Lecture 6: Writing and Using Functions

Statistical Computing, 36-350

Monday September 28, 2015

Outline

- Defining functions: tying related commands into bundles
- Interfaces: controlling what the function can see and do
- Example: parameter estimation code

Why functions?

Data structures tie related values into one object

Functions tie related commands into one object

In both cases: easier to understand, easier to work with, easier to build into larger things

Huber loss function

```
# "Huber" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x)
# Outputs: vector with x^2 for small entries, 2|x|-1 for large ones
psi.1 = function(x) {
  psi = ifelse(x^2 > 1, 2*abs(x)-1, x^2)
  return(psi)
}
```

Our functions get used just like the built-in ones:

```
z = c(-0.5,-5,0.9,9)
psi.1(z)
```

```
## [1]  0.25  9.00  0.81 17.00
```

Go back to the declaration and look at the parts:

```
# "Huber" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x)
# Outputs: vector with x^2 for small entries, 2|x|-1 for large ones
psi.1 = function(x) {
  psi = ifelse(x^2 > 1, 2*abs(x)-1, x^2)
  return(psi)
}
```

Two interfaces: the **inputs** or **arguments**; the **outputs** or **return value**

Calls other functions `ifelse()`, `abs()`, operators `^` and `>`. Could have also called other functions we've written

`return()` says what the output is. With no explicit return statement, the function just outputs what's on the last line

Comments: not required by R, but a very good idea! One-line description of purpose; listing of arguments; listing of outputs

What should be a function?

- Things you're going to re-run, especially if it will be re-run with changes to arguments
- Chunks of code you keep highlighting and hitting return on
- Chunks of code which are small parts of bigger analyses
- Chunks of code that are very similar to other chunks

Multiple arguments

```
# "Hubger" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x), scale for crossover (c)
# Outputs: vector with x^2 for small entries, 2c|x|-c^2 for large ones
psi.2 = function(x, c=1) {
  psi = ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(psi)
}
```

```
psi.1(z)
```

```
## [1]  0.25  9.00  0.81 17.00
```

```
psi.2(z,1) # Same
```

```
## [1]  0.25  9.00  0.81 17.00
```

Default values get used if arguments are missing:

```
psi.2(z) # Same
```

```
## [1]  0.25  9.00  0.81 17.00
```

Named arguments can go in any order when explicitly labeled:

```
psi.2(z,1)
```

```
## [1]  0.25  9.00  0.81 17.00
```

```
psi.2(z,c=1) # Same
```

```
## [1]  0.25  9.00  0.81 17.00
```

```
psi.2(x=z,c=1) # Same
```

```
## [1]  0.25  9.00  0.81 17.00
```

```
psi.2(c=1,x=z) # Same
```

```
## [1]  0.25  9.00  0.81 17.00
```

```
psi.2(1,z) # Different!
```

```
## [1] -1.25  1.00  0.99  1.00
```

Checking arguments

Odd behavior can occur when arguments are passed that we don't expect

```
psi.2(x=z,c=c(1,1,1,10))
```

```
## [1]  0.25  9.00  0.81 81.00
```

```
psi.2(x=z,c=-1)
```

```
## [1]   0.25 -11.00   0.81 -19.00
```

So we can put few sanity checks into the code

```
# "Huber" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x), scale for crossover (c)
# Outputs: vector with x^2 for small entries, 2c|x|-c^2 for large ones
psi.3 = function(x, c=1) {
  # Scale should be a single positive number
  stopifnot(length(c)==1, c>0)
  psi = ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(psi)
}
```

Arguments to `stopifnot()` are a series of expressions which should all be TRUE; execution halts, with error message, at first FALSE

What the function can see and do

- Each function has its own environment
- Names here over-ride names in the global environment
- Internal environment starts with the named arguments
- Assignments inside the function only change the internal environment
  (There are ways around this, but they are difficult and probably best avoided)
- Names undefined in the function are looked for in the environment the function gets called from

Environment examples

```
x = 7
y = c("A","C","G","T","U")
adder = function(y) { x = x+y; return(x) }
adder(1)
```

```
## [1] 8
```

```
x
```

```
## [1] 7
```

```
y
```

```
## [1] "A" "C" "G" "T" "U"
```

```
circle.area = function(r) { return(pi*r^2) }
circle.area(c(1,2,3))
```

```
## [1]  3.141593 12.566371 28.274334
```

```
truepi = pi
pi = 3 # Valid in 1800s Indiana
circle.area(c(1,2,3))
```

```
## [1]  3 12 27
```

```
pi = truepi # Restore sanity
circle.area(c(1,2,3))
```

```
## [1]  3.141593 12.566371 28.274334
```

Respect the interfaces!

Interfaces mark out a controlled inner environment for our code

Interact with the rest of the system only at the interface

Advice: arguments explicitly give the function all the information
- Reduces risk of confusion and error
- Exception: true universals like π

Likewise, output should only be through the return value

More about breaking up tasks and about environments later

Example: fitting a statistical model

Fact: bigger cities tend to produce more economically per capita

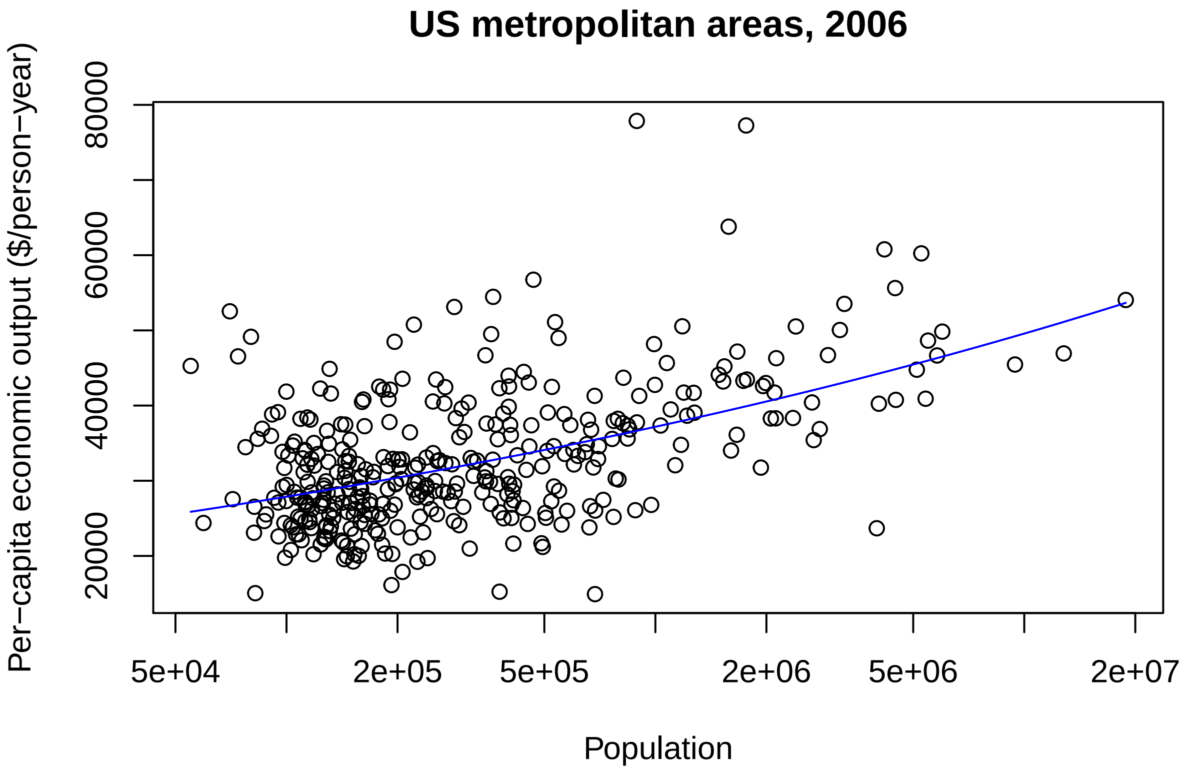
A proposed statistical model (Geoffrey West and others):

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

where Y is the per-capita "gross metropolitan product" of a city, N is its population, and y_0 and a are parameters

Some empirical evidence

```r
gmp = read.table("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/gmp.dat")
gmp$pop = gmp$gmp/gmp$pcgmp
plot(gmp$pop, gmp$pcgmp, log="x", xlab="Population",
  ylab="Per-capita economic output ($/person-year)",
  main="US metropolitan areas, 2006")
curve(6611*x^(1/8),add=TRUE,col="blue")
```



We want to fit the model

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

to some data. Take $y_0 = 6611$ for today

Unfortunately there's not an easy way to do this with a single mathematical formula. But we can do this *iteratively*. Let's approximate the derivative of error with respect to a, and move in the opposite direction

An actual first attempt at code:

```
maximum.iterations = 100
deriv.step = 1/1000
step.scale = 1e-12
stopping.deriv = 1/100
iteration = 0
deriv = Inf
a = 0.15
while ((iteration < maximum.iterations) &&
       (deriv > stopping.deriv)) {
  iteration = iteration + 1
  mse.1 = mean((gmp$pcgmp - 6611*gmp$pop^a)^2)
  mse.2 = mean((gmp$pcgmp - 6611*gmp$pop^(a+deriv.step))^2)
  deriv = (mse.2 - mse.1)/deriv.step
  a = a - step.scale*deriv
}
list(a=a,iterations=iteration,
     converged=(iteration<maximum.iterations))
```

```
## $a
## [1] 0.1258166
##
## $iterations
## [1] 58
##
## $converged
## [1] TRUE
```

What's wrong with this?

- Not encapsulated: re-run by cutting and pasting code—but how much of it? Also, hard to make part of something larger
- Inflexible: to change initial guess at a, have to edit, cut, paste, and re-run
- Error-prone: to change the data set, have to edit, cut, paste, re-run, and hope that all the edits are consistent
- Hard to fix: should stop when *absolute value* of derivative is small, but this stops when large and negative. Imagine having five copies of this and needing to fix same bug on each.

Let's turn this into a function and then improve it

Second attempt

Second attempt, with logic fix:

```
estimate.scaling.exponent.1 = function(a) {
  maximum.iterations = 100
  deriv.step = 1/1000
  step.scale = 1e-12
  stopping.deriv = 1/100
  iteration = 0
  deriv = Inf
  while ((iteration < maximum.iterations) &&
         (abs(deriv) > stopping.deriv)) {
    iteration = iteration + 1
    mse.1 = mean((gmp$pcgmp - 6611*gmp$pop^a)^2)
    mse.2 = mean((gmp$pcgmp - 6611*gmp$pop^(a+deriv.step))^2)
    deriv = (mse.2 - mse.1)/deriv.step
    a = a - step.scale*deriv
  }
  fit = list(a=a,y0=y0,iterations=iteration,
    converged=(iteration<maximum.iterations))
  return(fit)
}
```

Third attempt

All those magic numbers are bad! Let's make them defaults

```
estimate.scaling.exponent.2 = function(a, y0=6611,
  maximum.iterations=100, deriv.step=0.001,
  step.scale=1e-12, stopping.deriv=0.01) {

  iteration = 0
  deriv = Inf
  while ((iteration < maximum.iterations) &&
        (abs(deriv) > stopping.deriv)) {
    iteration = iteration + 1
    mse.1 = mean((gmp$pcgmp - y0*gmp$pop^a)^2)
    mse.2 = mean((gmp$pcgmp - y0*gmp$pop^(a+deriv.step))^2)
    deriv = (mse.2 - mse.1)/deriv.step
    a = a - step.scale*deriv
  }
  fit = list(a=a,y0=y0,iterations=iteration,
    converged=(iteration<maximum.iterations))
  return(fit)
}
```

Fourth attempt

Why type out the same calculation of the MSE twice? Let's create a function for this purpose

```
mse = function(a, y0, Y, N) { mean((Y-y0*N^a)^2) }

estimate.scaling.exponent.3 = function(a, y0=6611,
  maximum.iterations=100, deriv.step=0.001,
  step.scale=1e-12, stopping.deriv=0.01) {

  iteration = 0
  deriv = Inf

  while ((iteration < maximum.iterations) &&
        (abs(deriv) > stopping.deriv)) {
    iteration = iteration + 1
    deriv = (mse(a+deriv.step,y0,gmp$pcgmp,gmp$pop) -
              mse(a,y0,gmp$pcgmp,gmp$pop)) / deriv.step
    a = a - step.scale*deriv
  }
  fit = list(a=a,y0=y0,iterations=iteration,
    converged=(iteration<maximum.iterations))
  return(fit)
}
```

Fifth attempt

We're locked in to using specific columns of `gmp`; we shouldn't have to re-write code just to compare two data sets. Let's make more arguments, with defaults

```r
estimate.scaling.exponent.4 = function(a, y0=6611,
  Y=gmp$pcgmp, N=gmp$pop,
  maximum.iterations=100, deriv.step=0.001,
  step.scale=1e-12, stopping.deriv=0.01) {

  iteration = 0
  deriv = Inf

  while ((iteration < maximum.iterations) &&
         (abs(deriv) > stopping.deriv)) {
    iteration = iteration + 1
    deriv = (mse(a+deriv.step,y0,Y,N) -
               mse(a,y0,Y,N)) / deriv.step
    a = a - step.scale*deriv
  }
  fit = list(a=a,y0=y0,iterations=iteration,
    converged=(iteration<maximum.iterations))
  return(fit)
}
```

What have we done?

The final code is shorter, clearer, more flexible, and more re-usable

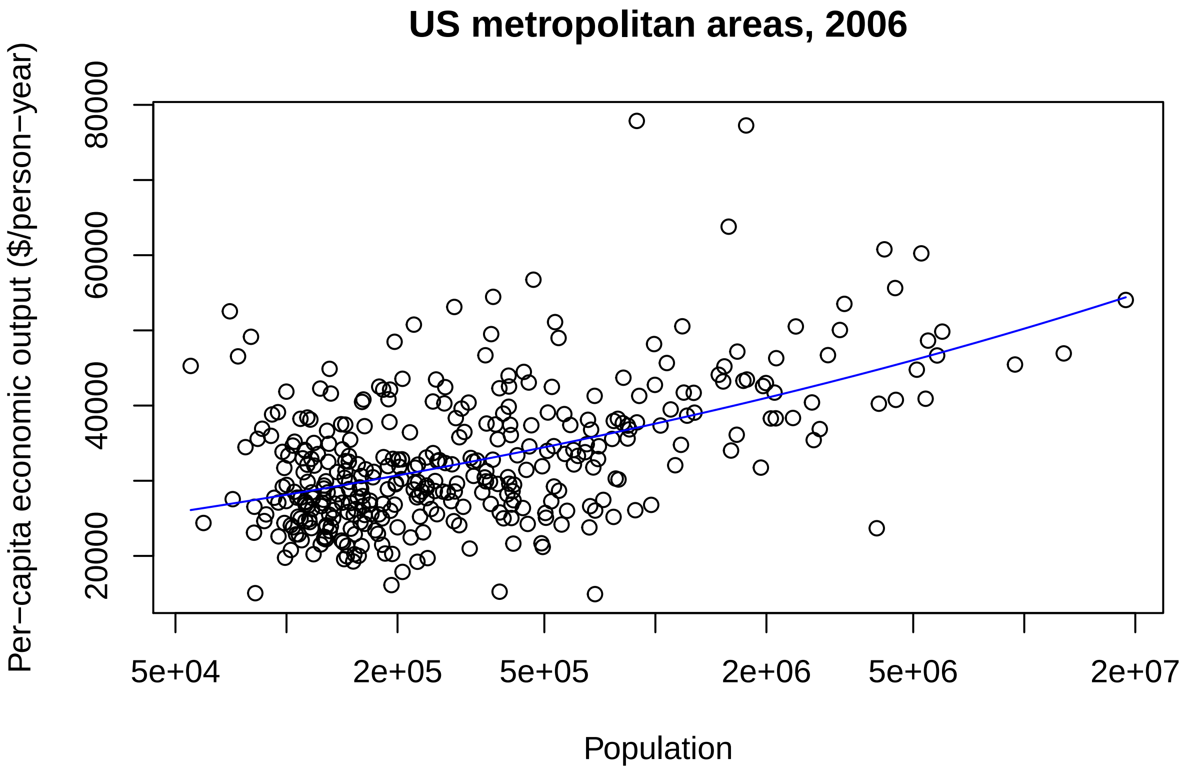
Exercises: - Run the code with the default values to get an estimate of a; plot the curve along with the data points - Randomly remove one data point—how much does the estimate change? - Run the code from multiple starting points—how different are the estimates of a?

Aren't you just a bit curious?

```r
plm = estimate.scaling.exponent.4(0.1)
plm
```

```
## $a
## [1] 0.1258166
##
## $y0
## [1] 6611
##
## $iterations
## [1] 62
##
## $converged
## [1] TRUE
```

```r
plot(gmp$pop, gmp$pcgmp, log="x", xlab="Population",
  ylab="Per-capita economic output ($/person-year)",
  main="US metropolitan areas, 2006")
curve(6611*x^plm$a,add=TRUE,col="blue")
```



We already wrote code plot this above … we've just copied and pasted it. What to do, if we were writing a report and needed to make many such plots (on say, different data sets)?

Yes, that's right. Write a function to make these kind of plots!

Plotting a fitted model

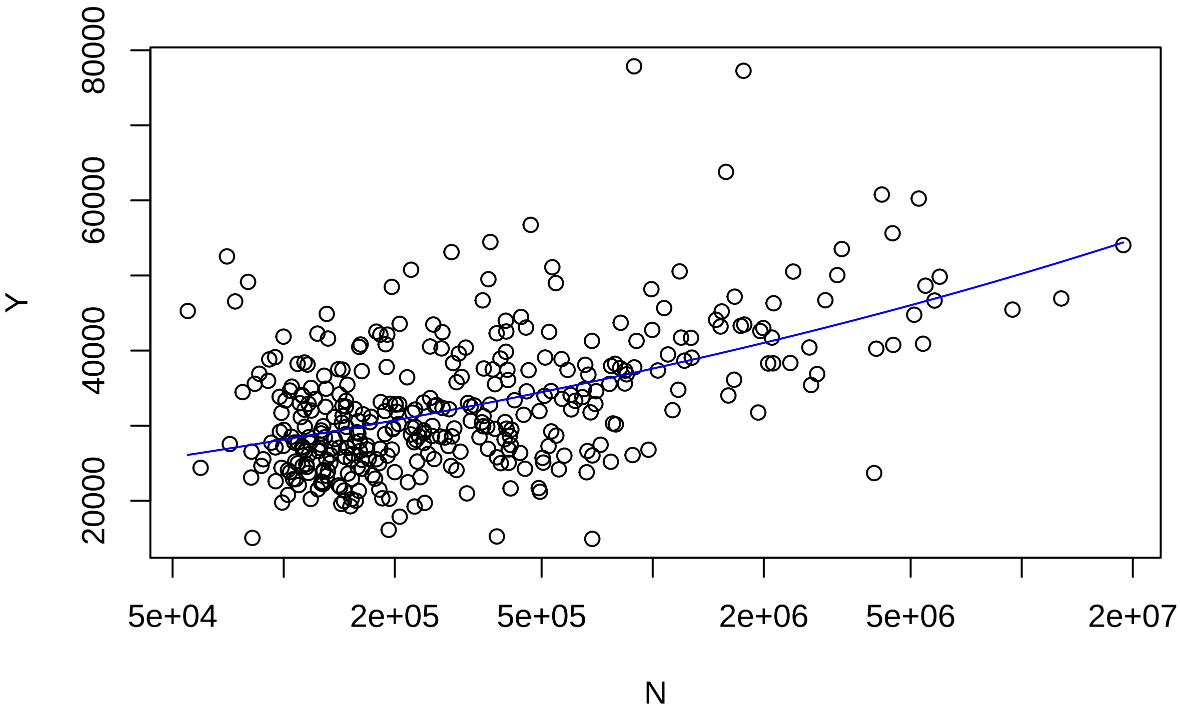
```
plot.plm = function(plm, curve.col="blue", log="x",
                    Y=gmp$pcgmp, N=gmp$pop, ...) {

  # Extract the parameters
  a = plm$a
  y0 = plm$y0
  # Plot the data
  plot(N,Y,log=log,...)
  # Draw the curve
  f = function(x) { return(y0*x^a) }
  curve(f(x),add=TRUE,col=curve.col)
  invisible(TRUE)
}
```

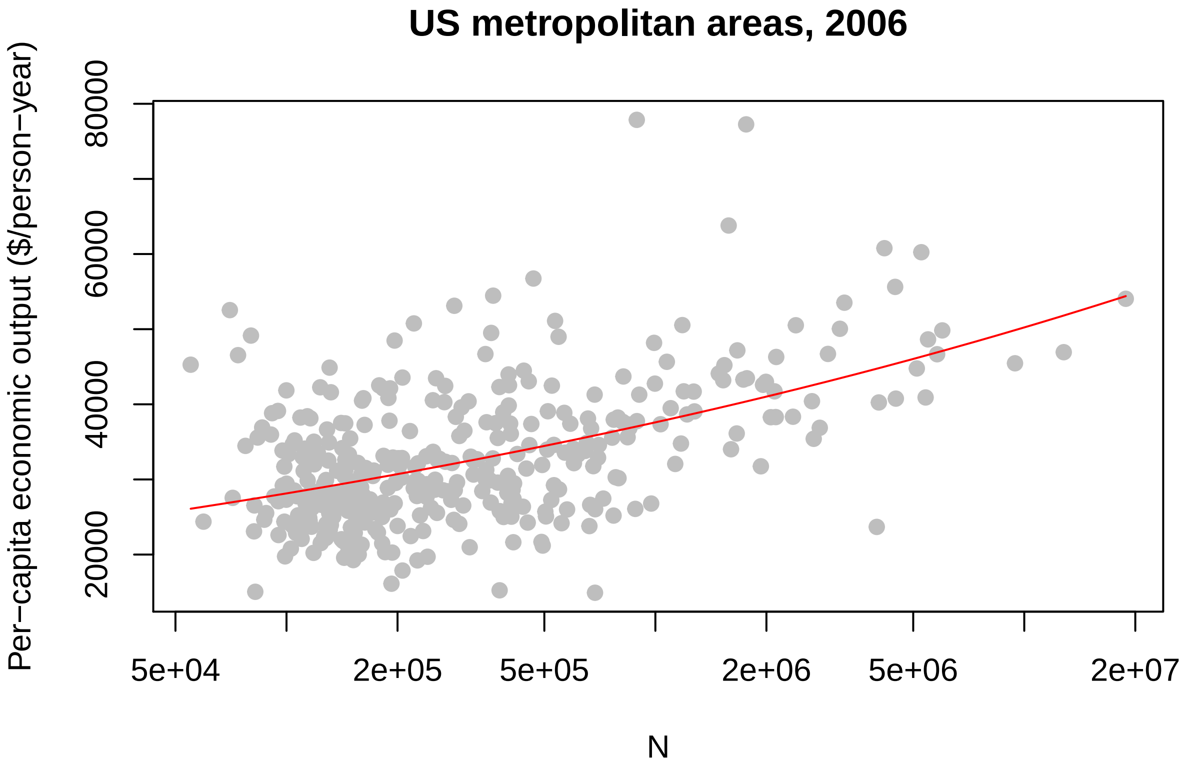
The `...` is a catch-all for any arguments the user wants to pass to the `plot()` function (e.g., `xlab` and 'ylab)

The function silently returns a TRUE (hence the `invisible()`, instead of a `return()`)

```
plot.plm(plm)
```



```
plot.plm(plm,curve.col="red",
         ylab="Per-capita economic output ($/person-year)",
         main="US metropolitan areas, 2006",
         pch=19,col="gray")
```



Summary

- Functions bundle related commands together into objects: easier to re-run, easier to re-use, easier to combine, easier to modify, less risk of error, easier to think about
- Interfaces control what the function can see (arguments, environment) and change (its internals, its return value)
- Calling functions we define works just like calling built-in functions: named arguments, defaults