

# Statistical Computing (36-350)

## Lecture 4: Writing and Calling Functions

Cosma Shalizi

9 September 2013

# Agenda

- Defining functions: Tying related commands into bundles
- Interfaces: Controlling what the function can see and do
- Example: Parameter estimation code

READING FOR FRIDAY: 1.3, 7.3–7.5, 7.11, 7.13 of Matloff  
(skipping “extended examples”)

# Why Functions?

Data structures tie related values into one object

Functions tie related commands into one object

For example:

```
# "Robust" 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:

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

**Interfaces:** the **inputs** or **arguments**; the **outputs** or **return value**

Calls other functions `ifelse()`, `abs()`, and operators `^` and `>`

could also call other functions we've written

`return()` says what the output is

alternately, return the last evaluation; I like explicit returns better

**Comments:** Not required by R, but a Very Good Idea

One-line description of purpose; listing of arguments; listing of outputs

## Named and default arguments

```
# "Robust" 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)
}

> identical(psi.1(z), psi.2(z,c=1))
[1] TRUE
```

Default values get used if names are missing:

```
> identical(psi.2(z,c=1), psi.2(z))
[1] TRUE
```

Named arguments can go in any order when explicitly tagged:

```
> identical(psi.2(x=z,c=2), psi.2(c=2,x=z))
[1] TRUE
```

## Checking Arguments

PROBLEM: Odd behavior when arguments aren't as we 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
```

SOLUTION: Put little sanity checks into the code

```
# "Robust" 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 evaluate to TRUE; execution halts, with error message, at *first* FALSE (try it!)

## What the function can see and do

Argument names over-ride those in the larger environment, *inside* the function

Changes made inside the function don't propagate

```
> 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"
```

There *are* ways around this, but they are difficult and best avoided  
(see Chambers, ch. 5)

## Looking in the Environment

Names undefined in the function are looked for in the environment  
R uses their value when the function is called, not when it's defined

```
> 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          # Only valid in 19th century Indiana, or drowned R'lyeh
> 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  
Interaction with the rest of the system is only as allowed at the interface

Good practice: explicitly give the function all the information it needs through the arguments; this minimizes the chances of confusion and error

Exception: true universals like  $\pi$

Likewise, output should only be through the return value

Will say more about breaking up tasks and about environments later

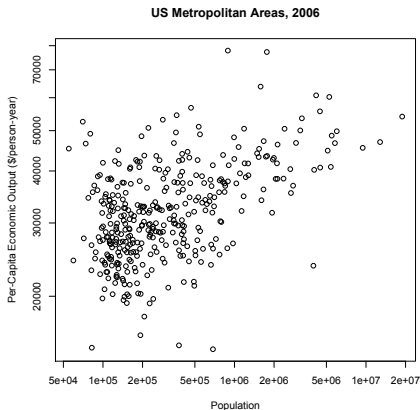
Further reading: Herbert Simon, *The Sciences of the Artificial*

## Example: Fitting a Model

Fact: bigger cities tend to produce more economically per capita  
A proposed statistical model (Geoffrey West et al.):

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

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



Log scales on both axes

```
gmp <- read.table("gmp.dat"); gmp$pop <- gmp$gmp/gmp$pcgmp  
plot(pcgmp~pop, data=gmp, xlab="Population", log="xy",  
     ylab="Per-Capita Economic Output ($/person-year)",  
     main="US Metropolitan Areas, 2006")
```

Want to fit the model

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

Take  $y_0 = 6611$  for today

Approximate the derivative of error w.r.t  $a$  and move against it

$$MSE(a) \equiv \frac{1}{n} \sum_{i=1}^n (Y_i - y_0 N_i^a)^2$$

$$MSE'(a) \approx \frac{MSE(a+h) - MSE(a)}{h}$$

$$a_{t+1} - a_t \propto -MSE'(a)$$

## 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))
```

# What's wrong with this?

## 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

## 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



## 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

## 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.

## 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.

Will turn this into a function and then improve it; comments omitted here, see online

## First 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,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
```

PROBLEM: All those magic numbers!

PROBLEM: All those magic numbers!

SOLUTION: Make them defaults

```
estimate.scaling.exponent.2 <- function(a, y0=6611, maximum.iterations=100,  
  deriv.step = 1/100, 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$pcgmp - y0*gmp$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)  
}
```

EXERCISE: Experiment with different values of deriv.step

PROBLEM: Why type out the same calculation of the MSE twice?

PROBLEM: Why type out the same calculation of the MSE twice?

SOLUTION: Declare a function

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

`mse()` declared inside `estimate.scaling.exponent.3()`, so it can see `y0`, but it's not added to the global environment



PROBLEM: Locked in to using specific columns of `gmp`; shouldn't have to re-write just to compare two data sets

PROBLEM: Locked in to using specific columns of `gmp`; shouldn't have to re-write just to compare two data sets

SOLUTION: More arguments, with defaults

```
estimate.scaling.exponent.4 <- function(a, y0=6611, response=gmp$pcgmp,
  predictor = gmp$pop, maximum.iterations=100, deriv.step = 1/100,
  step.scale = 1e-12, stopping.deriv = 1/100) {
  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)
}
```

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

```
estimate.scaling.exponent.5 <- function(a, y0=6611, response=gmp$pcgmp,
  predictor = gmp$pop, maximum.iterations=100, deriv.step = 1/100,
  step.scale = 1e-12, stopping.deriv = 1/100) {
  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)
}
```

## What have we done?

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

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

Next time: working with many functions