Introduction to Functions

Rebecca C. Steorts, Duke University

STA 325, Supplemental Material

Agenda

- Defining functions: Tying related commands into bundles

Why Functions?

Data structures tie related values into one object

Functions tie related commands into one object

Both data structurs and functions are easier to work with.

Defining a function

```
function.name <- function(arguments){
  # computations on the arguments
  # some other code
}
```

What should be a function?

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

Trivial Example

Suppose we'd like to write a function to take the square of a number (or vector of numbers).

Squaring a number

```r
# Input: a number (scalar, vector, matrix)
# Output: the number squared
squared <- function(number) {
     return(number^2)
}
```

Squaring a number

Let's test our function

```r
squared(8)
```

```
## [1] 64
```

```r
squared(c(1,2,3,4))
```

```
## [1]  1  4  9 16
```

```r
squared(matrix(c(1,2,3,4),nrow=2))
```

```
##      [,1] [,2]
## [1,]    1    9
## [2,]    4   16
```

Example: Fitting a Model

Fact: bigger cities tend to produce more economically per capita

A proposed statistical model (Geoffrey West et al.):

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

where we have the following notation:

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

Example: Fitting a Model

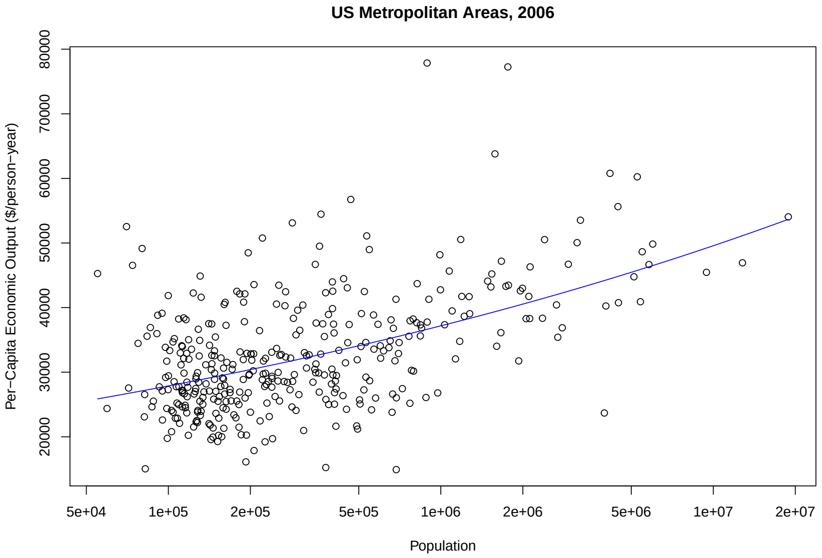
```r
gmp <- read.table("gmp.dat")
head(gmp)
```

```
##                            MSA         gmp pcgmp
## 1                  Abilene, TX 3.8870e+09 24490
## 2                    Akron, OH 2.2998e+10 32889
## 3                   Albany, GA 3.9550e+09 24269
## 4 Albany-Schenectady-Troy, NY 3.1321e+10 36836
## 5              Albuquerque, NM 3.0727e+10 37657
## 6               Alexandria, LA 3.8790e+09 25494
```

```r
(gmp$pop <- gmp$gmp/gmp$pcgmp)
```

```
##    [1]    158717.84    699261.15    162965.10    850282.33
##    [6]    152153.45    794430.63    125518.36    239816.08
##   [11]    358813.79    130999.87    176947.56    346177.74
##   [16]    216132.40    396976.90    182713.02   5113957.70
##   [21]    127938.40    522080.40   1527046.73    770226.28
```

Example: Fitting a Model



Example: Fitting a Model

Want to fit the model

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

Take $y_0 = 6611$ for today

Suppose we want to fit the above model, calculated it's mean squared error, and then we will stop fitting the model when the derivative the MSE "stop changing" by some small amount. Our goal will be to write this into a function.

Mean squared error (MSE) and it's derivative

Approximate the derivative of error w.r.t *a* using the following:

$$\begin{aligned} MSE(a) &\equiv \frac{1}{n}\sum_{i=1}^{n}(Y_i - y_0 N_i^a)^2 \\ MSE'(a) &\approx \frac{MSE(a+h) - MSE(a)}{h} \\ a_{t+1} - a_t &\propto -MSE'(a) \end{aligned}$$

The Quick Approach (No Function)

```r
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 try to gain improvements!

First Attempt

```r
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,iterations=iteration, converged=(iteration < maximum.iterations))
  return(fit)
}
estimate.scaling.exponent.1(a=0.15)
```

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

But why do we have many fixed constants running around?

Fixing the fixed constants

```r
estimate.scaling.exponent.2 <- function(a, y0=6611,
  maximum.iterations=100, deriv.step = .001,
  step.scale = 1e-12, stopping.deriv = .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,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
estimate.scaling.exponent.2(a=0.15)
```

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

Why type out the same calculation of the MSE twice? Instead, let's create a function for this.

Creating an MSE function

```r
estimate.scaling.exponent.3 <- function(a, y0=6611,
  maximum.iterations=100, deriv.step = .001,
  step.scale = 1e-12, stopping.deriv = .01) {
  iteration <- 0
  deriv <- Inf
  mse <- function(a) { mean((gmp$pcgmp - y0*gmp$pop^a)^2) }
  while ((iteration < maximum.iterations) && (abs(deriv) > stopping.deriv)) {
    iteration <- iteration + 1
    deriv <- (mse(a+deriv.step) - mse(a))/deriv.step
    a <- a - step.scale*deriv
  }
  fit <- list(a=a,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
estimate.scaling.exponent.3(a=0.15)
```

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

We're locked in to using specific columns of gmp; shouldn't have to re-write just to compare two data sets. Let's add more arguments for the response and preditor variable

More arguments (with defaults)

```r
estimate.scaling.exponent.4 <- function(a, y0=6611,
  response=gmp$pcgmp, predictor = gmp$pop,
  maximum.iterations=100, deriv.step = .001,
  step.scale = 1e-12, stopping.deriv = .01) {
  iteration <- 0
  deriv <- Inf
  mse <- function(a) { mean((response - y0*predictor^a)^2) }
  while ((iteration < maximum.iterations) && (abs(deriv) > stopping.deriv)) {
    iteration <- iteration + 1
    deriv <- (mse(a+deriv.step) - mse(a))/deriv.step
    a <- a - step.scale*deriv
  }
  fit <- list(a=a,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
estimate.scaling.exponent.4(a=0.15)
```

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

We could turn the `while()` loop into a `for()` loop, and nothing outside the function would care

Replacing `while()` loop with `for()` loop

```r
estimate.scaling.exponent.5 <- function(a, y0=6611,
  response=gmp$pcgmp, predictor = gmp$pop,
  maximum.iterations=100, deriv.step = .001,
  step.scale = 1e-12, stopping.deriv = .01) {
  mse <- function(a) { mean((response - y0*predictor^a)^2) }
  for (iteration in 1:maximum.iterations) {
    deriv <- (mse(a+deriv.step) - mse(a))/deriv.step
    a <- a - step.scale*deriv
    if (abs(deriv) <= stopping.deriv) { break() }
  }
  fit <- list(a=a,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
estimate.scaling.exponent.5(a=0.15)
```

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

Final Code

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

Exercise: Run the code with the default values to get an estimate of a; plot the curve along with the data points

Exercise: Randomly remove one data point — how much does the estimate change?

Exercise: Run the code from multiple starting points — how different are the estimates of a?

Summary

- **Functions** bundle related commands together into objects.
- Using functions make code easier to re-run, easier to re-use, easier to combine, easier to modify, less risk of error, easier to conceptualize
- We can write nice, clean programs by writing code that includes many functions.