Lecture 7: More on Functions

36-350

17 September 2014

How We Extend Functions

- Multiple functions: Doing different things to the same object
- Sub-functions: Breaking up big jobs into small ones
- Example: Back to resource allocation

Reading for Friday: 1.3, 7.3–7.5, 7.11, 7.13 of Matloff (skipping "extended examples")

In our last episode . . .

Functions tie together related commands:

```
my.clever.function <- function(an.argument,another.argument) {
  # many lines of clever calculations
  return(important.result)
}
```

Inputs/arguments and outputs/return values define the interface

A user only cares about turning inputs into outputs correctly

Why You Have to Write More Than One Function

Meta-problems:

- You've got more than one problem
- Your problem is too hard to solve in one step
- You keep solving the same problems

Meta-solutions:

- Write multiple functions, which rely on each other
- Split your problem, and write functions for the pieces
- Solve the recurring problems once, and re-use the solutions

Writing Multiple Related Functions

Statisticians want to do lots of things with their models: estimate, predict, visualize, test, compare, simulate, uncertainty, . . .

Write multiple functions to do these things

Make the model one object; assume it has certain components

Consistent Interfaces

- Functions for the same kind of object should use the same arguments, and presume the same structure
- Functions for the same kind of task should use the same arguments, and return the same sort of value

(to the extent possible)

Keep related things together

- Put all the related functions in a single file
- Source them together
- Use comments to note ***dependencies***

Power-Law Scaling for Urban Economies (cont'd.)

Remember the model:

$$Y = y_0 N^a + \text{noise}$$

$$\text{(output per person)} =$$

$$\text{(baseline)(population)}^{\text{scaling exponent}} + \text{noise}$$

Estimated parameters a, y_0 by minimizing the mean squared error

Exercise: Modify the estimation code from last time so it returns a list, with components `a` and `y0`

Example: Predicting from a Fitted Model

Predict values from the power-law model:

```
# Predict response values from a power-law scaling model
# Inputs: fitted power-law model (object), vector of values at which to make
  # predictions at (newdata)
# Outputs: vector of predicted response values
predict.plm <- function(object, newdata) {
  # Check that object has the right components
  stopifnot("a" %in% names(object), "y0" %in% names(object))
  a <- object$a
  y0 <- object$y0
  # Sanity check the inputs
  stopifnot(is.numeric(a),length(a)==1)
  stopifnot(is.numeric(y0),length(y0)==1)
  stopifnot(is.numeric(newdata))
  return(y0*newdata^a)  # Actual calculation and return
}
```

Example: Predicting from a Fitted Model

```
# Plot fitted curve from power law model over specified range
# Inputs: list containing parameters (plm), start and end of range (from, to)
# Outputs: TRUE, silently, if successful
# Side-effect: Makes the plot
plot.plm.1 <- function(plm,from,to) {
  # Take sanity-checking of parameters as read
  y0 <- plm$y0 # Extract parameters
  a <- plm$a
  f <- function(x) { return(y0*x^a) }
  curve(f(x),from=from,to=to)
  # Return with no visible value on the terminal
  invisible(TRUE)
}
```

Example: Predicting from a Fitted Model

When one function calls another, use `...` as a meta-argument, to pass along unspecified inputs to the called function:

```
plot.plm.2 <- function(plm,...) {
  y0 <- plm$y0
  a <- plm$a
  f <- function(x) { return(y0*x^a) }
  # from and to are possible arguments to curve()
  curve(f(x), ...)
  invisible(TRUE)
}
```

Sub-Functions

Solve big problems by dividing them into a few sub-problems

- Easier to understand: get the big picture at a glance
- Easier to fix, improve and modify: tinker with sub-problems at leisure
- Easier to design: for future lecture
- Easier to re-use solutions to recurring sub-problems

Rule of thumb: A function longer than a page is probably too long

Sub-Functions or Separate Functions?

Defining a function inside another function

- Pros: Simpler code, access to local variables, doesn't clutter workspace
- Cons: Gets re-declared each time, can't access in global environment (or in other functions)
- Alternative: Declare the function in the same file, source them together

Rule of thumb: If you find yourself writing the same code in multiple places, make it a separate function

Example: Plotting a Power-Law Model

Our old plotting function calculated the fitted values

But so does our prediction function

```
plot.plm.3 <- function(plm,from,to,n=101,...) {
  x <- seq(from=from,to=to,length.out=n)
  y <- predict.plm(object=plm,newdata=x)
  plot(x,y,...)
  invisible(TRUE)
}
```

Recursion

Reduce the problem to an easier one of the same form:

```
my.factorial <- function(n) {
  if (n == 1) {
    return(1)
  } else {
    return(n*my.factorial(n-1))
  }
}
```

Recursion

or multiple calls:

```
fib <- function(n) {
  if ( (n==1) || (n==0) ) {
   return(1)
  } else {
   return (fib(n-1) + fib(n-2))
  }
}
```

Exercise: Convince yourself that any loop can be replaced by recursion; can you always replace recursion with a loop?

Cleaner Resource Allocation

```
planner <- function(output,factory,available,slack,tweak=0.1) {
  needed <- plan.needs(output,factory)
  if (all(needed <= available) && all(available-needed <= slack)) {
    return(list(output=output,needed=needed))
  }
  else {
```

```
    output <- adjust.plan(output,needed,available,tweak)
    return(planner(output,factory,available,slack))
  }
}

plan.needs <- function(output,factory) { factory %*% output }

adjust.plan <- function(output,needed,available,tweak) {
  if (all(needed >= available)) { return(output*(1-tweak)) }
  if (all(needed < available)) { return((1+tweak)) }
  return(output*runif(n=length(output),min=1-tweak,max=1+tweak))
}
```

Summary

- ***Multiple functions*** let us do multiple related jobs, either on the same object or on similar ones
- ***Sub-functions*** let us break big problems into smaller ones, and re-use the solutions to the smaller ones
- ***Recursion*** is a powerful way of making hard problems simpler

Next time: Designing functions from the top down