CSC 120: Computer Science for the Sciences (R section)

Radford M. Neal, University of Toronto, 2016

http://www.cs.utoronto.ca/∼radford/csc120/

Week 6

Environments

An R *environment* is a collection of variables and their current values.

The *global* environment contains variables that are created when you assign to a name in a command typed at the R console (or as if typed in an R script).

For example, typing the command below creates (if it didn't exist already) a variable in the global environment named `fred`:

```
> fred <- 1+2
```

Calling a function creates a *local* environment used for just that call. Assignments inside the function create or change variables in that environment — below, the assignment to `fred` inside `f` changes `fred` in the local, not global, environment:

```
> f <- function (x) { fred <- 2*x; fred+1 }
> fred
[1] 3
> f(100)
[1] 201
> fred
[1] 3
```

Listing and Removing Variables

You can see what variables exist in the environment that is currently being used with the `ls` function, which returns a vector of strings with the names of variables.

You can remove a variable from the current environment with `rm`.

Here's an example (which assumes you haven't already defined other variables in the global environment):

```
> a <- 1
> b <- 2
> ls()
[1] "a" "b"
> rm(a)
> ls()
[1] "b"
> a
Error: object 'a' not found
```

Note: After `x <- "b"`, calling `rm(x)` removes variable `x`, *not* variable `b`.

Function Arguments in the Local Environment

When a function is called, all its arguments become variables in its local environment. Their values are what is was specified in the call of the function, or their default values if they were not specified.

We can see this by printing the result of `ls` inside a function:

```
> f <- function (x,y=100,z=1000) { print(ls()); x + y + z }
> f(7,z=10)
[1] "x" "y" "z"
[1] 117
```

If we create new variables by assignment, they also are in the local environment:

```
> g <- function (x,y=100,z=1000) { a <- x + y + z; print(ls()); a }
> g(7)
[1] "a" "x" "y" "z"
[1] 1107
```

The global environment isn't changed when local variables are created for arguments or by assignment. So after doing the above, in a new R session, we see

```
> ls()
[1] "f" "g"
```

Local and Global Variable References

When you reference a variable inside a function, it refers to the *local* variable of that name, if it exists, and if not, to the *global* variable of that name, if it exists.

Here's an example:

```
> f <- function (xyz,def) {
+     print (abc)  # refers to the global variable 'abc'
+     print (xyz)  # refers to the local variable (argument) 'xyz'
+     print (def)  # refers to the local variable (argument) 'def'
+     xyz + def + abc
+ }
>
> abc <- 1
> def <- 2
>
> f(200,3000)
[1] 1
[1] 200
[1] 3000
[1] 3201
```

Changing Local and Global Variables Inside a Function

Assigning a value to a name with `<-` (or with `=`) from inside a function creates or changes the *local* variable with that name. Assigning a value to a name with `<<-` creates or changes the *global* variable with that name. Here's an example:

```
> g <- function () {
+     x <- a     # creates a local variable 'x', with value from global 'a'
+     a <- 10    # creates a local variable 'a'; global 'a' is not affected
+     b <<- 300  # changes the global variable 'b'; doesn't create a local 'b'
+     a + b + x  # here, 'a' refers to the new local 'a', not the global 'a'
+ }
> g()
Error in g() : object 'a' not found
> a <- 100
> b <- 200
> g()
[1] 410
> a
[1] 100
> b
[1] 300
> x
Error: object 'x' not found
```

Assigning to Arguments Doesn't Change Them

Since assignments with `<-` inside a function change only the *local* environment, assigning to a function argument doesn't change what the caller passed.

For example:

```
> h <- function (x) { x[1] <- 0; sum(x) } # sum all but first element
> a <- c(3,4,1,7)
> h(a)
[1] 12
> a          # the global variable 'a' was not changed
[1] 3 4 1 7
> x <- c(10,20,30)
> h(x)
[1] 50
> x          # global 'x' unchanged - not the same as the local 'x'!
[1] 10 20 30
```

Exception: R has some "special" functions that do alter their arguments — for example, as we've seen, `rm(x)` actually removes `x`!

When and How to Use Local and Global Variables

When writing a function, you should try to

- Separate *what* the function does from *how* it does it, so someone using the function only needs to understand the "what".
- Make what the function does be easy to describe and understand.
- Make what the function does be general, so it will be useful in many contexts.

Functions should usually get input from their arguments, not global variables — they're then more generally useful, as it's easy to use different arguments in calls.

Functions should usually assign only to local variables. Putting intermediate results in global variables unnecessarily makes "how" the function works change what effects it has. Returning information to the caller via global variables makes it harder to use the function in a general way.

There are exceptions:

- If many functions all refer to the same data, having them all refer to a global `data` variable may be easier than passing a `data` argument to all of them.
- Assigning some intermediate result to a global variable may help when debugging a program (but take it out once the program is working).

Example: Separating Good Data From Bad — Version 1

This function gets inputs from its arguments, and returns results in its return value:

```
# READ DATA FROM A FILE, AND ELIMINATE VALUES EXCEEDING A THRESHOLD.
# Takes as its arguments the name of a file to read numerical data
# from and the threshold above which data is considered bad. Returns
# a list with element 'good' containing the vector of data values read
# that were no more than the threshold, and element 'bad' containing
# the data values read that were above the threshold.

read_data_1 <- function (file, threshold) {
    data_read <- scan(file)
    good_data <- numeric(0)
    bad_data <- numeric(0)
    for (i in 1:length(data_read)) {
        if (data_read[i] <= threshold)
            good_data <- c (good_data, data_read[i])
        else
            bad_data <- c (bad_data, data_read[i])
    }
    list (good=good_data, bad=bad_data)
}
```

Example: Separating Good Data From Bad — Version 2

This version uses global variables for some of its input and output:

```
# READ DATA FROM A FILE, AND ELIMINATE VALUES EXCEEDING A THRESHOLD.
# Takes as its argument the name of a file to read numerical data
# from.  Returns the vector of data values read, but with values
# greater than the global variable 'threshold' eliminated.  The global
# variable 'bad_data' is set to the vector of values that were above
# the threshold.

read_data_2 <- function (file) {
    data_read <- scan(file)
    good_data <- numeric(0)
    bad_data <<- numeric(0)
    for (i in 1:length(data_read)) {
        if (data_read[i] <= threshold)
            good_data <- c (good_data, data_read[i])
        else
            bad_data <<- c (bad_data, data_read[i])
    }
    good_data
}
```

How These Two Versions of the Function Would be Used

Here's how we would use the first version of this function:

```
> data <-
+  read_data_1("http://www.cs.utoronto.ca/~radford/csc120/test_data",10)
Read 11 items
> data$good
[1] 1 3 0 5 9 2 3 1
> data$bad
[1] 11 12 18
```

Here's how we would use the second version:

```
> threshold <- 10
> good_data <-
+  read_data_2("http://www.cs.utoronto.ca/~radford/csc120/test_data")
Read 11 items
> good_data
[1] 1 3 0 5 9 2 3 1
> bad_data
[1] 11 12 18
```

Is there Any Good Reason to Use the Second Version?

The first version — providing inputs via function arguments, and outputs via the function return value — should be how you usually write functions.

Are there any good arguments for the second version?

- Perhaps we almost always use the same threshold of 10. With the second version, we can assign 10 to the global variable `threshold`, and then not have to set it most times.
- Perhaps we usually don't care about the bad data (but do want to look at it once in a while). With the second version, we can usually ignore the fact that the `bad_data` variable was set, but it's there if we need it.

These arguments seem most persuasive if we originally wrote a `read_data` function that always used a threshold of 10, and that didn't save the bad data. Then we realized that we sometimes need a different threshold and sometimes need to look at the bad data.

But we'd rather not have to change all the uses of the old `read_data` function to add a second argument and to extract the good data from a list that is returned.

Is There a Better Way?

But in that situation, is there some better way of changing the `read_data` function?

If we usually use a threshold of 10, we can make that the *default* value for the `threshold` argument, and not have to specify it each time. We just use a function definition like

```
read_data <- function (file, threshold=10) { ... }
```

That's easier and less error-prone than using a global `threshold` variable.

But there isn't as easy a way of letting the caller have access to the bad data if it needs it, without changing how the good data is returned.

(There's an advanced R feature of "attributes" that might help, but it wouldn't be a perfect solution.)

One final note: The way these functions are written, they will be very slow if the amount of data is large, because `c(good_data, data_read[i])` gets slower and slower as there is more data. We'll see a better way later...