Good Habits - Part 1

STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

gastonsanchez.com
github.com/gastonstat/stat133
Course web: gastonsanchez.com/teaching/stat133

Good Coding Habits I

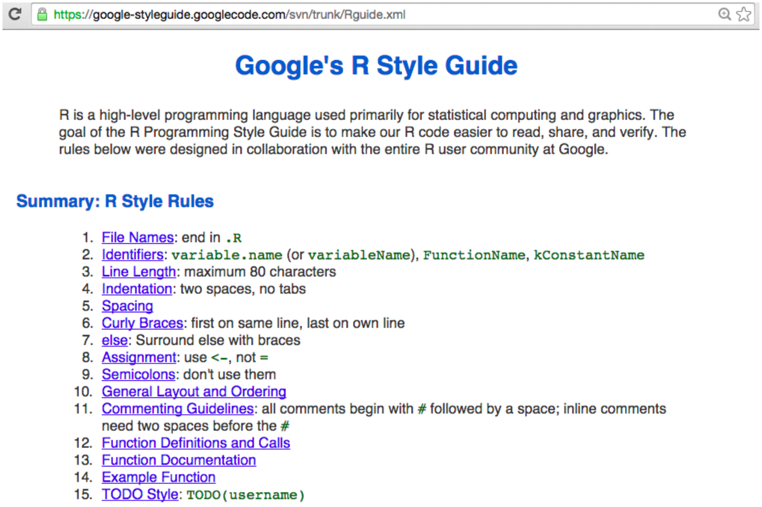
Code Habits

Now that you've worked with various R scripts, written some functions, and done some data manipulation, it's time to look at some good coding practices.

Code Habits

Popular style guides among useR's

- https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml
- http://adv-r.had.co.nz/Style.html

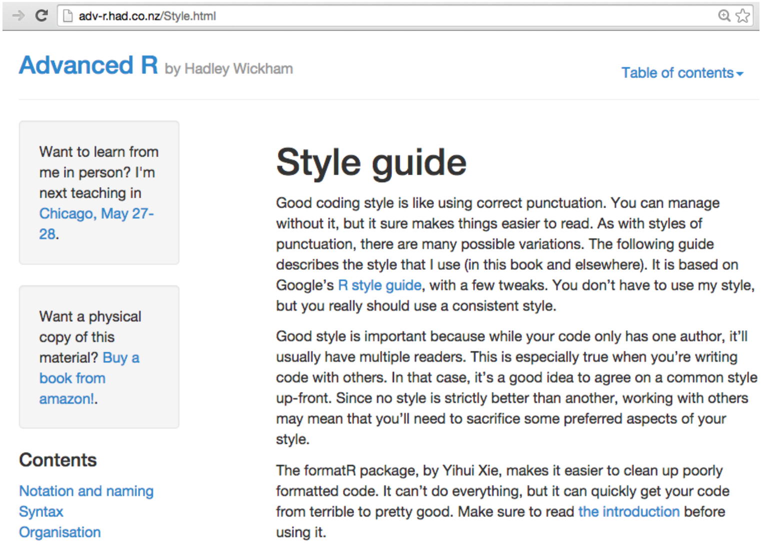

https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml

Google's R Style Guide

R is a high-level programming language used primarily for statistical computing and graphics. The goal of the R Programming Style Guide is to make our R code easier to read, share, and verify. The rules below were designed in collaboration with the entire R user community at Google.

Summary: R Style Rules

1. File Names: end in `.R`
2. Identifiers: `variable.name` (or `variableName`), `FunctionName`, `kConstantName`
3. Line Length: maximum 80 characters
4. Indentation: two spaces, no tabs
5. Spacing
6. Curly Braces: first on same line, last on own line
7. else: Surround else with braces
8. Assignment: use `<-`, not `=`
9. Semicolons: don't use them
10. General Layout and Ordering
11. Commenting Guidelines: all comments begin with `#` followed by a space; inline comments need two spaces before the `#`
12. Function Definitions and Calls
13. Function Documentation
14. Example Function
15. TODO Style: `TODO(username)`



Advanced R by Hadley Wickham

Table of contents

Want to learn from me in person? I'm next teaching in Chicago, May 27-28.

Want a physical copy of this material? Buy a book from amazon!.

Contents

Notation and naming
Syntax
Organisation

Style guide

Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read. As with styles of punctuation, there are many possible variations. The following guide describes the style that I use (in this book and elsewhere). It is based on Google's R style guide, with a few tweaks. You don't have to use my style, but you really should use a consistent style.

Good style is important because while your code only has one author, it'll usually have multiple readers. This is especially true when you're writing code with others. In that case, it's a good idea to agree on a common style up-front. Since no style is strictly better than another, working with others may mean that you'll need to sacrifice some preferred aspects of your style.

The formatR package, by Yihui Xie, makes it easier to clean up poorly formatted code. It can't do everything, but it can quickly get your code from terrible to pretty good. Make sure to read the introduction before using it.

R Scripts

Include Header information such as

- Who wrote / programmed it
- When was it done
- What is it all about
- How the code might fit within a larger program

R Scripts

Header example:

```
# ===================================================
# Some Title
# Author(s): First Last
# Date: month-day-year
# Description: what this code is about
# Data: perhaps is designed for a specific data set
# ===================================================
```

R Scripts

If you need to load R packages, do so at the beginning of your script, after the header:

```r
# ===================================================
# Some Title
# Author(s): First Last
# Date: month-day-year
# Description: what this code is about
# Data: perhaps is designed for a specific data set
# ===================================================

library(stringr)
library(ggplot2)
library(MASS)
```

Syntax

Indentation

- Do this systematically (RStudio editor helps a lot)
- This helps you and others to read and understand the code
- Can help in detecting errors in your code because it can expose lack of symmetry

Indentation

```r
# Don't do this!
if(!is.vector(x)) {
stop('x must be a vector')
} else {
if(any(is.na(x))){
x <- x[!is.na(x)]
}
}
total <- length(x)
x_sum <- 0
for (i in seq_along(x)) {
x_sum <- x_sum + x[i]
}
x_sum / total
```

Indentation

```r
# better with indentation
if(!is.vector(x)) {
  stop('x must be a vector')
} else {
  if(any(is.na(x))) {
    x <- x[!is.na(x)]
  }
}
total <- length(x)
x_sum <- 0
for (i in seq_along(x)) {
  x_sum <- x_sum + x[i]
}
x_sum / total
```

Syntax

White Spaces

- Use a lot of it
- around operators (assignment and arithmetic)
- between function arguments and list elements
- between matrix/array indices, in particular for missing indices
- Split long lines at meaningful places

White spaces

Avoid this

```
a<-2
x<-3
y<-log(sqrt(x))
3*x^7-pi*x/(y-a)
```

Much Better

```
a <- 2
x <- 3
y <- log(sqrt(x))
3*x^7 - pi * x / (y - a)
```

White spaces

```
# Don't
plot(x,y,col=rgb(0.5,0.7,0.4),pch='+',cex=5)


# OK
plot(x, y, col = rgb(0.5, 0.7, 0.4), pch = '+', cex = 5)
```

Readability

Lines should be broken/wrapped around so that they are less than 80 columns wide

```
# lines too long
histogram <- function(data){
hist(data, col = 'gray90', xlab = 'x', ylab = 'Frequency', main= 'Histogram of x
abline(v = c(min(data), max(data), median(data), mean(data)),
col = c('gray30', 'gray30', 'orange', 'tomato'), lty = c(2,2,1,1), lwd = 3)
}
```

Readability

Lines should be broken/wrapped aroung so that they are less than 80 columns wide

```r
# lines too long
histogram <- function(data) {
  hist(data, col = 'gray90', xlab = 'x', ylab = 'Frequency',
       main = 'Histogram of x')
  abline(v = c(min(data), max(data), median(data), mean(data)),
         col = c('gray30', 'gray30', 'orange', 'tomato'),
         lty = c(2,2,1,1), lwd = 3)
}
```

White spaces

```r
# this can be improved
stats <- c(min(x), max(x), max(x)-min(x),
  quantile(x, probs=0.25), quantile(x, probs=0.75), IQR(x),
  median(x), mean(x), sd(x)
)
```

White spaces

Don't be afraid of splitting one long line into individual pieces:

```r
# much better
stats <- c(
  min(x),
  max(x),
  max(x) - min(x),
  quantile(x, probs = 0.25),
  quantile(x, probs = 0.75),
  IQR(x),
  median(x),
  mean(x),
  sd(x)
)
```

White spaces

You can even do this:

```r
# also OK
stats <- c(
  min    = min(x),
  max    = max(x),
  range  = max(x) - min(x),
  q1     = quantile(x, probs = 0.25),
  q3     = quantile(x, probs = 0.75),
  iqr    = IQR(x),
  median = median(x),
  mean   = mean(x),
  stdev  = sd(x)
)
```

Syntax: Parentheses

Use parentheses for clarity even if not needed for order of operations.

```
a <- 2
x <- 3
y <- 4

a/y*x

## [1] 1.5

# better
(a / y) * x

## [1] 1.5
```

Use Parentheses

```r
# confusing
1:3^2

## [1] 1 2 3 4 5 6 7 8 9

# better
1:(3^2)

## [1] 1 2 3 4 5 6 7 8 9
```

Comments

Comment your code

- Add lots of comments
- But don't belabor the obvious
- Use blank lines to separate blocks of code and comments to say what the block does
- Remember that in a few months, you may not follow your own code any better than a stranger
- Some key things to document:
  - summarizing a block of code
  - explaining a very complicated piece of code
  - explaining arbitrary constant values

Line spaces and Comments

```r
MV <- get_manifests(Data, blocks)
check_MV <- test_manifest_scaling(MV, specs$scaling)
gens <- get_generals(MV, path_matrix)
names(blocks) <- gens$lvs_names
block_sizes <- lengths(blocks)
blockinds <- indexify(blocks)
```

Line spaces and Comments

```r
# ==================================================
# Preparing data and blocks indexification
# ==================================================
# building data matrix 'MV'
MV <- get_manifests(Data, blocks)
check_MV <- test_manifest_scaling(MV, specs$scaling)

# generals about obs, mvs, lvs
gens <- get_generals(MV, path_matrix)

# indexing blocks
names(blocks) <- gens$lvs_names
block_sizes <- lengths(blocks)
blockinds <- indexify(blocks)
```

Comments

Include comments to say what a block does, or what a block is intended for

```
# =====================================================
# Data: liga2015
# =====================================================
# For this session we'll be using the dataset that
# comes in the file 'liga2015.csv' (see github repo)
# This dataset contains basic statistics from the
# Spanish soccer league during the season 2014-2015
```

Comments

```r
x <- matrix(1:10, nrow = 2, ncol = 5)

# mean vectors by rows and columns
xmean1 <- apply(x, 1, mean)
xmean2 <- apply(x, 2, mean)

# Subtract off the mean of each row/column
y <- sweep(x, 1, xmean1)
z <- sweep(x, 2, xmean2)

# Multiply by the mean of each column (for some reason)
w <- sweep(x, 2, xmean1, FUN = "*")
```

Naming Style

Choose a consistent naming style for objects and functions

- `someObject` (lowerCamelCase)
- `SomeObject` (UpperCamelCase)
- `some_object` (underscore separation)
- `some.object` (dot separation)

Naming Style

Avoid using names of standard R objects

- `mean`
- `vector`
- `list`
- `data`
- `c`
- `colors`
- *ETC*

Naming Style

Better to do something like this

- `xmean`
- `xvector`
- `xlist`
- `xdata`
- `xc`
- `xcolors`
- *ETC*

Good coding practices: Syntax

Code Files

- Break code into separate files (¡2000-3000 lines per file)
- Give files meaningful names
- Group related functions within a file

Literate Programming

"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do"

Donald Knuth. "Literate Programming (1984)"

Literate Programming

Instructing a computer what to do

```
# good for computers
if (is.numeric(x) & x > 0 & x %% 1 == 0) TRUE else FALSE
```

Literate Programming

Instructing a computer what to do

```
# good for computers
if (is.numeric(x) & x > 0 & x %% 1 == 0) TRUE else FALSE
```

Explaining a human being what we want a computer to do

```
# good for humans
is_positive_integer(x)
```

Functions

Functions

- Functions are tools and operations
- Functions form the building blocks for larger tasks
- Functions allow us to reuse blocks of code easily for later use
- Use functions whenever possible
- Try to write functions rather than carry out your work using blocks of code

Length of Functions

- Don't write long functions
- Rewrite long functions by converting collections of related expression into separate functions
- Smaller functions are easier to debug, easier to understand, and can be combined in a modular fashion
- Functions shouldn't be longer than one visible screen (with reasonable font)

Length of Functions

- Separate small functions
- are easier to reason about and manage
- are easier to test and verify they are correct
- are more likely to be reusable

Some considerations

- Think about different scenarios and contexts in which a function might be used
- Can you generalize it?
- Who will use it?
- Who is going to maintain the code?

Naming Functions

- Use descriptive names
- Readers (including you) should infer the operation by looking at the call of the function

Functions

Functions should:

- be modular (having a single task)
- have meaningful name
- have a comment describing their purpose, inputs and outputs

Functions: Example

```r
# find mean of Y on the data z, Y in last column, and predict at xnew
meany <- function(predpt,nearxy) {
   ycol <- ncol(nearxy)
   mean(nearxy[,ycol])
}

# find variance of Y in the neighborhood of predpt
vary <- function(predpt,nearxy) {
   ycol <- ncol(nearxy)
   var(nearxy[,ycol])
}

# fit linear model to the data z, Y in last column, and predict at xnew
loclin <- function(predpt,nearxy) {
   ycol <- ncol(nearxy)
   bhat <- coef(lm(nearxy[,ycol] ~ nearxy[,-ycol]))
   c(1,predpt) %*% bhat
}
```

source: Norm Matloff

Functions and Global Variables

- Functions should not modify global variables
- except connections or environments
- should not change global `par()` settings

Strongly Recommended

Look at other people's code

- https://github.com/hadley
- https://github.com/yihui
- https://github.com/karthik
- https://github.com/kbroman
- https://github.com/cboettig
- https://github.com/garrettgman

Test Yourself

What's wrong with this function?

```r
average <- function(x) {
  l <- length(x)
  for(i in l) {
    y[i] <- x[i]/l
    z <- sum(y[1:l])
    return(as.numeric(z))
  }
}
```

Test Yourself

What's wrong with this function?

```r
freq_table <- function(x) {
  table <- table(x)
  'category' <- levels(x)
  'count' <- print(table)
  'prop' <- table/length(x)
  'cumcount' <- print(table)
  'cumprop' <- table/length(x)
  if(is.factor(x)) {
    return(data.frame(rownames=c('category', 'count','prop',
                                'cumcount','cumprop')))
  } else {
    stop('Not a factor')
  }
}
```

Discussion

- What other suggestions do you have?
- How could we restructure the code, to make it easier to read?
- Grab a buddy and practice "code review". We do it for methods and papers, why not code? Our code is a major scientific product and the result of a lot of hard work!