Statistical Computing (36-350)

Lecture 15: Abstraction I: Refactoring

Cosma Shalizi and Vincent Vu

19 October 2011

Agenda

- Abstraction adjusts programming to human strengths
- Refactoring adjusts code to bring out commonalities
- Ways of refactoring: names, objects, common operations, general operations
- Example: The jack-knife

Abstraction

The point of abstraction is to 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 do this

We have talked about lots of examples of this already

Data structures; Functions; Interfaces; Functions as objects

Refactoring

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

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

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

Naming

R puts next to no limits on names of variables and functions
∴ we should use names that make sense to humans

- Names should indicate purpose or meaning. Call something `plot` or `predict` when, but only when, it plots or predicts.
- Similar objects should have similar names.

Example: conventions for functions related to random variables

`dnorm`	probability *d*ensity of *norm*al r.v.
`rnorm`	*r*andom value from *norm*al r.v.
`pnorm`	cumulative *p*robability of *norm*al r.v.
`qnorm`	*q*uantile of *norm*al r.v.
`dgamma`	probability *d*ensity of *gamma* r.v.
`ppois`	?
`rt`	?
`qchisq`	?

Why Care About Names?

Your code is easier to understand

Because it is easier to understand, it is easier to make repairs and improvements

Because it is easier to understand, people (including you) do not waste time trying to puzzle it out

On the other hand, because it is easier to understand, you are more easily replaced as a programmer

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 check 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

Re-factoring tends to make code look more like the result of top-down design
This is no accident

Extended example: the jackknife

Let's look at an example of using refactoring

Remember the jackknife from assignments: we have an estimator $\hat{\theta}$ of a parameter θ, and want to know the standard error of our estimate, $se_{\hat{\theta}}$.

The jackknife approximation is: omit case i, get estimate $\hat{\theta}_{(-i)}$. Take the variance of all the $\hat{\theta}_{(-i)}$, and multiply by $\frac{(n-1)^2}{n}$ to get $\approx$ variance of $\hat{\theta}$; then $se_{\hat{\theta}} =$ square root of that variance.

(Why $\frac{(n-1)^2}{n}$? Think about just getting the standard error of the mean)

Jackknife for gamma parameters

```
gamma.jackknife <- function(data) {
  n <- length(data)
  jackknife.ests <- matrix(NA,nrow=2,ncol=n)
  rownames(jackknife.ests) = c("a","s")
  for (omitted.point in 1:n) {
    fit <- gamma.est(data[-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 the mean

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

Jackknife for linear regression coefficients

```
jackknife.lm <- function(data,p) {
  n <- nrow(data)
  jackknife.ests <- matrix(0,nrow=p,ncol=n)
  for (omit in 1:n) {
    new.coefs <- lm(your.formula.here,data=data[-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.vars)
  return(jackknife.stderr)
}
```

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

Works for vectors, lists, 1D and 2D arrays, matrices, data frames:

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

EXERCISE: Modify so it also handles higher-dimensional arrays

The General Operation

```
jackknife <- function(estimator,data) {
  if (is.null(dim(data))) { n <- length(data) }
  else { n <- nrow(data) }
  jackknife.ests <- c()
  for (omit in 1:n) {
    reestimate <- estimator(omit.case(data,omit))
    jackknife.ests <- cbind(jackknife.ests,reestimate)
  }
  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,data=rnorm(n=400,mean=7,sd=5))
[1] 0.2361081
> est.coefs <- function(data) {
  return(lm(Hwt~Bwt,data=data)$coefficients)
}
> est.coefs(cats)
(Intercept)          Bwt
 -0.3566624   4.0340627
> jackknife(estimator=est.coefs,data=cats)
(Intercept)          Bwt
  0.8314142   0.3166847
```

Summary

Refactoring adjusts code to emphasize patterns

- Names
- Objects
- Common operations
- General operations

Refactoring makes code look more like top-down design
Refactoring usually involves abstraction
Abstraction emphasizes human strengths