- array(rnorm(2*2*10), c(2,2,10))
> apply(x, c(1,2), sum)
    [,1]      [,2]
[1,]  4.277454  2.407804
[2,] -0.5435999 3.988917

> rowSums(x, dims=2)
    [,1]      [,2]
[1,]  4.277454  2.407804
[2,] -0.5435999 3.988917

> rowSums(x)
[1]  6.685258  3.445317
```
We’ve already seen `tapply` before:

```r
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

Applying a function to each (non-empty) group of values given by a unique combination of the levels of certain factors.

- **X**: an atomic object, typically a vector
- **INDEX**: list of factors, each of same length as `X`. The elements are coerced to factors by `as.factor()`
- **FUN**: the function to be applied
- **...**: optional arguments to `FUN`
- **simplify**: if `FALSE`, returns a list. If `TRUE`, then if `FUN` always returns a scalar, `tapply` returns an array with the mode of the scalar.
> x<-c(rnorm(5),rnorm(5,1),rnorm(5,2),rnorm(5,3))
> f1<-factor(rep(1:2,each=10))
> f1
  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
Levels: 1 2
> f2<-factor(rep(rep(3:4,each=5),times=2))
> f2
  [1] 3 3 3 3 3 4 4 4 4 4 3 3 3 3 3 4 4 4 4 4
Levels: 3 4
> f<-list(f1,f2)
> tapply(x,f,mean)
     3     4
1 0.4687781 0.9727993
2 1.7442365 3.4148615
> x<-c(rnorm(10),rnorm(10,1),rnorm(10,2))
> f<-gl(3,10)
> f
[1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
[27] 3 3 3 3
Levels: 1 2 3

> tapply(x,f,mean)
          1          2          3
0.3104300 0.9665058 2.0069397

> tapply(x,f,mean,simplify=FALSE)
$'1'
[1] 0.3104300
$'2'
[1] 0.9665058
$'3'
[1] 2.006940
Find group ranges.

> tapply(x,f,range)

$’1’$
[1]  -1.087020  1.801051

$’2’$
[1]  -0.3175924  2.4043049

$’3’$
[1]   0.9338877  3.4817331
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)

mapply is a multivariate version of sapply. mapply applies FUN to the first elements of each ... argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.

- **FUN**: the function to be applied
- **...**: arguments to apply over
- **MoreArgs**: a list of other arguments to FUN
- **SIMPLIFY**: logical; whether the result should be simplified to a vector or matrix.
Example:

```r
## tedious to type
> list(rep(1,4),rep(2,3),rep(3,2),rep(4,1))

## use mapply instead
> mapply(rep,1:4, 4:1)
[[1]]
[1] 1 1 1 1
[[2]]
[1] 2 2 2
[[3]]
[1] 3 3
[[4]]
[1] 4
```
noise <- function(n, mean, sd) {
  rnorm(n, mean, sd)
}

> noise(3, 1, 2)
[1] 5.164179 1.353838 -2.573485

> noise(1:3, 1:3, 2)
[1] -3.8190429 1.6455998 0.4092931
> mapply(noise,1:3,1:3,2)
[[1]]
[1] 3.619997

[[2]]
[1] 5.641030 -2.344175

[[3]]
[1] 0.7557551 -0.4642377 4.8742734

which is the same as

list(noise(1,1,2),noise(2,2,2),noise(3,3,2))
split(x, f, drop = FALSE, ...)  

split divides the data in the x into the groups defined by a factor or list of factors. It is often followed by lapply.

- x: a vector or data frame
- f: a factor or a list of factors
- drop: logical; whether empty levels should be dropped
```r
> x <- c(rnorm(4), runif(4), rgamma(4, 1, 1))
> f <- gl(3, 4)
> split(x, f)
$'1'
 [1]  -2.03109144 -0.08146077 -0.17701322 -0.78487670

$'2'
 [1]  0.1401723  0.6657537  0.6366146  0.7639057

$'3'
 [1]  0.4280483  0.6125946  1.6261508  1.6642573
```
> lapply(split(x,f),mean)

$'1'$
[1] -0.7686105

$'2'$
[1] 0.5516116

$'3'$
[1] 1.082763
> x<-rnorm(10)
> f1<-gl(2,5)
> f2<-gl(5,2)
> f1
  [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
  [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> interaction(f1,f2)
  [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
Interactions can create empty levels.

> str(split(x,list(f1,f2)))
List of 10
$ 1.1: num [1:2] 0.23 0.862
$ 2.1: num(0)
$ 1.2: num [1:2] -0.849 0.511
$ 2.2: num(0)
$ 1.3: num -1.33
$ 2.3: num -3.12
$ 1.4: num(0)
$ 2.4: num [1:2] 1.3 -2.11
$ 1.5: num(0)
$ 2.5: num [1:2] -0.726 0.839
Splitting on more than one level

Empty levels can be dropped.

```r
> str(split(x, list(f1, f2), drop=TRUE))

List of 6
 $ 1.1: num [1:2] 0.23 0.862
 $ 1.2: num [1:2] -0.849 0.511
 $ 1.3: num -1.33
 $ 2.3: num -3.12
 $ 2.4: num [1:2] 1.3 -2.11
 $ 2.5: num [1:2] -0.726 0.839
```