R for Programmers

John D. Cook[*]

September 17, 2007

Contents

[*] `http://www.JohnDCook.com`

1 Introduction

I have written software professionally in perhaps a dozen programming languages, and the hardest language for me to learn has been R. The language is actually fairly simple, but it is unconventional. These notes are intended to make the language easier to learn for someone used to more commonly used languages such as C++, Java, Perl, etc.

R is more than a programming language. It is an interactive environment for doing statistics. I find it more helpful to think of R as *having* a programming language than *being* a programming language. The R language is the scripting language for the R environment, just as VBA is the scripting language for Microsoft Excel. Some of the more unusual features of the R language begin to make sense when viewed from this perspective.

This document is a work in progress. Corrections and comments are welcome.

2 Assignment and underscore

The assignment operator in R is `<-` as in `e <- m*c*c`. It is also possible, though uncommon, to reverse the arrow and put the receiving variable on the right, as in `m*c*c -> e`. It is sometimes possible to use `=` for assignment, though I don't understand when this is and is not allowed. Most people avoid the issue by always using the arrow.

However, when supplying default function arguments or calling functions with named arguments, you *must* use the `=` operator and cannot use the arrow.

3 Variable name gotchas

At some time in the past R, or its ancestor S, used underscore as assignment. This meant that the C convention of using underscores as separators in multi-word variable names was not only disallowed but produced strange side effects. For example, `first_name` would not be a single variable but rather the instruction to assign the value of `name` to the variable `first`! S-PLUS still follows this use of the underscore. However, R allows underscore as a variable character and not as an assignment operator.

Because historically the underscore was not allowed as a variable character, the convention arose to use a dot as a name separator. Unlike its use in object oriented languages, the dot character in R has no special significance. (I think there's an exception to this, but I can't think what it is at this time.)

R uses `$` in a manner analogous to the way other languages use dot.

R has several one-letter reserved words: `c`, `q`, `s`, `t`, `C`, `D`, `F`, `I`, and `T`.

4 Vectors

The primary data type in R is the vector. Before describing how vectors work in R, it is helpful to distinguish two ideas of vectors in order to set the correct expectations.

The first idea of a vector is what I will call a computer science (CS) vector. The CS vector is a container, an ordered collection of numbers with no other structure, such as the `vector<>` container in C++. The length of a vector is the number of elements in the container. Operations are applied componentwise. For example, given two vectors `x` and `y` of equal length, `x*y` would be the vector whose nth component is the product of the nth components of `x` and `y`. Also, `log(x)` would be the vector whose nth component is the logarithm of the nth component of `x`.

The other idea of a vector is a mathematical vector, an element of a vector space. In this context "length" means geometrical length determined by an inner product; the number of components is called "dimension." In general, operations are not applied componentwise. The expression `x*y` is a single number, the inner product of the vectors. The expression `log(x)` is meaningless. The `Vector` class in M. D. Anderson's BiostatGeneral library behaves more like a mathematical vector than a CS vector.

A vector in R is a CS vector, a statistician's collection of data, not a physicist's mathematical vector. The R language is designed around the assumption that a vector is an ordered set of measurements rather than a geometrical position or a physical state. (R supports mathematical vector operations, but they are secondary in the design of the language.) This helps explain, for example, R's otherwise inexplicable vector recycling feature.

What would you expect if asked a computer to add a vector of length 22 and a vector of length 45? If you have mathematical vectors in mind, you would expect an error. The programming language should throw an exception because the programmer has made an error and the program is now in an undefined state.

However, R allows adding two vectors regardless of their relative lengths. The elements of the shorter summand are recycled as often as necessary to create a vector the length of the longer summand. This is not attempting to add physical vectors that are incompatible for addition, but rather a syntactic convenience for manipulating sets of data.

The R language has no provision for scalars, nothing like a `double` in C-derived languages. The only way to represent a single number in a variable is to use a vector of length one. And while it is possible to iterate through vectors as one might to in a `for` loop in C, it is usually clearer and more efficient in R to operate on vectors as a whole.

Vectors are created using the `c` function. For example, `p <- c(2,3,5,7)` sets `p` to the vector containing the first four prime numbers.

Vectors in R are indexed starting with 1 and matrices in are stored in column-major order. In both of these ways R resembles FORTRAN.

Elements of a vector can be accessed using `[]`. So in the above example, `p[3]` is 5.

Vectors automatically expand when assigning to an index past the end of the vector, as in Perl.

Negative indices are legal, but they have a very different meaning than in some other languages. If `x` is an array in Python or Perl, `x[-n]` returns the nth element from the end of the vector. In R, `x[-n]` returns a copy of `x` with the nth element removed.

5 Sequences

The expression `seq(a, b, n)` creates a closed interval from `a` to `b` in steps of size `n`. For example, `seq(1, 10, 3)` returns the vector containing 1, 4, 7, and 10. This is similar to `range(a, b, n)` in Python, except Python uses open intervals and so the 10 would not be included in this example. The step size argument `n` defaults to 1 in both R and Python.

The notation `a:b` is an abbreviation for `seq(a, b, 1)`.

The notation `seq(a, b, length=n)` is a variation that will set the step size to $(b - a - 1)/n$ so that the sequence has n points.

6 Types

The type of a vector is the type of the elements it contains and must be one of the following: `logical`, `integer`, `double`, `complex`, `character`, or `raw`. All elements of a vector must have the same underlying type. This restriction does not apply to lists.

Type conversion functions have the naming convention `as.xxx` for the function converts its argument to type `xxx`. For example, `as.integer(3.2)` returns the integer 3, and `as.character(3.2)` returns the string "3.2".

7 Boolean operators

You can input `T` or `TRUE` for true values and `F` or `FALSE` for false values.

The operators `&` and `|` apply element-wise on vectors. The operators `&&` and `||` are often used in conditional statements and use lazy evaluation as in C: the operators will not evaluate their second argument if the return value is determined by the first argument.

8 Lists

Lists are like vectors, except elements need not all have the same type. For example, the first element of a list could be an integer and the second element be a string or a vector of Boolean values.

Lists are created using the `list` function. Elements can be access by position using `[[]]`. Named elements may be accessed either by position or by name.

Named elements of lists act like C structs, except a dollar sign rather than a dot is used to access elements. For example, consider

```
a <- list(name="Joe", 4, foo=c(3,8,9))
```

Now `a[[1]]` and `a$name` both equal the string “Joe”.

If you attempt to access a non-existent element of a list, say `a[4]` above, you will get an error. However, you can *assign* to a non-existent element of a list, thus extending the list. If the index you assign to is more than one past the end of the list, intermediate elements are created and assigned `NULL` values. You can also assign to non-existent named fields, such as saying `a$baz = TRUE`.

9 Matrices

In a sense, R does not support matrices, only vectors. But you can change the dimension of a vector, essentially making it a matrix.

For example, `m <- array( c(1,2,3,4,5,6), dim=c(2,3) )` creates a matrix `m`. However, it may come as a surprise that the first row of `m` has elements 1, 3, and 5. This is because by default, R fills matrices by column, like FORTRAN. To fill `m` by row, add the argument `by.row = T` to the call to the `array` function.

10 Missing values and `NaN`s

As in other programming languages, the result of an operation on numbers may return `NaN`, the symbol for “not a number.” For example, an operation might overflow the finite range of a machine number, or a program might request an undefined operation, such as dividing by zero.

R also has a different type of non-number, `NA` for “not applicable.” `NA` is used to indicate missing data, and is unfortunately fairly common in data sets. `NA` in R is similar to `NULL` in SQL or nullable types in C#. However, one must be more careful about `NA` values in R than about nulls in SQL or C#. The designer of database or the author of a piece of C# code specifies which values are nullable and can avoid the issue by simply not allowing such values. The author of an R function, however, has no control over the data his function will receive because `NA` is a legal value inside an R vector. There is no way to specify that a function takes only vectors with non-null components. You must handle `NA` values, even if you handle them by returning an error.

The function `is.nan` will return `TRUE` for those components of its argument that are `NaN`. The function `is.na` will return true for those components that are `NA` *or* `NaN`.

11 Comments

Comments begin with `#` and continue to the end of the line, as in Python or Perl. Comments may not appear within function argument definitions.

12 Functions

Function definition syntax in R is similar to that in JavaScript. Example:

```
f <- function(a, b)
{
    return (a+b)
}
```

The function `function` returns a function, which is usually assigned to a variable, `f` in this case, but need not be. You may use the `function` statement to create an anonymous function (lambda expression).

Note that `return` is a function; its argument must be contained in parentheses, unlike C where parentheses are optional.

Default values are defined similarly to C++. In the following example, `b` is set to 10 by default.

```
f <- function(a, b=10)
{
    return (a+b)
}
```

So `f(5, 1)` would return 6, and `f(5)` would return 15. R allows more sophisticated default values than does C++. A default value in R need not be a static types and could, for example, be a function of other arguments.

C++ requires that if an argument has a default value then so do all values to the right. This is not the case in R, though it is still a good idea. The function definition

```
f <- function(a=10, b)
{
    return (a+b)
}
```

is legal, but calling `f(5)` would cause an error. The argument `a` would be assigned 5, but no value would be assigned to `b`. The reason such a function definition is not illegal is that one could still call the function with one *named* argument. For example, `f(b=2)` would return 12.

Function arguments are passed by value. The most common mechanism for passing variables by reference is to use non-local variables. (Not necessary global variables, but variables in the calling routine's scope.) A safer alternative is to explicitly pass in all needed values and return a list as output.

13 Scope

R uses lexical scoping while S-PLUS uses static scope. The difference can be subtle, particularly when using closures.

Since variables cannot be declared – they pop into existence on first assignment – it is not always easy to determine the scope of a variable.

You cannot tell just by looking at the source code of a function whether a variable is local to that function.

14 Misc.

Here are a few miscellaneous facts about R you may find useful.

`help(fctn)` displays help on `fctn` as in Python.

To invoke complex arithmetic, add `0i` to a number. For example, `sqrt(-1)` returns `NaN`, but `sqrt(-1 +0i)` returns `0 + 1i`.

`sessionInfo()` prints the R version, OS, packages loaded, etc.

`ls()` prints which objects are defined.

`rm(list=ls())` clears all defined objects.

`win.graph()` opens a new plotting window without overwriting the previous one.

The function `sort()` does not change its argument.

Distribution function prefixes `d`, `p`, `q`, `r` stand for density (PDF), probability (CDF), quantile (CDF^{-1}), and random sample. For example, `dnorm` is the density function of a normal random variable and `rnorm` generates a sample from a normal random variable.