I don't get your code
Neither do I
But it seems to work
The art of programing
Writing clear code
Tools for Reproducible Research

Karl Broman
Biostatistics & Medical Informatics, UW–Madison

kbroman.org
github.com/kbroman
@kwbroman
Course web: kbroman.org/Tools4RR
Basic principles

- **Code that works**
  No bugs; efficiency is secondary (or tertiary)

- **Readable**
  Fixable; extendible

- **Reusable**
  Modular; reasonably general

- **Reproducible**
  Re-runnable

- **Think before you code**
  More thought $\implies$ fewer bugs/re-writes

- **Learn from others' code**
  R itself; key R packages
Write programs for people, not computers

Break code into small functions

```r
get_grid_index <- function(vec, step) {
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  if(any(is.na(index)))
    index <- sapply(grid, function(a,b) {
      d <- abs(a-b)
      wh <- which(d==min(d))
      if(length(wh)>1) wh <- sample(wh, 1)
      wh
    }, vec)

  index
}
```
sampleone <-
function(vec)
   ifelse(length(vec)==1, vec, sample(vec, 1))

get_grid_index <-
function(vec, step)
{
   grid <- seq(min(vec), max(vec), by=step)
   index <- match(grid, vec)

   if(any(is.na(index)))
      index <- sapply(grid,
         function(a,b) {
            d <- abs(a-b)
            sampleone(which(d == min(d)))
         }, vec)

   index
}

sampleone <- function(vec) {
  ifelse(length(vec) == 1, vec, sample(vec, 1))
}

get_grid_index <- function(vec, step) {
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  if(any(is.na(index))) {
    for(i in seq(along=grid)) {
      d <- abs(grid[i] - vec)
      index[i] <- sampleone(which(d == min(d)))
    }
  }

  index
}
sampleone <-
function(vec)
  ifelse(length(vec) == 1, vec, sample(vec, 1))

get_grid_index <-
function(vec, step)
{
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  missing <- is.na(index)

  if(any(missing)) {
    for(i in which(missing)) {
      d <- abs(grid[i] - vec)
      index[i] <- sampleone(which(d == min(d)))
    }
  }

  index
}
Another example

```r
# rmvn: simulate from multivariate normal distribution
rmvn <- function(n, mu=0, V=diag(rep(1, length(mu)))) {
  p <- length(mu)

  if(any(dim(V) != p))
    stop("Dimension problem!")

  D <- chol(V)

  matrix(rnorm(n*p),ncol=p) %*% D + rep(mu,each=n)
}
```
Further examples

```r
# colors from blue to red
revrainbow <- function(n=256, ...) 
  rev(rainbow(start=0, end=2/3, n=n, ...))

# move values above/below quantiles to those quantiles
winsorize <- function(vec, q=0.006) 
{
  lohi <- quantile(vec, c(q, 1-q), na.rm=TRUE)
  if(diff(lohi) < 0)
    lohi <- rev(lohi)

  vec[!is.na(vec) & vec < lohi[1]] <- lohi[1]
  vec[!is.na(vec) & vec > lohi[2]] <- lohi[2]

  vec
}
```
Writing functions

- Break large tasks into small units.
  - Make each discrete unit a function.

- If you write the same code more than once, make it a function.

- If a line/block of code is complicated, make it a function.
Don't repeat yourself (or others)

► Avoid having repeated blocks of code.
► Create functions, and call those functions repeatedly.
► This is easier to maintain.
  – If something needs to be fixed/revised, you just have to do it the one time.
► Look at others' libraries/packages.
  – Don't write what others have already written (especially if they've done it better than you would have).
Don't make things too specific

- Write code that is a bit more general than your specific data
  - Don't assume particular data dimensions.
  - Don't forget about the possibility of missing values (even if your data doesn't have any).
  - Aim for re-use.

- Use function arguments
  - Don't assume particular data file names
  - Don't hard-code tuning parameters
  - R scripts can take command-line arguments:
    ```
    Rscript myscript.R input_file output_file
    args <- commandArgs(TRUE)
    ```
No global variables, ever!

- Don't refer directly to objects in your workspace.
- If a function needs something, pass it as an argument.
- (But what about really big data sets?)
No magic numbers

- Name numbers and use the names
  
  ```r
  max_iter <- 1000
  tol_convergence <- 0.0001
  ```

- Even better: include them as function arguments
# move values above/below quantiles to those quantiles
winsorize <- function(vec, q=0.006) {
  lohi <- quantile(vec, c(q, 1-q), na.rm=TRUE)
  if(diff(lohi) < 0)
    lohi <- rev(lohi)
  vec[!is.na(vec) & vec < lohi[1]] <- lohi[1]
  vec[!is.na(vec) & vec > lohi[2]] <- lohi[2]
  vec
}
# move values above/below quantiles to those quantiles

```r
winsorize<-function(vec,q=0.006)
{lohi<-quantile(vec,c(q,1-q),na.rm=TRUE)
  if(diff(lohi)<0)lohi<-rev(lohi)
  vec[!is.na(vec)&vec<lohi[1]]<-lohi[1]
  vec[!is.na(vec)&vec>lohi[2]]<-lohi[2]
  vec}
```

Don't let lines get too long

```r
get_grid_index <-
function(vec, step)
{
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  if(any(is.na(index)))
    index <- sapply(grid, function(a,b) { d <- abs(a-b); sampleone(which(d == min(d))) }, vec)

  index
}
```
Use parentheses to avoid ambiguity

```r
if( (ndraws1==1) && (ndraws2>1) ) {
  ...
}
leftval <- which( (map - start) <=0 )
```
Names: meaningful

- Make names descriptive but concise
- Avoid tmp1, tmp2, ...
- Only use i, j, x, y in the simplest situations
- If a function is named fV, what might it do?
- If an object is called nms, what could it be?
- Functions as verbs; objects as nouns
Names: consistent

- markers vs mnames
- camelCase vs. pothole_case
- nind vs n.var
- If a function/object has one of these, there shouldn't be a function/object with the other.
Names: avoid confusion

- Don't use both total and totals
- Don't use both n.cluster and n.clusters
- Don't use both result and results
- Don't use both Mat and mat
- Don't use both g and gg
Don't be cute

Richie Cotton
@richierocks

Giving myself Best Named Function Award for drop_the_bom that removes byte order marks from UTF-8 files.

5:17 AM - 11 Oct 2013

1 RETWEET 1 FAVORITE
Comments

- Comment the tricky bits and the major sections
- Don't belabor the obvious
- Don't comment bad code; rewrite it
- Document the input/output and purpose, not the mechanics
- Don't contradict the code
  - this happens if you revise the code but don't revise the related comments
- Comment code as you are writing it (or before)
- Plan to spend 1/4 of your time commenting
Error/warning messages

- Explain what's wrong (and where)
  - `error("nrow(X) != nrow(Y)")`

- Suggest corrective action
  - "You need to first run calc.genoprob()."

- Give details
  - "nrow(X) (", nrX, ") != nrow(Y) (", nrY, ")"

- Don't give error/warning messages that users won't understand.
  - X'X is singular.

- Don't let users do something stupid without warning

- Include error checking even in personal code.
Check data integrity

- Check that the input is as expected, or give warnings/errors.
- Write these in the first pass (though they're dull).
  - You may not remember your assumptions later
- These are useful for documenting the assumptions.
Program organization

- Break code into separate files (say 300 lines?)
- Each file includes related functions
- Files should be named meaningfully
- Include a brief comment at the top.
Create an R package!

▶ Make a personal package with bits of your own code
▶ Mine is R/broman, github.com/kbroman/broman

```r
# qqline corresponding to qqplot
qqline2 <- function(x, y, probs = c(0.25, 0.75), qtype = 7, ...) {
  stopifnot(length(probs) == 2)
  x <- quantile(x, probs, names=FALSE, type=qtype, na.rm = TRUE)
  y <- quantile(y, probs, names=FALSE, type=qtype, na.rm = TRUE)
  slope <- diff(y)/diff(x)
  int <- y[1L] - slope*x[1L]
  abline(int, slope, ...)
  invisible(c(intercept=int, slope=slope))
}
```
Complex data objects

- Keep disparate data together in a more complex structure.
  - lists in R
  - I also like to hide things in object attributes

- It's easier to pass such objects between functions

- Consider object-oriented programming
Avoiding bugs

- Learn to type well.
- Think before you type.
- Consider commenting before coding.
- Code defensively
  - Handle cases that "can't happen"
- Code simply and clearly
- Use modularity to advantage
- Think through all special cases
- Don't be in too much of a hurry
Basic principles

► Code that works
   No bugs; efficiency is secondary (or tertiary)

► Readable
   Fixable; extendible

► Reusable
   Modular; reasonably general

► Reproducible
   Re-runnable

► Think before you code
   More thought $\rightarrow$ fewer bugs/re-writes

► Learn from others' code
   R itself; key R packages
Summary

- Get the correct answers.
- Find a clear style and stick to it.
- Plan for the future.
- Be organized.
- Don't be too hurried.
- Learn from others.