Lecture 4: Writing and Using Functions

S. Morteza Najibi

Shiraz University

April 15, 2019

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
- Getting data into and out of the system when it's already in R format
- Import and export when the data is already very structured and machine-readable
- Dealing with less structured data

In our last episode . . .

Scientific Data Analysis Team

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

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

Keep related things together

- Put all the related functions in a single file
- use `source(your_function.R)` to use them together
- Use comments to note ***dependencies***

Remember the model:

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

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

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

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

Scientific Data Analysis Team

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

Scientific Data Analysis Team

```
# 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

Scientific Data Analysis Team

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)
}
```

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
- 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

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)
}
```

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))
  }
}
```

or multiple calls:

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

Cleaner Resource Allocation

Scientific Data Analysis Team

```
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

Import and Export Data – Agenda

- Getting data into and out of the system when it's already in R format
- Import and export when the data is already very structured and machine-readable
- Dealing with less structured data

Reading Data from R

- You can load and save R objects
  - R has its own format for this, which is shared across operating systems
  - It's an open, documented format if you really want to pry into it
- `save(thing, file="name")` saves `thing` in a file called `name` (conventional extension: `rda` or `Rda`)
- `load("name")` loads the object or objects stored in the file called `name`, *with their old names*

```
gmp <- read.table("https://smnajibi.github.io/statcomp/data/gmp.dat")
#gmp <- read.table("gmp.dat")
gmp$pop <- round(gmp$gmp/gmp$pcgmp)
save(gmp,file="gmp.Rda")
rm(gmp)
exists("gmp")
```

```
## [1] FALSE
```

```
load(file="gmp.Rda")
colnames(gmp)
```

```
## [1] "MSA"   "gmp"   "pcgmp" "pop"
```

- We can load or save more than one object at once; this is how RStudio will load your whole workspace when you're starting, and offer to save it when you're done
- Many packages come with saved data objects; there's the convenience function `data()` to load them

```
 data(cats,package="MASS")
summary(cats)
```

```
##  Sex         Bwt              Hwt
##  F:47   Min.   :2.000   Min.   : 6.30
##  M:97   1st Qu.:2.300   1st Qu.: 8.95
##         Median :2.700   Median :10.10
##         Mean   :2.724   Mean   :10.63
##         3rd Qu.:3.025   3rd Qu.:12.12
##         Max.   :3.900   Max.   :20.50
```

Non-R Data Tables

- Tables full of data, just not in the R file format
- Main function: `read.table()`
  - Presumes space-separated fields, one line per row
  - Main argument is the file name or URL
  - Returns a dataframe
  - Lots of options for things like field separator, column names, forcing or guessing column types, skipping lines at the start of the file...
- `read.csv()` is a short-cut to set the options for reading comma-separated value (CSV) files
  - Spreadsheets will usually read and write CSV

Writing Dataframes

- Counterpart functions `write.table()`, `write.csv()` write a dataframe into a file
- Drawback: takes a lot more disk space than what you get from `load` or `save`
- Advantage: can communicate with other programs, or even edit manually

Less Friendly Data Formats

- The `foreign` package on CRAN has tools for reading data files from lots of non-R statistical software
- Spreadsheets are special

Summary

- Loading and saving R objects is very easy
- Reading and writing dataframes is pretty easy