Date Science with R and Python

Writing and Using Functions

Peng Zhang

School of Mathematical Sciences, Zhejiang University

2023/07/06

Loading [MathJax]/jax/output/CommonHTML/jax.js

Create an R script

Now, we are going to create an R script. What is an R script? It's a text file with a .R extension. We've been writing R code in R Markdown files so far; R scripts are just R code without the Markdown along with it.

Go to File > New File > R Script (or click the green plus in the top left corner).

Let's start off with a few comments so that we know what it is for, and save it:

```
## r-programming.R
## condition, iteration and functions
## Peng Zhang pengz@zju.edu.cn
```

Agenda

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

Control and iteration

```
library(datasets)
states <- data.frame(state.x77, abb=state.abb, region=state.region, 
```

Conditionals

Have the computer decide what to do next

- Mathematically:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}, \quad \psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

Exercise: plot ψ in R

- Computationally:

```
if the country code is not "US"
multiply prices by current exchange rate
```

if()

Simplest conditional:

```
if (x >= 0) {
  x
} else {
  -x
}
```

Condition in `if` needs to give *one* `TRUE` or `FALSE` value

`else` clause is optional

one-line actions don't need braces

```
if (x >= 0) x else -x
```

Nested if()

`if` can *nest* arbitrarily deeply:

```
if (x^2 < 1) {
  x^2
} else {
  if (x >= 0) {
    2*x-1
  } else {
     -2*x-1
  }
}
```

Can get ugly though

Combining Booleans: && and ||

& work | like + or *: combine terms element-wise

Flow control wants *one* Boolean value, and to skip calculating what's not needed

&& and || give *one* Boolean, lazily:

```
(0 > 0) && (all.equal(42%%6, 169%%13))
```

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

This *never* evaluates the complex expression on the right

Use && and || for control, & and | for subsetting

Iteration

Repeat similar actions multiple times:

```r
table.of.logarithms <- vector(length=7,mode="numeric")
table.of.logarithms
```

```
## [1] 0 0 0 0 0 0 0
```

```r
for (i in 1:length(table.of.logarithms)) {
  table.of.logarithms[i] <- log(i)
}
table.of.logarithms
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
```

for()

```
for (i in 1:length(table.of.logarithms)) {
  table.of.logarithms[i] <- log(i)
}
```

`for` increments a **counter** (here `i`) along a vector (here `1:length(table.of.logarithms)`) and **loops through** the **body* until it runs through the vector

"**iterates over** the vector"

N.B., there is a better way to do this job!

The body of the for() loop

Can contain just about anything, including:

- if() clauses
- other for() loops (nested iteration)

Nested iteration example

```
c <- matrix(0, nrow=nrow(a), ncol=ncol(b))
if (ncol(a) == nrow(b)) {
  for (i in 1:nrow(c)) {
    for (j in 1:ncol(c)) {
      for (k in 1:ncol(a)) {
        c[i,j] <- c[i,j] + a[i,k]*b[k,j]
      }
    }
  }
} else {
  stop("matrices a and b non-conformable")
}
```

while(): conditional iteration

```r
x <- 1:10
while (max(x) - 1 > 1e-06) {
  x <- sqrt(x)
}
```

Condition in the argument to `while` must be a single Boolean value (like `if`)

Body is looped over until the condition is `FALSE` so can loop forever

Loop never begins unless the condition starts `TRUE`

for() vs. while()

for() is better when the number of times to repeat (values to iterate over) is clear in advance

while() is better when you can recognize when to stop once you're there, even if you can't guess it to begin with

Every for() could be replaced with a while()
Exercise: show this

Avoiding iteration

R has many ways of *avoiding* iteration, by acting on whole objects

- It's conceptually clearer
- It leads to simpler code
- It's faster (sometimes a little, sometimes drastically)

Vectorized arithmetic

How many languages add 2 vectors:

```
c <- vector(length(a))
for (i in 1:length(a)) {  c[i] <- a[i] + b[i]  }
```

How R adds 2 vectors:

```
a+b
```

or a triple `for()` loop for matrix multiplication vs. `a %*% b`

Advantages of vectorizing

- Clarity: the syntax is about *what* we're doing
- Concision: we write less
- Abstraction: the syntax hides *how the computer does it*
- Generality: same syntax works for numbers, vectors, arrays, …
- Speed: modifying big vectors over and over is slow in R; work gets done by optimized low-level code

Vectorized calculations

Many functions are set up to vectorize automatically

```
abs(-3:3)
```

```
## [1] 3 2 1 0 1 2 3
```

```
log(1:7)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
```

See also `apply()` from last week

We'll come back to this in great detail later

Vectorized conditions: ifelse()

```
ifelse(x^2 > 1, 2*abs(x)-1, x^2)
```

1st argument is a Boolean vector, then pick from the 2nd or 3rd vector arguments as `TRUE` or `FALSE`

Summary

- Dataframes
- `if`, nested `if`, `switch`
- Iteration: `for`, `while`
- Avoiding iteration with whole-object ("vectorized") operations

What Is Truth?

0 counts as `FALSE`; other numeric values count as `TRUE`; the strings "TRUE" and "FALSE" count as you'd hope; most everything else gives an error

Advice: Don't play games here; try to make sure control expressions are getting Boolean values

Conversely, in arithmetic, `FALSE` is 0 and `TRUE` is 1

```r
mean(states$Murder > 7)
```

```
## [1] 0.48
```

switch()

Simplify nested `if` with `switch()`: give a variable to select on, then a value for each option

```
switch(type.of.summary,
       mean=mean(states$Murder),
       median=median(states$Murder),
       histogram=hist(states$Murder),
       "I don't understand")
```

Exercise (off-line)

Set `type.of.summary` to, succesively, "mean", "median", "histogram", and "mode", and explain what happens

Unconditional iteration

```
repeat {
  print("Help! I am Dr. Morris Culpepper, trapped in an endless loop!")
}
```

"Manual" control over iteration

```
repeat {
  if (watched) { next() }
  print("Help! I am Dr. Morris Culpepper, trapped in an endless loop!")
  if (rescued) { break() }
}
```

`break()` exits the loop; `next()` skips the rest of the body and goes back into the loop

both work with `for()` and `while()` as well

Exercise: how would you replace `while()` with `repeat()`?

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

For example

```r
# "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:

```r
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:

```r
# "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()`, 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

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

will say more about design later

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 one
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 on
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 (try it!)

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 best avoided; see Chambers, ch. 5, if you must
- Names undefined in the function are looked for in the environment the function gets called from
  not the environment of definition

Internal environment examples

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

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

```r
x
```

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

```r
y
```

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

Internal environment examples cont'd.

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

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

```r
truepi <- pi
pi <- 3   # Valid in 1800s Indiana, or drowned R'lyeh
circle.area(c(1,2,3))
```

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

```r
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

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.):

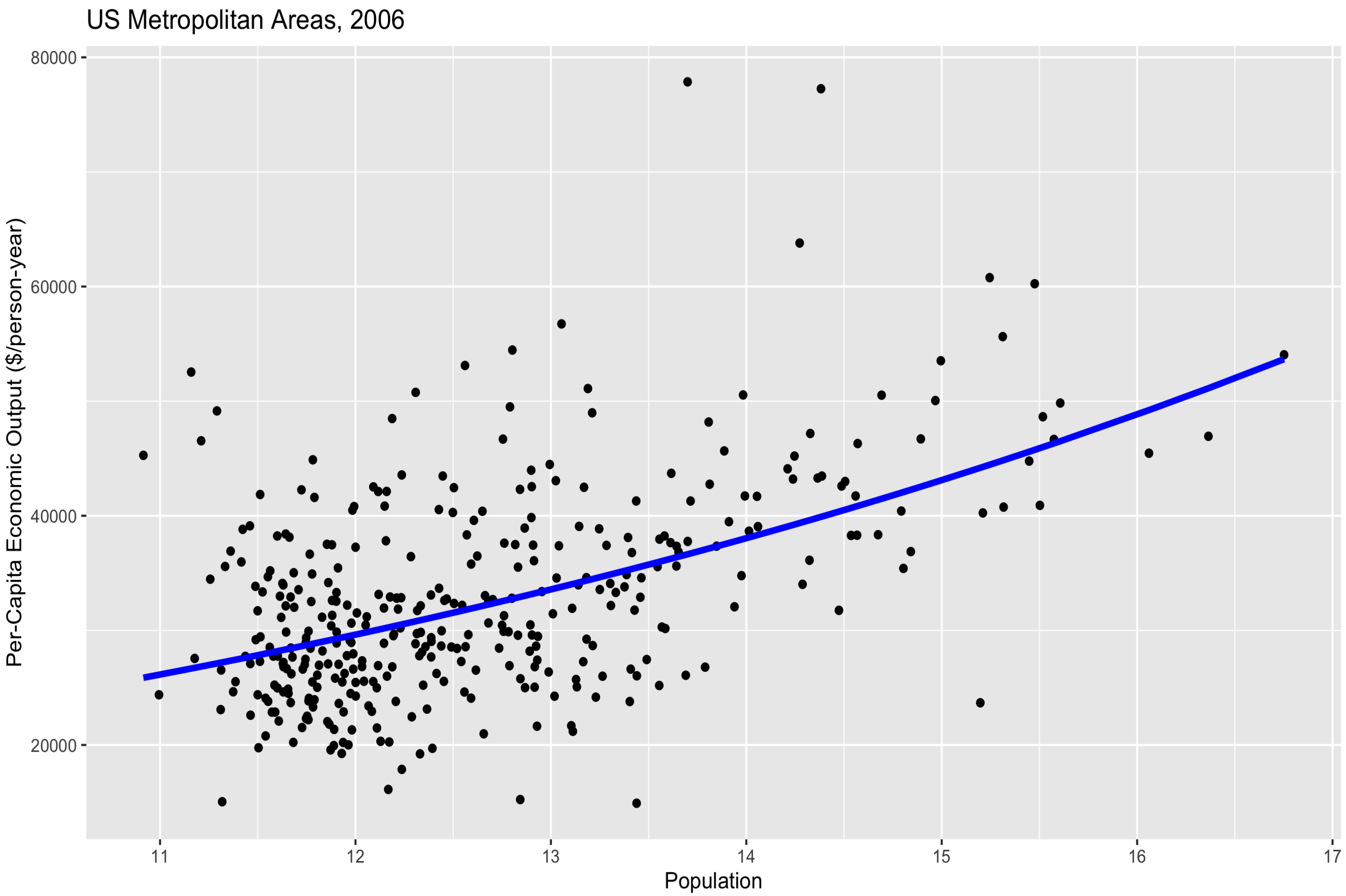
$$Y = y_0 N^a + \mathrm{noise}$$

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

Evidence

```
gmp <- read.table("data/gmp.dat")
gmp <- gmp %>% mutate(pop = gmp/pcgmp, nlmfit = 6611*pop^(1/8))
```

```
gmp %>% ggplot() + geom_point(aes(x = pop, y = pcgmp))+
  labs(x = "Population", y = "Per-Capita Economic Output ($/person-ye
  title = "US Metropolitan Areas, 2006")+
  scale_x_log10()+
  geom_line(aes(x = pop, y = nlmfit), col = 'blue', size = 1.5)
```


US Metropolitan Areas, 2006
Per-Capita Economic Output ($/person-year)
80000
60000
40000
20000
11
12
13
14
15
16
17
Population

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

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

An actual first attempt at code:

```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.iterati
```

```
## $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.

Will turn this into a function and then improve it

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.d
    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)
}
```

```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.d
    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)
}
```

```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.d
    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 the function, so it can see `y0`, but it's not added to the global environment

```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.d
    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

```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)
}
```

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

- **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

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

In our last episode

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

Why You Have to Write More Than One Function

Meta-problems:

- You've got more than one problem
- Your problem is too hard to solve in one step
- You keep solving the same problems

Meta-solutions:

- Write multiple functions, which rely on each other
- Split your problem, and write functions for the pieces
- Solve the recurring problems once, and re-use the solutions

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

Consistent Interfaces

- Functions for the same kind of object should use the same arguments, and presume the same structure
- Functions for the same kind of task should use the same arguments, and return the same sort of value

(to the extent possible)

Keep related things together

- Put all the related functions in a single file
- Source them together
- Use comments to note ***dependencies***

Power-Law Scaling for Urban Economies (cont'd.)

Remember the model:

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

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

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

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

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

```
# 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)
}
```

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)
}
```

Sub-Functions

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: tinker with sub-problems at leisure
- Easier to design: for future lecture
- 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

Rule of thumb: If you find yourself writing the same code in multiple places, make it a separate function

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)
}
```

Recursion

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))
  }
}
```

Exercise: Convince yourself that any loop can be replaced by recursion; can you always replace recursion with a loop?

Cleaner Resource Allocation

```
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(output * (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

Top-down design of programs

Abstraction

- The point of abstraction: program in ways which don't use people as bad computers
- Economics says: rely on *comparative* advantage
    - Computers: Good at tracking arbitrary details, applying rigid rules
    - People: Good at thinking, meaning, discovering patterns

- ∴ organize programming so that people spend their time on the big picture, and computers on the little things

- Abstraction --- hiding details and specifics, dealing in generalities and common patterns --- is a way to program so you do what you're good at, and the computer does what it's good at

- We have talked about lots of examples of this already
    - Names; data structures; functions; interfaces

Top-Down Design

- Start with the big-picture view of the problem
- Break the problem into a few big parts
- Figure out how to fit the parts together
- Go do this for each part

The Big-Picture View

- Resources: what information is available as part of the problem?
  - Usually arguments to a function
- Requirements: what information do we want as part of the solution?
  - Usually return values
- What do we have to do to transform the problem statement into a solution?

Breaking Into Parts

- Try to break the calculation into a *few* (say ≤ 5) parts
  - Bad: write 500 lines of code, chop it into five 100-line blocks
  - Good: each part is an independent calculation, using separate data
- Advantages of the good way:
  - More comprehensible to human beings
  - Easier to improve and extend (respect interfaces)
  - Easier to debug
  - Easier to test

Put the Parts Together

- *Assume* that you can solve each part, and their solutions are functions
- Write top-level code for the function which puts those steps together:

```
# Not actual code
big.job <- function(lots.of.arguments) {
   intermediate.result <- first.step(some.of.the.args)
   final.result <- second.step(intermediate.result,rest.of.the.args)
   return(final.result)
}
```

- The sub-functions don't have to be written when you *declare* the main function, just when you *run* it

What About the Sub-Functions?

- Recursion: Because each sub-function solves a single well-defined problem, we can solve it by top-down design
- The step above tells you what the arguments are, and what the return value must be (interface)
- The step above doesn't care how you turn inputs to output (internals)
- Stop when we hit a sub-problem we can solve in a few steps with *built-in* functions

Thinking Algorithmically

- Top-down design only works if you understand
  - the problem, and
  - a systematic method for solving the problem
- ∴ it forces you to think **algorithmically**
- First guesses about how to break down the problem are often wrong
  - but functional approach contains effects of changes
  - ∴ don't be afraid to change the design

Combining the Practices

- Top-down design fits naturally with functional coding
    - Each piece of code has a well-defined interface, no (or few) side-effects
- Top-down design makes debugging easier
    - Easier to see where the bug occurs (higher-level function vs. sub-functions)
    - Easier to fix the bug by changing just one piece of code
- Top-down design makes testing easier
    - Each function has one *limited* job

Refactoring

- One mode of abstraction is **refactoring**
- The metaphor: numbers can be factored in many different ways; pick ones which emphasize the common factors

$$\begin{aligned} 144 = 9 \times 16 = 3 \times 3 \times 4 \times 2 \times 2 \\ 360 = 6 \times 60 = 3 \times 3 \times 4 \times 2 \times 5 \end{aligned}$$

Then you can re-use the common part of the work

Once we have some code, and it (more or less) works, re-write it to emphasize commonalities:

- Parallel and transparent naming
- Grouping related values into objects
- Common or parallel sub-tasks become shared functions
- Common or parallel over-all tasks become general functions

Grouping into Objects

- *Notice* that the same variables keep being used together
- *Create* a single data object (data frame, list, …) that includes them all as parts
- *Replace* mentions of the individual variables with mentions of parts of the unified object

Advantages of Grouping

- Clarity (especially if you give the object a good name)
- Makes sure that the right values are always present (pass the object as an argument to functions, rather than the components)
- Memorization: if you know you are going to want to do the same calculation many times on these data values, do it once when you create the object, and store the result as a component

Extracting the Common Sub-Task

- *Notice* that your code does the same thing, or nearly the same thing, in multiple places, as part doing something else
- *Extract* the common operation
- *Write* one function to do that operation, perhaps with additional arguments
- *Call* the new function in the old locations

Advantages of Extracting Common Operations

- Main code focuses on *what* is to be done, not *how* (abstraction, human understanding)
- Only have to test (and debug) one piece of code for the sub-task
- Improvements to the sub-task propagate everywhere
    - Drawback: bugs propagate everywhere too

Extracting General Operations

- *Notice* that you have several functions doing parallel, or nearly parallel, operations
- *Extract* the common pattern or general operation
- *Write* one function to do the general operation, with additional arguments (typically including functions)
- *Call* the new general function with appropriate arguments, rather than the old functions

Advantages of Extracting General Patterns

- Clarifies the logic of what you are doing (abstraction, human understanding, use of statistical theory)
- Extending the same operation to new tasks is easy, not re-writing code from scratch
- Old functions provide test cases to check if general function works
- Separate testing/debugging "puts the pieces together properly" from "gets the small pieces right"

Refactoring vs. Top-down design

Re-factoring tends to make code look more like the result of top-down design

This is no accident

Extended example: the jackknife

- Have an estimator $\hat{\theta}$ of parameter θ
  want the standard error of our estimate, $se_{\hat{\theta}}$
- The jackknife approximation:
  - omit case i, get estimate $\hat{\theta}_{(-i)}$
  - Take the variance of all the $\hat{\theta}_{(-i)}$
  - multiply that variance by $\frac{(n-1)^2}{n}$ to get $\approx$ variance of $\hat{\theta}$
- then $se_{\hat{\theta}} =$ square root of that variance

(Why $(n-1)^2/n$? Think about just getting the standard error of the mean)

Jackknife for the mean

```
mean.jackknife <- function(a_vector) {
  n <- length(a_vector)
  jackknife.ests <- vector(length=n)
  for (omitted.point in 1:n) {
    jackknife.ests[omitted.point] <- mean(a_vector[-omitted.point])
  }
  variance.of.ests <- var(jackknife.ests)
  jackknife.var <- ((n-1)^2/n)*variance.of.ests
  jackknife.stderr <- sqrt(jackknife.var)
  return(jackknife.stderr)
}
```

```
some_normals <- rnorm(100,mean=7,sd=5)
(formula_se_of_mean <- sd(some_normals)/sqrt(length(some_normals)))
```

```
## [1] 0.484986
```

```
all.equal(formula_se_of_mean,mean.jackknife(some_normals))
```

```
## [1] TRUE
```

Jackknife for Gamma Parameters

Recall our friend the method of moments estimator:

```
gamma.est <- function(the_data) {
  m <- mean(the_data)
  v <- var(the_data)
  a <- m^2/v
  s <- v/m
  return(c(a=a,s=s))
}
```

```
gamma.jackknife <- function(a_vector) {
  n <- length(a_vector)
  jackknife.ests <- matrix(NA,nrow=2,ncol=n)
  rownames(jackknife.ests) = c("a","s")
  for (omitted.point in 1:n) {
    fit <- gamma.est(a_vector[-omitted.point])
    jackknife.ests["a",omitted.point] <- fit["a"]
    jackknife.ests["s",omitted.point] <- fit["s"]
  }
  variance.of.ests <- apply(jackknife.ests,1,var)
  jackknife.vars <- ((n-1)^2/n)*variance.of.ests
  jackknife.stderrs <- sqrt(jackknife.vars)
  return(jackknife.stderrs)
```

Jackknife for Gamma Parameters

```
data("cats",package="MASS")
gamma.est(cats$Hwt)
```

```
##          a          s 
## 19.0653121  0.5575862
```

```
gamma.jackknife(cats$Hwt)
```

```
##          a          s 
## 2.74062279 0.07829436
```

Jackknife for linear regression coefficients

```
jackknife.lm <- function(df,formula,p) {
  n <- nrow(df)
  jackknife.ests <- matrix(0,nrow=p,ncol=n)
  for (omit in 1:n) {
    new.coefs <- lm(as.formula(formula),data=df[-omit,])$coefficients
    jackknife.ests[,omit] <- new.coefs
  }
  variance.of.ests <- apply(jackknife.ests,1,var)
  jackknife.var <- ((n-1)^2/n)*variance.of.ests
  jackknife.stderr <- sqrt(jackknife.var)
  return(jackknife.stderr)
}
```

```
cats.lm <- lm(Hwt~Bwt,data=cats)
# "Official" standard errors
sqrt(diag(vcov(cats.lm)))
```

```
## (Intercept)         Bwt
##   0.6922770   0.2502615
```

```
jackknife.lm(df=cats,formula="Hwt~Bwt",p=2)
```

```
## [1] 0.8314142 0.3166847
```

Refactoring the Jackknife

- Omitting one point or row is a common sub-task
- The general pattern:

```
figure out the size of the data
for each case
   omit that case
   repeat some estimation and get a vector of numbers
take variances across cases
scale up variances
take the square roots
```

- Refactor by extracting the common "omit one" operation
- Refactor by defining a general "jackknife" operation

The Common Operation

- *Problem*: Omit one particular data point from a larger structure
- *Difficulty*: Do we need a comma in the index or not?
- *Solution*: Works for vectors, lists, 1D and 2D arrays, matrices, data frames:

```
omit.case <- function(the_data,omitted_point) {
  data_dims <- dim(the_data)
  if (is.null(data_dims) || (length(data_dims)==1)) {
    return(the_data[-omitted_point])
  } else {
    return(the_data[-omitted_point,])
  }
}
```

Exercise: Modify so it also handles higher-dimensional arrays

The General Operation

```r
jackknife <- function(estimator,the_data) {
  if (is.null(dim(the_data))) { n <- length(the_data) }
  else { n <- nrow(the_data) }
  omit_and_est <- function(omit) {
    estimator(omit.case(the_data,omit))
  }
  jackknife.ests <- matrix(sapply(1:n, omit_and_est), ncol=n)
  var.of.reestimates <- apply(jackknife.ests,1,var)
  jackknife.var <- ((n-1)^2/n)* var.of.reestimates
  jackknife.stderr <- sqrt(jackknife.var)
  return(jackknife.stderr)
}
```

Could allow other arguments to `estimator`, spin off finding `n` as its own function, etc.

It works

```
jackknife(estimator=mean,the_data=some_normals)
```

```
## [1] 0.484986
```

```
all.equal(jackknife(estimator=mean,the_data=some_normals),
          mean.jackknife(some_normals))
```

```
## [1] TRUE
```

```
all.equal(jackknife(estimator=gamma.est,the_data=cats$Hwt),
          gamma.jackknife(cats$Hwt))
```

```
## [1] "names for current but not for target"
```

```
all.equal(jackknife(estimator=gamma.est,the_data=cats$Hwt),
          gamma.jackknife(cats$Hwt), check.names=FALSE)
```

```
## [1] TRUE
```

Exercise: Have `jackknife()` figure out component names for its output, if `estimator` has named components

It works

```
est.coefs <- function(the_data) {
  return(lm(Hwt~Bwt,data=the_data)$coefficients)
}
est.coefs(cats)
```

```
## (Intercept)         Bwt
##  -0.3566624   4.0340627
```

```
all.equal(est.coefs(cats), coefficients(cats.lm))
```

```
## [1] TRUE
```

```
jackknife(estimator=est.coefs,the_data=cats)
```

```
## [1] 0.8314142 0.3166847
```

```
all.equal(jackknife(estimator=est.coefs,the_data=cats),
          jackknife.lm(df=cats,formula="Hwt~Bwt",p=2))
```

```
## [1] TRUE
```

Refactoring + Testing

We have just tested the new code against the old to make sure we've not *added* errors

i.e., we have done **regression testing**

Summary

1. Top-down design is a recursive heuristic for coding
    - Split your problem into a few sub-problems; write code tying their solutions together
    - If any sub-problems still need solving, go write their functions
2. Leads to multiple short functions, each solving a limited problem
3. Disciplines you to think algorithmically
4. Once you have working code, re-factor it to make it look more like it came from a top-down design
    - Factor out similar or repeated sub-operations
    - Factor out common over-all operations

Further Refactoring of jackknife()

The code for `jackknife()` is still a bit clunky:

- Ugly `if-else` for finding `n`
- Bit at the end for scaling variances down to standard errors

```r
data_size <- function(the_data) {
  if (is.null(dim(the_data))) { n <- length(the_data) }
  else { n <- nrow(the_data) }
}
```

```r
scale_and_sqrt_vars <- function(jackknife.ests,n) {
  var.of.reestimates <- apply(jackknife.ests,1,var)
  jackknife.var <- ((n-1)^2/n)* var.of.reestimates
  jackknife.stderr <- sqrt(jackknife.var)
  return(jackknife.stderr)
}
```

Further Refactoring of jackknife()

Now invoke those functions

```
jackknife <- function(estimator,the_data) {
  n <- data_size(the_data)
  omit_and_est <- function(omit) {
    estimator(omit.case(the_data,omit))
  }
  jackknife.ests <- matrix(sapply(1:n, omit_and_est), ncol=n)
  return(scale_and_sqrt_vars(jackknife.ests,n))
}
```