Programming in R

Duncan Murdoch

Department of Statistical and Actuarial Sciences
University of Western Ontario

December 5, 2008

Outline

The World According to R

- The user's workspace is called the "global environment".
- R's objects are in a search list of environments.
- There's also a file system on the host computer, and possibly some graphics devices and other external things like databases, etc.

Simple Computations

```
> ls()
character(0)
> x <- 1
> ls()
[1] "x"
> x
[1] 1
> search()
 [1] ".GlobalEnv"        "package:patchDVI"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

(Almost) Everything is a Vector

Most R objects are vectors, and most operations work element-by-element.

```
> x <- 1:5
> x
[1] 1 2 3 4 5
> y <- 6:10
> y
[1]  6  7  8  9 10
> x + y
[1]  7  9 11 13 15
```

Indexing Vectors

Vectors may be indexed by numbers (starting from 1) or logicals (keep the `TRUE` entries). Negative indices mean "leave this out".

```
> x[4:5] <- NA
> x
[1]  1  2  3 NA NA
> x[!is.na(x)]
[1] 1 2 3
> x[-(1:2)]
[1]  3 NA NA
```

Matrices

Matrices and arrays are vectors with dimension. Data frames act like matrices in many ways. Watch for dropped dimensions!

```
> print(m <- matrix(1:6, nrow=2))
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(m)
[1] 2 3
> m[, 3]
[1] 5 6
> m[1, 2]
[1] 3
> m[1, 2, drop=FALSE]
     [,1]
[1,]    3
```

Types of Objects

Every object in R has a type. The vector types are:

- The atomic vectors: `logical`, `integer`, `double`, `complex`, `character` and `raw`.
- List vectors: `list`

The type of an object is set when it is created, and can change as the program runs.

Atomic types

logical	`TRUE` or `FALSE`
integer	whole numbers from `-2147483647L` to `2147483647L`
double	floating point numbers: 0 and `2.225074e-308` to `1.797693e+308` (of both signs), plus special values: `-Inf`, `Inf`, `NaN`
complex	pairs of `double` values treated as complex numbers
character	character strings
raw	raw bytes

Each (except `raw`) has an `NA`, the missing value.

Examples

```
> typeof(TRUE)
[1] "logical"
> x <- c("hello", "goodbye")
> typeof(x)
[1] "character"
> x <- 1 + 2i
> typeof(x)
[1] "complex"
```

Examples

How a value prints doesn't always indicate its type:

```
> as.integer(1)
[1] 1
> typeof(as.integer(1))
[1] "integer"
> 1
[1] 1
> typeof(1)
[1] "double"
```

Recursive Types versus Atomic Types

- Lists are a *recursive* type. Each element of a list can be of a different type.

```
> y <- list(3, "abc", c(1.3, 4.5))
> typeof(y)
[1] "list"
> typeof(y[[2]])
[1] "character"
> typeof(y[1:2])
[1] "list"
```

We use `[[ ]]` to extract an element of a list, and `[ ]` to create a subset of a vector.

Atomic vectors must be all of one type. `[[ ]]` and `[ ]` are the same.

```
> z <- c("abc", "def", "ghi")
> typeof(z)
[1] "character"
> typeof(z[2])
[1] "character"
> typeof(z[[2]])
[1] "character"
```

Other types

Other types include:

- `closure` (interpreted function)
- `environment`
- `NULL`
- Other types: `symbol`, `pairlist`, `promise`, `language`, `special`, `builtin`, `char`, `expression`, `externalptr`, `weakref`, `S4`, plus a few others.

Examples

In R, the `typeof()` function is the main way for a program to determine the type. There are also `mode()` and `storage.mode()`, mainly for S compatibility.
A user friendly function is `str()` (for "structure"):

```
> str(x)
 cplx 1+2i
> str(1)
 num 1
> str(as.integer(1))
 int 1
```

Examples

```
> y[[4]] <- list(1, 1 + 2i)
> str(y)
List of 4
 $ : num 3
 $ : chr "abc"
 $ : num [1:2] 1.3 4.5
 $ :List of 2
  ..$ : num 1
  ..$ : cplx 1+2i
```

Vector elements can be named

```
> names(z) <- c("Tom", "Dick", "Harry")
> z
  Tom  Dick Harry
"abc" "def" "ghi"
> z["Harry"]
Harry
"ghi"
> names(y) <- LETTERS[1:4]
> y$C   # $ works for lists, not atomic vectors
[1] 1.3 4.5
```

Environments

Environments are recursive types similar to lists, but:

- They are unordered; objects in them can only be retrieved by the exact name.
- Assigning an environment to a variable only creates a new reference to the same environment, not a new copy.
- They have a *parent* environment; (some) searches for objects will continue the search in the parent if the object is not found in the current environment.

The global environment is an object of type *environment*.

Environments cont'd

```
> x <- 123
> e <- new.env()
> e$y <- 456
> e$x
NULL
> e$y
[1] 456
> get("x", e)
[1] 123
> get("y", e)
[1] 456
```

Evaluation

Expressions are always evaluated in an environment:

- The global environment when typed at the console.
- When a function call is evaluated, a new local environment is created, called the *evaluation frame* (more on this later).
- Wherever you want:

```
> y <- 321
> evalq(x + y)
[1] 444
> evalq(x + y, e)
[1] 579
```

The `with()` function

The `with()` function allows temporary work within a data frame (or an environment, or list...).

```
> iris$Sepal.Length[1:10]
 [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
> iris$Sepal.Width[1:10]
 [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1
> y <- with(iris, Sepal.Length + Sepal.Width)
> y[1:10]
 [1] 8.6 7.9 7.9 7.7 8.6 9.3 8.0 8.4 7.3 8.0
```

The `within()` function

The `within()` function can make changes within a data frame.

```
> newiris <- within(iris,
+   { Sepal.Area <- Sepal.Length * Sepal.Width
+     Petal.Area <- Petal.Length * Petal.Width })
> head(newiris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
  Petal.Area Sepal.Area
1       0.28      17.85
2       0.28      14.70
3       0.26      15.04
4       0.30      14.26
5       0.28      18.00
6       0.68      21.06
```

The `if` statement

R evaluates statements in order. Flow control statements modify this. For conditional execution of some statements, use `if`:

```
> if (condition) {
+     TRUEcommands
+ }
```

To choose between two sets of statements, use `if else`:

```
> if (condition) {
+     TRUEcommands
+ } else {
+     FALSEcommands
+ }
```

Looping statements

Because R operates on vectors, in many cases loops are not necessary. But when they are, there are 3 versions: the `for` loop, the `while` loop, and the `repeat` loop.

```
> for (name in vector) {
+     commands
+ }
> while (condition) {
+     commands
+ }
> repeat {
+     commands
+ }
```

The `break` and `next` statements modify flow within a loop.

Outline

(Almost) Everything is an Object

- One of the great strengths of the S language is that almost everything is an object.
- Even *functions* are objects.
- You can do computations on functions: “computing on the language”.

Writing Functions

Functions have 3 parts: the header, the body, and the environment. For example,

```
> mymean <- function(x) {
+       n <- length(x)
+       sum <- sum(x)
+       return(sum / n)
+ }
```

- The header is `function(x)`.
- The body is the block of three lines in braces.
- The environment determines the context in which the lines are evaluated.

Function Headers

- The header of the function determines how it looks from the outside.
- It is formed as `function(`*`list of arguments`*`)`.
- The arguments are one of

name	a required parameter
name = value	an optional parameter with a default value
...	an optional list of unspecified parameters

Parameter Matching

- If you give names, partial matching may be used.
- Unnamed parameters are matched by position.
- Don't confuse yourself with code like this! —

```
> f <- function(firstarg, secondarg) {
+     cat("firstarg=", firstarg,
+         "secondarg=",secondarg,"\n")
+ }
> f(1,first=2)
firstarg= 2 secondarg= 1
```

The Function Body

- The body of a function is the statement following the header. Usually we put several statements in braces for a complex function, but the braces are not necessary if there's only one line.
- When you invoke (call) the function, an environment called the *evaluation frame* is created, containing objects corresponding to each argument in the header.
- Assignments in the body create objects in this environment.
- The value of the function is the last value calculated, or (better!) the value passed to `return()`.

Lazy Evaluation

Formal arguments to the function are created as *promises*, which are not evaluated until used.

```
> f <- function() cat("f was called\n")
> g <- function() cat("g was called\n")
> h <- function(x, y) y
> h(f(), g())
g was called
```

Promises

- You can see the expression associated with a promise using `substitute()`.

```
> showarg <- function(x) {
+     cat("The expression was",
+         deparse(substitute(x)), "\n")
+     cat("Its value is", x, "\n")
+ }
> showarg( 1 + 1 )
The expression was 1 + 1
Its value is 2
```

- Promises may also be created explicitly, using `delayedAssign()`, but for historical reasons `substitute()` doesn't work properly in the global environment.

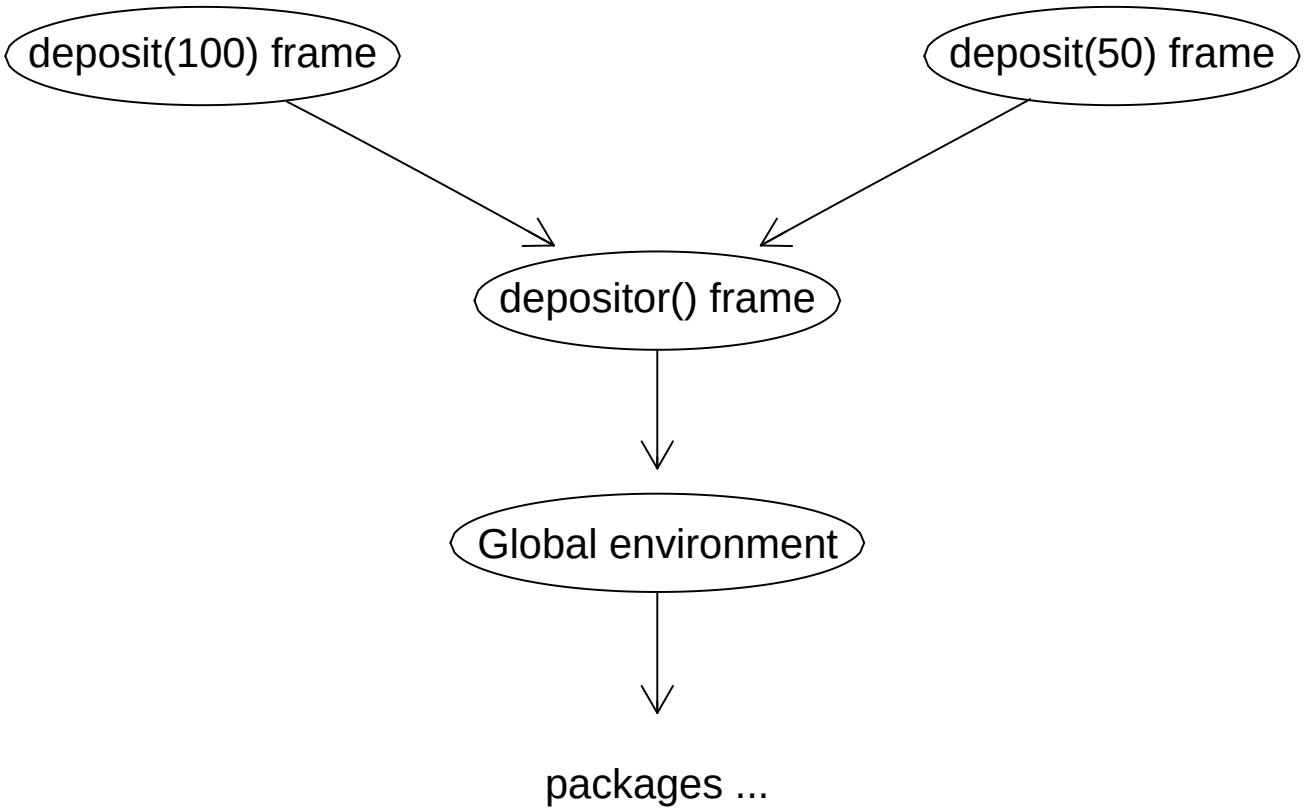
The Function Environment

- A function's environment is normally the environment in which the function was created:
  - the global environment for most user functions
  - the evaluation frame for functions created in other functions
  - the namespace of a package
- The parent of the evaluation frame is the function environment.

Environment Example

```
> depositor <- function() {
+     balance <- 0
+     function(amount) {
+         balance <<- balance + amount
+         return(balance) }
+ }
> deposit <- depositor()
> environment(deposit)$balance
[1] 0
> deposit(100)
[1] 100
> deposit(50)
[1] 150
```

Environment Example cont'd



Default Arguments

- Arguments passed to the function are evaluated in the environment of the caller.
- Default values for skipped arguments are evaluated in the evaluation frame.

```
> rect <- function(height = width/2,
+                  width = height*2) {
+   list(height=height, width=width)
+ }
> rect(height = 4)
$height
[1] 4

$width
[1] 8
```

Outline

Many Different Paradigms!

R has inherited 3 different ideas of "objects" from S:

1. Everything is an object of one of the fixed types described earlier.
2. R supports S version 3 objects: a fairly unstructured scheme based on the `class` attribute.
3. R supports S version 4 objects: a highly structured scheme inspired by the Common Lisp Object System (CLOS).

And that's not all...

There are other schemes in packages as well:

- The `R.oo` package supports a style of object somewhat similar to Delphi, Java or C++.
- The `proto` package implements prototype-based programming.
- The `tcltk` package provides an interface to a TCL/TK interpreter for GUI objects.
- The `RJava` package provides an interface to a Java interpreter.
- Other experimental packages...

S Version 3 Objects

This is the easiest and least sophisticated style of object-oriented programming.

- Objects have a `class` attribute (a character vector of class names).
- *Generic functions* (e.g. `print`) can automatically call *methods* based on the class of the first argument.

```
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
> head(methods(print))
[1] "print.acf"      "print.anova"
[3] "print.aov"      "print.aovlist"
[5] "print.ar"       "print.Arima"
```

- The name of the method is `<generic>.<class>`.

S Version 3 Example

```
> covariate <- rnorm(100)
> response <- 1.5 + 3.1*covariate + rnorm(100)
> fit3 <- lm(response ~ covariate)
> typeof(fit3)
[1] "list"
> class(fit3)
[1] "lm"
> fit3
Call:
lm(formula = response ~ covariate)

Coefficients:
(Intercept)    covariate
      1.490        3.272
```

```
> str(fit3)
List of 12
 $ coefficients : Named num [1:2] 1.49 3.27
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "covariate"
 $ residuals    : Named num [1:100] -0.832  0.879 -0.294 -0.259..
  ..- attr(*, "names")= chr [1:100] "1" "2" "3" "4" ...
 $ effects      : Named num [1:100] -15.645  32.210  -0.248  -0..
  ..- attr(*, "names")= chr [1:100] "(Intercept)" "covariate" "..
 $ rank         : int 2
 $ fitted.values: Named num [1:100] 7.86 1.87 1.12 2.26 1.26 ...
  ..- attr(*, "names")= chr [1:100] "1" "2" "3" "4" ...
 $ assign       : int [1:2] 0 1
 $ qr           :List of 5
  ..$ qr   : num [1:100, 1:2] -10 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0..
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:100] "1" "2" "3" "4" ...
[38 lines deleted]
  .. .. ..- attr(*, "predvars")= language list(response, covari..
  .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "..
  .. .. .. ..- attr(*, "names")= chr [1:2] "response" "covariate"
 - attr(*, "class")= chr "lm"
```

Common S3 generics

It is standard practice to supply methods for at least some of the following generics:

- `print()`, `summary()`, `plot()` for display
- `formula()`, `predict()` and `residuals()` for model-fitting functions
- `model.frame()`, `model.matrix()`, `case.names()`, `variable.names()` for programming with regression-like models.

What's Wrong with S3?

Some problems:

- Is `t.test()` the `t()` method for class `test`? (No.)
- What if several arguments have classes? (Only the first one matters.)
- Anyone can define new generics; how can I be sure my class will work with them? (I can't.)

S Version 4 Objects

A new object system was introduced by John Chambers in his *Programming with Data* book. This system is based on CLOS:

- Formal definitions: `setGeneric()`, `setMethod()`, etc.
- Function dispatch is based on the *signature* of the call to the generic: the classes of *all* parameters.
- Multiple inheritance is formalized in several ways.

Example

```
> library(stats4)
> x <- 0:10
> y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
> ll <- function(ymax=15, xhalf=6) {
+     if(ymax > 0 && xhalf > 0) {
+       -sum(dpois(y, lambda=ymax/(1+x/xhalf),
+                  log=TRUE))
+     } else NA
+ }
> print(fit4 <- mle(ll))
Call:
mle(minuslogl = ll)

Coefficients:
     ymax     xhalf
24.993092  3.057062
```

```
> str(fit4, strict.width="cut")
Formal class 'mle' [package "stats4"] with 8 slots
  ..@ call     : language mle(minuslogl = ll)
  ..@ coef     : Named num [1:2] 24.99  3.06
  .. ..- attr(*, "names")= chr [1:2] "ymax" "xhalf"
  ..@ fullcoef : Named num [1:2] 24.99  3.06
  .. ..- attr(*, "names")= chr [1:2] "ymax" "xhalf"
  ..@ vcov     : num [1:2, 1:2] 17.85 -3.72 -3.72  1.07
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:2] "ymax" "xhalf"
  .. .. ..$ : chr [1:2] "ymax" "xhalf"
  ..@ min      : num 28.6
  ..@ details  :List of 6
  .. ..$ par        : Named num [1:2] 24.99  3.06
  .. .. ..- attr(*, "names")= chr [1:2] "ymax" "xhalf"
  .. ..$ value      : num 28.6
  .. ..$ counts     : Named int [1:2] 25 18
  .. .. ..- attr(*, "names")= chr [1:2] "function" "gradient"
  .. ..$ convergence: int 0
  .. ..$ message    : NULL
  .. ..$ hessian    : num [1:2, 1:2] 0.203 0.706 0.706 3.388
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:2] "ymax" "xhalf"
  .. .. .. ..$ : chr [1:2] "ymax" "xhalf"
  ..@ minuslogl:function (ymax = 15, xhalf = 6)
  .. ..- attr(*, "source")= chr [1:6] "function(ymax=15, xhalf=..
  ..@ method   : chr "BFGS"
```

No more S4 Today...

We won't be studying the details of S4 methods:

- It is a very rich system; it would take a whole day by itself!
- I don't use it much: I find it *too* rich.
- I think it still has the S3 problem that anyone can define new generics for which my class will fail, but this is open to debate.