Statistical Computing (36-350)

Undelivered Lecture: More Design, and Scoping

Cosma Shalizi

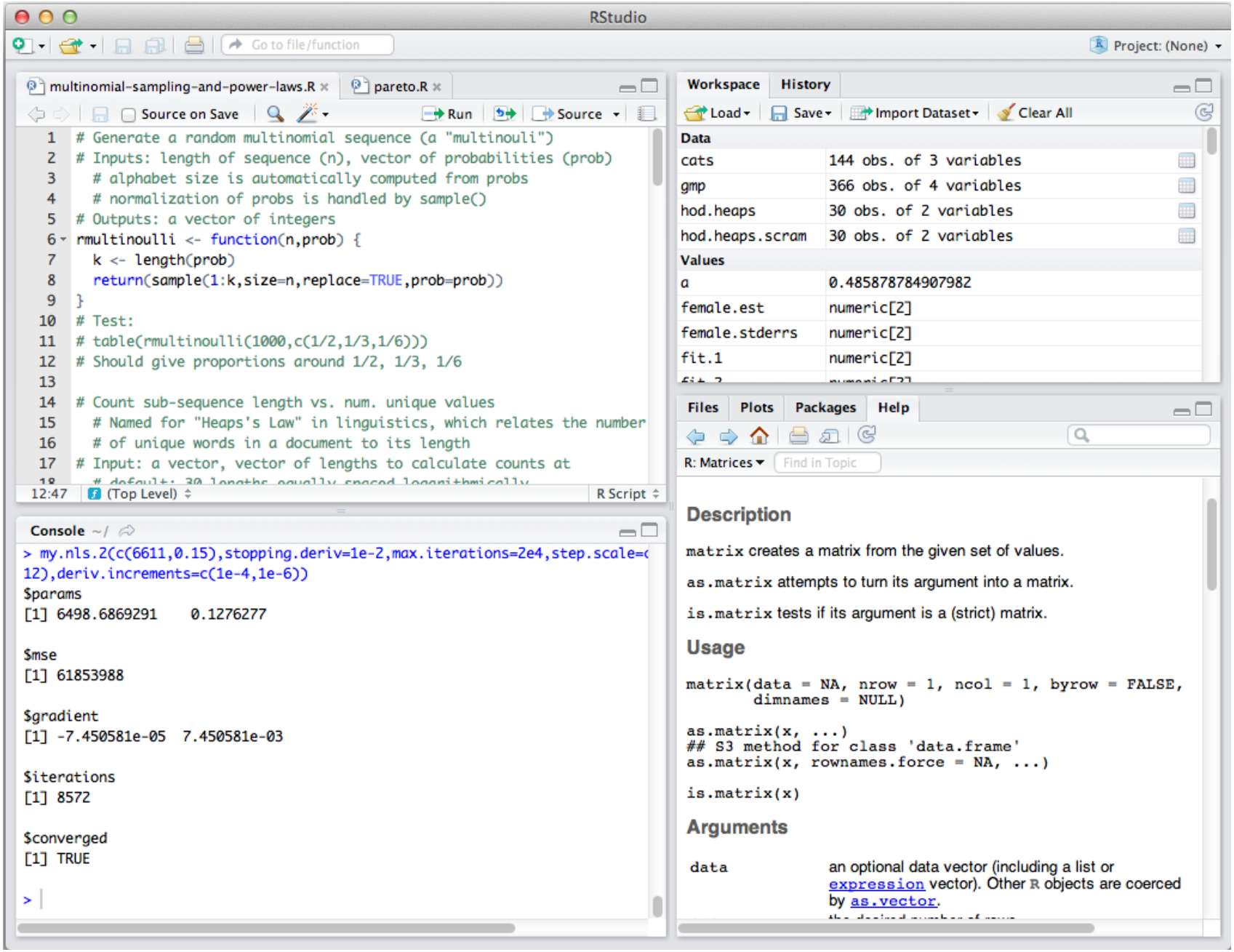
Not delivered 2013

Agenda

- The scope of names: what they mean where
- Example: Not passing lots of arguments

Looking Up Names

When R sees a variable name, it needs to look up what value goes with that name
It consults the **environment**, a list of name/value pairs
If the name isn't in the current environment, it looks in the larger, **parent** environment, and so on to the global environment
The global environment is what we interact with at the terminal


RStudio
Go to file/function
Project: (None)
multinomial-sampling-and-power-laws.R
pareto.R
Source on Save
Run
Source
1 # Generate a random multinomial sequence (a "multinouli")
2 # Inputs: length of sequence (n), vector of probabilities (prob)
3 # alphabet size is automatically computed from probs
4 # normalization of probs is handled by sample()
5 # Outputs: a vector of integers
6 rmultinoulli <- function(n,prob) {
7 k <- length(prob)
8 return(sample(1:k,size=n,replace=TRUE,prob=prob))
9 }
10 # Test:
11 # table(rmultinoulli(1000,c(1/2,1/3,1/6)))
12 # Should give proportions around 1/2, 1/3, 1/6
13
14 # Count sub-sequence length vs. num. unique values
15 # Named for "Heaps's Law" in linguistics, which relates the number
16 # of unique words in a document to its length
17 # Input: a vector, vector of lengths to calculate counts at
12:47
(Top Level)
R Script
Console ~/
> my.nls.2(c(6611,0.15),stopping.deriv=1e-2,max.iterations=2e4,step.scale=c(
12),deriv.increments=c(1e-4,1e-6))
$params
[1] 6498.6869291 0.1276277
$mse
[1] 61853988
$gradient
[1] -7.450581e-05 7.450581e-03
$iterations
[1] 8572
$converged
[1] TRUE
>
Workspace
History
Load
Save
Import Dataset
Clear All
Data
cats 144 obs. of 3 variables
gmp 366 obs. of 4 variables
hod.heaps 30 obs. of 2 variables
hod.heaps.scram 30 obs. of 2 variables
Values
a 0.485878784907982
female.est numeric[2]
female.stderrs numeric[2]
fit.1 numeric[2]
Files
Plots
Packages
Help
R: Matrices
Find in Topic
Description
matrix creates a matrix from the given set of values.
as.matrix attempts to turn its argument into a matrix.
is.matrix tests if its argument is a (strict) matrix.
Usage
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
dimnames = NULL)
as.matrix(x, ...)
S3 method for class 'data.frame'
as.matrix(x, rownames.force = NA, ...)
is.matrix(x)
Arguments
data
an optional data vector (including a list or expression vector). Other R objects are coerced by as.vector.

name	value
...	...
x	c(1,2,3,4)
y	3.7
cats	*a data frame with three columns*

psi	`function(x,c=1) {`
	`  ifelse(abs(x)>c,2*c*abs(x)-c^ 2,x^ 2)}`
	`}`

parent environment	*a pointer telling R where to look in its memory*

The Scope of Names

Because R "goes up the chain", if this environment and its parent share a name, R uses the local name — that's the **scope** of the assignment

Assignment with <- or = only affects the *current* environment

Changes in this environment do not affect its parents

Changes in the parents affect this one, *unless* over-ridden locally

Functions and Environments

Inside a function definition, we have a new, internal environment
The new environment starts with just the named arguments of the function
Names in this environment *locally* over-ride those outside
Changing/creating variables does not affect other environments
The parent of the function's internal environment is the one it was defined in, not the one it was called in

Examples:

```
> f <- function(x) {
+   f <- x^2*exp(-x^2)
+   return(f)
+ } # Assigns this function the name "f"
> f  # What value goes with the name "f"?
function(x) {
  f <- x^2*exp(-x^2)
  return(f)
}
> x <- 3  # Assigns x the value 3, globally
> f(7)    # Assigns x the value 7, INSIDE f
[1] 2.569014e-20
> f(x)    # Did not change x globally
[1] 0.001110688
> f       # Also did not change the global value of "f"
function(x) {
  f <- x^2*exp(-x^2)
  return(f)
}
```

More examples:

```
g <- function(x) {
  gamma <- 2*x*exp(-x^2)
  kappa <- -2*x^3*exp(-x^2)
  eta <- gamma+kappa
  return(eta)
}

h <- function(y) {
  return(eta*sin(y))
}
```

More examples:

```
g <- function(x) {
  gamma <- 2*x*exp(-x^2)
  kappa <- -2*x^3*exp(-x^2)
  eta <- gamma+kappa
  return(eta)
}

h <- function(y) {
  return(eta*sin(y))
}
```

Q: what happens if we run

```
g(3)
h(pi)
```

More examples:

```
g <- function(x) {
  gamma <- 2*x*exp(-x^2)
  kappa <- -2*x^3*exp(-x^2)
  eta <- gamma+kappa
  return(eta)
}

h <- function(y) {
  return(eta*sin(y))
}
```

Q: what happens if we run

```
g(3)
h(pi)
```

A: Depends on what `eta` was in the parent environment, before we ran these!

Environment of definition vs. execution

```
> wheel <- function(r) {2*pi*r}
> wheel.inside.wheel <- function(r,pi) { return(wheel(r)) }
> wheel(1)                    # Acts naturally
[1] 6.283185
> wheel.inside.wheel(1,3)     # Will not be 2*3*1
[1] 6.283185
```

vs.

```
> wheel.inside.wheel <- function(r,pi) {
+   wheel <- function(r) { 2*pi*r }
+   return(wheel(r))
+ }
> wheel.inside.wheel(1,pi)    # Acts naturally
[1] 6.283185
> wheel.inside.wheel(1,3)     # Will be 2*3*1
[1] 6
```

Why Does R Do This To Us?

No interference between the insides of separate functions
∴ no restrictions on naming arguments, or on using other people's code, whatever their internal names
Looking to larger environments is a convenience: share information by nesting functions, and allow global constants
Making the parent the environment of definition lets us preserve functions easily (e.g., statistical models) — will see more when we look at functions as objects

Design Implications

Compartmentalize information
Sometimes encourages nested functions

Example: The curve-fitting homework

First sketch of parts:

```
my.nls <- function(first_guess_at_parameters, data, controls) {
  until we run out of time
    find the gradient at the current parameter guess
    if the gradient is small, stop,
    otherwise move the parameters against the direction of the gradient
  gather up return values
}
```

(not really code!)

Translate into code:

```
my.nls <- function(params, N=gmp$pop, Y=gmp$pcgmp, stopping.deriv,
  max.iterations, step.scale,deriv.increments) {
  for (iteration in 1:max.iterations) {
    gradient <- mse.grad(params,deriv.increments)
    if(all(abs(gradient)) < stopping.deriv) { break() }
    params <- params - step.scale*gradient
  }
  fit <- list(params=params,gradient=gradient,iterations=iteration,
    converged=(iteration < max.iterations))
  return(fit)
}
```

needs an `mse.grad` function

Building `mse.grad`

Skipping preliminary analysis:

```
mse.grad <- function(params,deriv.increments) {
  p <- length(params)
  stopifnot(p==length(deriv.increments))
  # Matrix with 1 row per tweaked parameter value
  new.params <- matrix(rep(params,p),nrow=p,byrow=TRUE)+diag(deriv.increments)
  new.mse <- apply(new.params,1,mse)
  return((new.mse - mse(params))/deriv.increments)
}
```

Needs an `mse()` function

Finally, the `mse` function:

```
mse <- function(params,N=gmp$pop,Y=gmp$pcgmp) {
  predictions <- params[1]*N^params[2]
  mse <- mean((Y-predictions)^2)  # Why doesn't this clobber the function?
  return(mse)
}
```

Integration

Problem: how to get `mse.grad` to notice if the data changes?
Solution 1: change arguments to `mse.grad`, to include data, which it passes to `mse` (as in the solutions)
Solution 2: manipulate scope, remembering environment of definition is what matters
Solution 3: functions as arguments (a later lecture)
Let's look at solution 2

All together:

```
my.nls.2 <- function(params, N=gmp$pop, Y=gmp$pcgmp, stopping.deriv,
  max.iterations, step.scale,deriv.increments) {

  mse <- function(params) { return(mean((Y-params[1]*N^params[2])^2)) }
  mse.grad <- function(params) {
    p <- length(params)
    stopifnot(p==length(deriv.increments))
    new.params <- matrix(rep(params,p),nrow=p,byrow=TRUE)+diag(deriv.increments)
    new.mse <- apply(new.params,1,mse)
    return((new.mse - mse(params))/deriv.increments)
  }

  for (iteration in 1:max.iterations) {
    gradient <- mse.grad(params)
    if(all(abs(gradient) < stopping.deriv)) { break() }
    params <- params - step.scale*gradient
  }
  fit <- list(params=params,mse=mse(params),gradient=gradient,
    iterations=iteration,converged=(iteration < max.iterations))
  return(fit)
}
```

Summary

1. Environments control the values of names
2. Values and assignments in local environments over-rule more global ones
3. "Local" goes by definition, not execution
4. Use scoping to control information sharing between functions