R: Introduction to Reference Classes

Aleix Ruiz de Villa

RUGBCN

December 15th, 2011

An environment is:

- A collection of R objects
- A link (reference) to another environment (enclosing environment, parent)

CODE

```
> newEnv = new.env()
> newEnv$newEle = 3
> newEnv$newFunc = function(x){
+   print(2*x)
+ }
> newEnv$newFunc(newEnv$newEle)

[1] 6
```

Environments are ordered. .GlobalEnv contains functions and objects that are assigned during the session.

CODE

```
> search()

[1] ".GlobalEnv"         "package:stats"
[3] "package:graphics"   "package:grDevices"
[5] "package:utils"      "package:datasets"
[7] "package:methods"    "Autoloads"
[9] "package:base"

> parentEnv = parent.env(.GlobalEnv)
> environmentName(parentEnv)

[1] "package:stats"
```

- A call to a function creates a new environment
- The environment disapears when the call is complete (and local objects are removed)

CODE

```
> auxFunc = function(){
+    a = 2
+    print(a)
+ }
> auxFunc()

[1] 2

> print(a)

[1] "Error in print(a) : object 'a' not found\n"
```

- Using $=$ or $<-$ we make local assignments (in the environment)

CODE

```
> a = 1
> auxFunc = function(a){
+    a = a + 1
+    print(a)
+ }
> auxFunc(a)

[1] 2

> print(a)

[1] 1
```

- The parent environment is the one that created the function (not the calling environment).
- We can acces to objects outside the environment using $<<-$. R searches for the name through all the enclosing environments. If there is an existing object with this name, then the assignment takes place there. Otherwise, the object is assigned in the global environment.

CODE

```
> a = 1
> auxFunc = function(a){
+   a <<- a + 1
+   print(a)
+ }
> auxFunc(a)

[1] 1

> print(a)

[1] 2
```

- A call to a function copies all its arguments except environments. Reference classes are implemented using environments.
- Reference classes provide the oportunity of passing objects wihtout copying them, so that they can be modified inside a function.
- S3 and S4 where the previous models for classes in R. Reference classes have an OOP (Object Oriented Programing) fashion, so the code is better organized.

A class is structure containing:

- R objects (Fields)
- Functions (Methods)

Instances of a class are called objects. Notice the difference of names between R objects and objects.
Every object has to be created and removed by the initialize and finalize methods.

Definition

```
> MatrixClass = setRefClass(
+   Class = "MatrixClass",
+   fields = list(
+     dataMat = "matrix",
+     detMat ="numeric",
+     inverseMat = "matrix"))
```

Constructor

```
> MatrixClass$methods(
+   initialize = function(extMat = diag(1), ...){
+
+   dataMat <<- extMat
+
+   detMat <<- det(dataMat)
+   if(abs(detMat) > 1e-7 ){
+     inverseMat <<- solve(dataMat)
+   }else{
+     inverseMat <<- NULL
+   }
+
+   callSuper(...)
+ })
```

Remark: *The expression "extMat = diag(1)" means that when no argument is passed, extMat will value diag(1). This is necessary if we use our classes for "inheriting" (we will talk about it later).*

CODE

```
> extMat = diag(3) * c(1,2,3)
> newMat = MatrixClass$new(extMat)
> newMat

Reference class object of class "MatrixClass"
Field "dataMat":
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
Field "detMat":
[1] 6
Field "inverseMat":
     [,1] [,2]      [,3]
[1,]    1  0.0 0.0000000
[2,]    0  0.5 0.0000000
[3,]    0  0.0 0.3333333

> newMat$dataMat

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

Destructor

```
> MatrixClass$methods(finalize = function(){
+
+   print(objects(.self))
+   for(objName in objects(.self)){
+     obj = get(objName, env = .self)
+     if(!is.function(obj)){
+       print(objName)
+       print(obj)
+     }
+   }
+
+ })
```

Remark: *The object ".self" is the class object itself.*

CODE

```
> newMat$finalize()

[1] "dataMat"     "detMat"      "field"       "finalize"
[5] "initFields" "initialize" "inverseMat" "show"
[1] "dataMat"
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
[1] "detMat"
[1] 6
[1] "inverseMat"
     [,1] [,2]      [,3]
[1,]    1  0.0 0.0000000
[2,]    0  0.5 0.0000000
[3,]    0  0.0 0.3333333
```

Methods

```
> MatrixClass$methods(linearEq = function(b){
+    # Ax = b
+    x = inverseMat %*% b
+
+ })
```

CODE

```
> altMat = newMat
> altMat$dataMat = diag(2) * c(1,2)
> altMat$dataMat

     [,1] [,2]
[1,]    1    0
[2,]    0    2

> newMat$dataMat

     [,1] [,2]
[1,]    1    0
[2,]    0    2
```

Copy

```
> altMat = newMat$copy()
> altMat$dataMat = diag(2) * c(1,2)
> altMat$dataMat

     [,1] [,2]
[1,]    1    0
[2,]    0    2

> newMat$dataMat

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

We can build new classes using old classes. This is called inheritance.

Inheritance

```
> SymMatrixClass = setRefClass(
+   Class = "SymMatrixClass",
+   fields = list(
+     eigenValues = "vector"),
+   contains = "MatrixClass")

> SymMatrixClass$methods(
+   initialize = function(extMat,...){
+     if(!all(extMat == t(extMat))){
+       print("Non symmetric matrix")
+       stop()
+     }
+     eigenValues <<- eigen(extMat)$values
+     callSuper(extMat,...)
+   })
```

CODE

```
> newSymMat = SymMatrixClass$new(diag(3) * c(1,2,3))
> newSymMat

Reference class object of class "SymMatrixClass"
Field "dataMat":
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
Field "detMat":
[1] 6
Field "inverseMat":
     [,1] [,2]      [,3]
[1,]    1  0.0 0.0000000
[2,]    0  0.5 0.0000000
[3,]    0  0.0 0.3333333
Field "eigenValues":
[1] 3 2 1
```