Environnments and R's Reference Classes

Overview

A saying people sometimes use to summarize the philosophy of R is, "Every thing is an *object*, and every action is a *function*." I like to add, "And functions are objects." :-)

So, for instance, the **ls()** function is an object, and addition is a function (and thus an object):

```
> 3 + 8
[1] 11
> `+`(3,8)  # + is actually a function!
[1] 11
> a <- list(u = 1, v = 12)
> a$v
[1] 12
> `$`(a,v)  # $ is actually a function!
[1] 12
```

However, R's class nature has traditionally been rather different from what C/C++/Java/Python programmers are used to.

S3 Classes

The classic R class structure is S3, taken from the old S language. It packages various data items into an object, thus featuring the *encapsulation* tenet of the OOP philosophy. But class functions are not packaged inside the class. Instead, the notion of *generic functions* and *function call dispatch* are used to achieve *polymorphism*, which means that the same function will work in different ways on different types of objects. Polymorphism is one of the core tents of the object-oriented programming philosophy.

Say for example we create a new class **cls**. We probably would want users to be able to nicely print out instances of this class, which we do by writing a function **print.cls()**. Then if **x** is an instance of the class, typing

```
> print(x)
```

will result in the R interpreter dispatching this call to the actual call **print.cls(x)**.

We say that **print()** is a generic function. Other common examples are **plot()** and **summary()**.

An S3 object is simply an R list, with a **class** attribute tacked on.

```
> l <- list(u=3, v=8)
> class(l) <- 'cls'
```

```
> l
$u
[1] 3

$v
[1] 8

attr(,"class")
[1] "cls"
> l$v
[1] 8
> l$v <- 88
> l$v
[1] 88
```

Since S3 classes are lists, the '$' symbol is used to delineate members.

S4 Classes

Some people felt that S3 classes have two flaws: (a) They don't locate functions physically within the class (violating the encapsulation tenet), and (b) they are not "safe," in that more members of the class can be added later, possibly by accidents such as misspelling. S4 classes addresses those concerns, giving "C++-style" clases for those who want them. (Many, actually most, of us stuck to S3.)

S4 objects are created as in C++ and Java, using **new()**. Class member delineation is via the '@' symbol.

Environnments

An R *environment* is a collection of objects, working something like a list.

We speak of the *top-level* environment. When you are working at the interactive prompt '>', that environment, named **.GlobalEnv**, consists of the objects you created at that level. (If you have an **.Rprofile** startup file, it's run when you start R, so whatever is created there is also part of this top-level environment.)

Use in function calls

If you create a function, say **f()**, at the top-level, its environment will be the top-level one. *Within* the call, an environment is temporarily created for that call, consisting of the arguments, local variables, and a pointer to the parent environment.

This then occurs recursively, for functions defined within functions, and so on. E.g.,

```
> f <- function(x) {z <- (x+y)^2; g <- function(w) w*z*y; g(5)}
> y <- 6
> f(3)
[1] 2430
```

What happened? Here **y** was in the environment of **f()**. When **x+y** was evaluated, the current environment consisted of **x** and a pointer to the environment of **f()**. which included **y**. Thus **y** was found and **z** was computed, and **z**, a new local, was added to the current environment, which now consisted of **x**, **z** and the pointer to the top-level environment.

Next, **g()** was created, a new local, and added to the current environment, now consisting of **x**, **z**, **g** and a pointer to the top-level. Then during the call to **g()**, the current environment consisted of **w**, plus a pointer to the environment of **f()**'s call; so, all of **w**, **y** and **z** were available.

Accessing Other Environnments

In the above example, we call the top level the *enclosing* or *parent* environment to the environment in effect during the call to **f()**. The upward pointer can be followed using **parent.frame()**:

```
> h <- function(x) {print(ls()); print(ls(envir=parent.frame()))}
> h(5)
[1] "x"
 [1] "a"       "bld"     "d"       "dbb"
 [5] "dbcurr"  "dbl"     "dbsrci"  "dbtb"
 [9] "e"       "evalr"   "f"       "fedit"
[13] "g"       "lsp"     "odf"     "pr2file"
[17] "relib"   "srcname" "y"
```

The first **ls()** call just printed the current environment, i.e. just **x**. But the second went to the parent environment. I had a lot of stuff in my top-level environment, but you can see **y** there.

We can assign things up the chain using the '<<-' operator, but the **assign()** function is more specific.

```
> h <- function(x) assign('q',(x+y)^2,envir=parent.frame())
> h(18)
> q
[1] 576
```

The **get()** function fetches something for us from another environment.

Making our own environments

It's useful to form environments apart from function call uses. We can create an environment via the **new.env()** function, as you'll see below.

As a functional language, in R there are (almost) no *side effects*. The latter means that in a call **k(v)**, the actual argument corresponding to a formal argument **v** will not change, even if **v** changes within the function. (The **v** within the function is just a local copy.) If one does want the original **v** to change, one must reassign to it from the return value of the function.

An exception to that is R environments.

```
> d <- list(x = 3, w = 1:5)
> f <- function(obj) obj$x <- 12
> f(d)
> d$x  # still 3, no change
[1] 3
> e <- new.env()
> e$x <- 3
> e$w <- 1:5
> f(e)
> e$x  # not 3 anymore, changed to 12
[1] 12
```

Environnments as a Place for Globals

Many R specialists disdain global variables, considering them "unsafe." I disagree – careful use of globals can make code simpler and clearer, thus "safer" – but those who dislike globals would say "OK, if you must use globals, don't use '<<-'. Instead, put all your globals in an environment, say **globals**.

Their objection to '<<-' is that one may accidentally have a variable of the same name at an intermediate level, even though we want the top level. Then we would be assigning to ther intermediate one, not the top one as desired. They also believe that packaging the globals into one container is clearer. Again, the latter is the principle of encapsulation.

Reference Classes

At any rate, environments give us the ability to have list-like structures while not being restricted by the no-side-effects functional programming ethos.

Reference classes are then essentially S4 classes but stored as environments rather than lists. A function can then change the element of such a class, without

having to reassign, and most important, without having to create a new copy of the whole class instance.

Say we have a list **z** with element **u** and **v**, vector of length 3 and 100 million, respectively. If our function changes only **u**, a new copy of **z** would be created, meaning memory must be allocated and copied to, very time consuming. With an environment, that would not occur.

(Note though that it is still the case that with any vector **x**, doing something like **x[5] <- 12** risks reallocation of **x**.)

A good starting example of using reference classes is this implementation of a queue data structure.