Lecture 15: Top-Down Design and Refactoring

36-350

15 October 2014

In Previous Episodes

- Functions
- Multiple functions
- Debugging
- Testing

Agenda

- Top-down design of programs
- Re-factoring existing code to a better design
- Example: Jackknife

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

Abstraction

- 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

What About the Sub-Functions?

credit: [http://cheezburger.com/View/4517375744]

Thinking Algorithmically

- Top-down design only works if you understand
  - the problem, and

  - a systematic method for solving the problem
- $\therefore$ 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
  - $\therefore$ 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

Refactoring

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)
}
```

Jackknife for the mean

```
some_normals <- rnorm(100,mean=7,sd=5)
mean(some_normals)
```

```
## [1] 7.668
```

```
(formula_se_of_mean <- sd(some_normals)/sqrt(length(some_normals)))
```

```
## [1] 0.4844
```

```
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))
}
```

Jackknife for Gamma Parameters

```
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
## 19.0653  0.5576
```

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

```
##       a       s
## 2.74062 0.07829
```

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)
}
```

Jackknife for linear regression coefficients

```
cats.lm <- lm(Hwt~Bwt,data=cats)
coefficients(cats.lm)
```

```
## (Intercept)         Bwt
##     -0.3567      4.0341
```

```
# "Official" standard errors
sqrt(diag(vcov(cats.lm)))
```

```
## (Intercept)         Bwt
##      0.6923      0.2503
```

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

```
## [1] 0.8314 0.3167
```

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

```
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] 0.4844
```

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

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

It works

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

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

It works

```
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
##     -0.3567      4.0341
```

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

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

It works

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

```
## [1] 0.8314 0.3167
```

```
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

Further Refactoring of jackknife()

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

```
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))
}
```