Statistical Computing (36-350)

Lecture 7: More Design, and Scoping

Cosma Shalizi and Vincent Vu

21 September 2011

Agenda

- The scope of names: what they mean where
- Example: The last homework

ABSOLUTELY ESSENTIAL READING FOR FRIDAY: Sec. 4.4 of the textbook

MERELY USEFUL READING: Chapter 3

HOMEWORK 4 will be on the website later today, and due at 11:59 pm on Tuesday, 27 September 2011

Looking Up Names

When R sees a variable name, it needs to look up what value goes with that name

It consults the **environment**, a list of name/value pairs

If the name isn't in the current environment, it looks in the larger, **parent** environment, and so on to the global environment

The global environment is what we interact with at the terminal

name	value
...	...
x	c(1,2,3,4)
y	3.7
cats	*a data frame with three columns*
psi	`function(x,c=1) {ifelse(abs(x)<c,c x,x^ 2)}`
parent	*a pointer telling R where*
environment	*to look in its memory*

The Scope of Names

Because R "goes up the chain", if this environment and its parent share a name, R uses the local name — that's the **scope** of the assignment

When we make an assignment with $<-$ or $=$, we only modify the *current* environment

Changes in this environment do not affect its parents

Changes in the parents affect this one, *unless* over-ridden locally

Functions and Environments

Inside a function definition, we have a new environment
Giving a function named arguments means that, inside the function, those names refer to the argument values
The same names might refer to something else outside; doesn't matter
Parent environment is the one of definition, not execution

Examples:

```
> f <- function(x) {
+   f <- x^2*exp(-x^2)
+   return(f)
+ } # Assigns this function the name "f"
> f  # What value goes with the name "f"?
function(x) {
  f <- x^2*exp(-x^2)
  return(f)
}
> x <- 3  # Assigns x the value 3, globally
> f(7)    # Assigns x the value 7, INSIDE f
[1] 2.569014e-20
> f(x)    # Did not change x globally
[1] 0.001110688
> f       # Also did not change the global value of "f"
function(x) {
  f <- x^2*exp(-x^2)
  return(f)
}
```

More examples:

```
g <- function(x) {
  eta <- 2*x*exp(-x^2)
  kappa <- -2*x^3*exp(-x^2)
  return(eta+kappa)
}

h <- function(y) {
  return(eta*sin(y))
}
```

Q: what happens if we run

```
g(3)
h(pi)
```

A: Depends on what `eta` is in the parent environment!

Environment of definition vs. execution

```
> wheel <- function(r) {2*pi*r}
> wheel.inside.wheel <- function(r,pi) { return(wheel(r)) }
> wheel(1)
[1] 6.283185
> wheel.inside.wheel(1,3)
[1] 6.283185
```

vs.

```
> wheel.inside.wheel <- function(r,pi) {
+   wheel <- function(r) { 2*pi*r }
+   return(wheel(r))
+ }
> wheel.inside.wheel(1,pi)
[1] 6.283185
> wheel.inside.wheel(1,3)
[1] 6
```

Why Does R Do This To Us?

No interference between the insides of separate functions
$\therefore$ no restrictions on naming arguments, or on using other people's code, whatever their internal names
Looking to larger environments is a convenience: share information by nesting functions, and allow global constants

Design Implications

Compartmentalize information
Sometimes encourages nested functions

Example: The last homework

First sketch of parts:

```
my.nls <- function(params, data.values, controls) {
  until the gradient is small or we run out of time
    find the gradient at the current parameter guess
    adjust the parameters against the direction of the gradient
    if the gradient is small, stop, otherwise continue
  gather up return values
}
```

(not really code!)

Translate into code:

```
my.nls <- function(params, N=gmp$pop, Y=gmp$pcgmp, stopping.deriv,
  max.iterations, step.scale,deriv.increments) {
  for (iteration in 1:max.iterations) {
    gradient <- mse.grad(params,deriv.increments)
    params <- params - step.scale*gradient
    if(all(abs(gradient)) < stopping.deriv) { break() }
  }
  converged <- (iteration < max.iterations)
  fit <- list(params=params,gradient=gradient,iterations=iteration,
    converged=converged)
  return(fit)
}
```

needs an `mse.grad` function

Skipping preliminary analysis:

```
mse.grad <- function(params,deriv.increments) {
  p <- length(params)
  stopifnot(p==length(deriv.increments))
  gradient <- vector(length=p)
  mse.0 <- mse(params)
  for (i in 1:p) {
    new.params <- params
    new.params[i] <- params[i] + deriv.increments[i]
    new.mse <- mse(new.params)
    gradient[i] <- (new.mse-mse.0)/deriv.increments[i]
  }
  return(gradient)
}
```

Could just do each param. separately if we know how many there are
Could further vectorize, though this is OK
Needs an `mse()` function

Finally, the mse function:

```
mse <- function(params,N=gmp$pop,Y=gmp$pcgmp) {
  predictions <- params[1]*N^params[2]
  mse <- mean((Y-predictions)^2)  # Why doesn't this clobber the function?
  return(mse)
}
```

Integration

Problem: how to get `mse.grad` to notice if the data changes?
Solution 1: change arguments to `mse.grad`, to include data, which it passes to `mse`
Solution 2: manipulate scope, remembering environment of definition is what matters
Solution 3: functions as arguments (a later lecture)
Let's look at solution 2

All together:

```
my.nls.2 <- function(params, N=gmp$pop, Y=gmp$pcgmp, stopping.deriv,
  max.iterations, step.scale,deriv.increments) {
  mse <- function(params) {return(mean((Y-params[1]*N^params[2])^2)) }
  mse.grad <- function(params,deriv.increments) {
    p <- length(params); stopifnot(p==length(deriv.increments))
    gradient <- vector(length=p)
    mse.0 <- mse(params)
    for (i in 1:p) {
      new.params <- params
      new.params[i] <- params[i] + deriv.increments[i]
      new.mse <- mse(new.params)
      gradient[i] <- (new.mse-mse.0)/deriv.increments[i]
    }
    return(gradient)
  }
  for (iteration in 1:max.iterations) {
    gradient <- mse.grad(params,deriv.increments)
    params <- params - step.scale*gradient
    if(all(abs(gradient) < stopping.deriv)) { break() }
  }
  fit <- list(params=params,gradient=gradient,iterations=iteration,
    converged=(iteration < max.iterations))
  return(fit)
}
```

Summary

1. Environments control the values of names
2. Values and assignments in local environments over-rule more global ones
3. "Local" goes by definition, not execution
4. Use scoping to control information sharing between functions