# Good Habits in R Programming

## STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

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

# Good Coding Habits

# 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)`

adv-r.had.co.nz/Style.html

**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.

# Editor

## Text Editor

- Text editor $\neq$ word processor
- Use a good text editor
- e.g. vim, sublime text, text wrangler, notepad, etc
- With syntax highlighting
- Or use an Integrated Development Environment (IDE) like RStudio

# Without Syntax Highlighting

```
a <- 2
x <- 3
y <- log(sqrt(x))
3*x^7 - pi * x / (y - a)
"some strings"
dat <- read.table(file = 'data.csv', header = TRUE)
```

# Syntax Highlighting

```r
a <- 2
x <- 3
y <- log(sqrt(x))
3*x^7 - pi * x / (y - a)
"some strings"
dat <- read.table(file = 'data.csv', header = TRUE)
```

# Syntax Highlight

Without highlighting it's harder to detect syntax errors:

```
numbers <- c("one", "two, "three")

if (x > 0) {
  3 * x + 19
} esle {
  2 * x - 20
}
```

# Syntax Highlight

With highlighting it's easier to detect syntax errors:

```
numbers <- c("one", "two, "three")
```

```
if (x > 0) {
  3 * x + 19
} esle {
  2 * x - 20
}
```

# Your Turn

Which instruction is free of errors

A) `mean(numbers, na.mr = TRUE)`

B) `read.table(~/Documents/rawdata.txt, sep = '\t')`

C) `barplot(x, horiz = TURE)`

D) `matrix(1:12, nrow = 3, ncol = 4)`

# Use an IDE

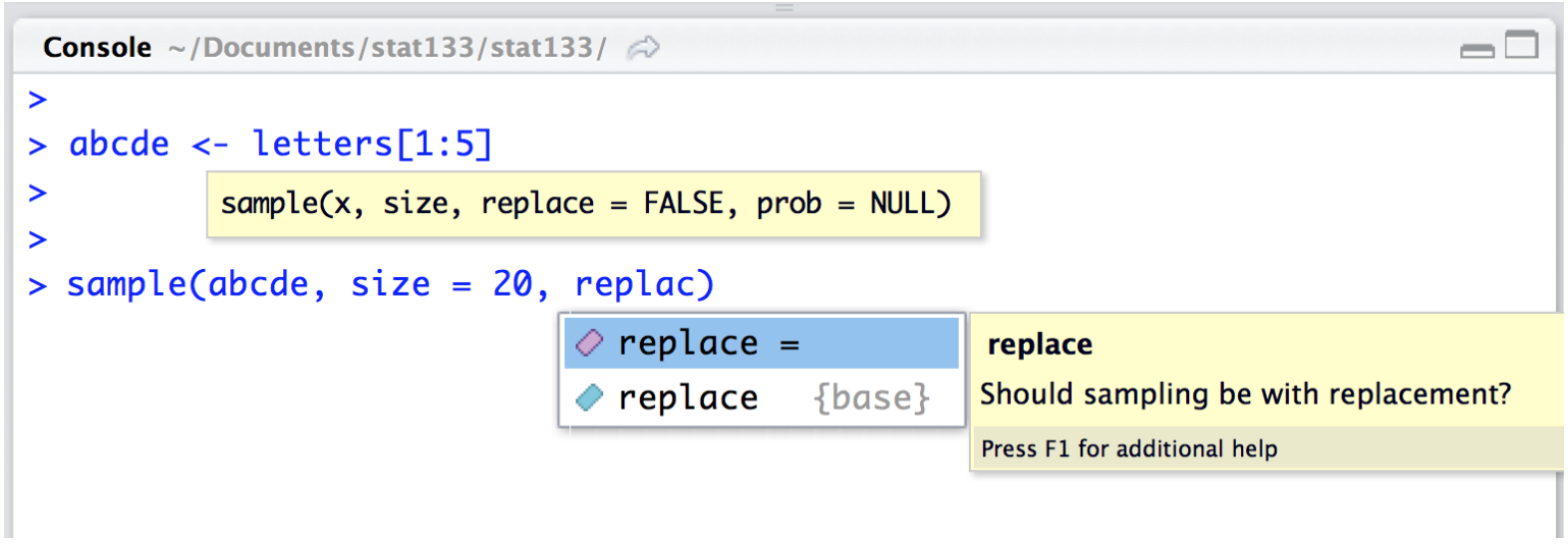
- Syntax highlighting
- Syntax aware
- Able to evaluate R code
  - by line
  - by selection
  - entire file
- Command completion

# Use an IDE

## Use an IDE with autocompletion

# Use an IDE

Use an IDE that provides helpful documentation



# Good Source Code

# Literate Programming

Think about programs/scripts/code as **works of literature**

# Important Aspects

- Indentation of lines
- Use of spaces
- Use of comments
- Naming style
- Use of white space
- Consistency

# Literate Programming

## Good source code

- Well readable by humans
- As much self-explaining as possible

# 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

- Choose the names of variables carefully
- Explain what each variable means
- Strive for a program that is comprehensible
- Introduce concepts in an order that is best for human understanding

(From Donald Knuth's: Literate Programming, 1984)

# Literate Programming

Instructing a computer what to do

```
# good for computers (not much for humans)
if (is.numeric(x) & x > 0 & x %% 1 == 0) TRUE else FALSE
```

# Literate Programming

Instructing a computer what to do

```
# good for computers (not much for humans)
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)
```

# Literate Programming

```
# example
is_positive_integer <- function(x) {
  (is.numeric(x) & x > 0 & x %% 1 == 0)
}

is_positive_integer(2)

## [1] TRUE

is_positive_integer(2.1)

## [1] FALSE
```

# Indentation

- Keep your indentation style consistent
- There is more than one way of indenting code
- There is no "best" style that everyone should be following
- You can indent using spaces or tabs (but don't mix them)
- Can help in detecting errors in your code because it can expose lack of symmetry
- Do this systematically (RStudio editor helps a lot)

# Indentation

```
# 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
}
```

# Indenting Styles

```
# style 1
find_roots <- function(a = 1, b = 1, c = 0)
{
  if (b^2 - 4*a*c < 0)
  {
    return("No real roots")
  } else
  {
    return(quadratic(a = a, b = b, c = c))
  }
}
```

# Indenting Styles

```
# style 2
find_roots <- function(a = 1, b = 1, c = 0) {
  if (b^2 - 4*a*c < 0) {
    return("No real roots")
  } else {
    return(quadratic(a = a, b = b, c = c))
  }
}
```

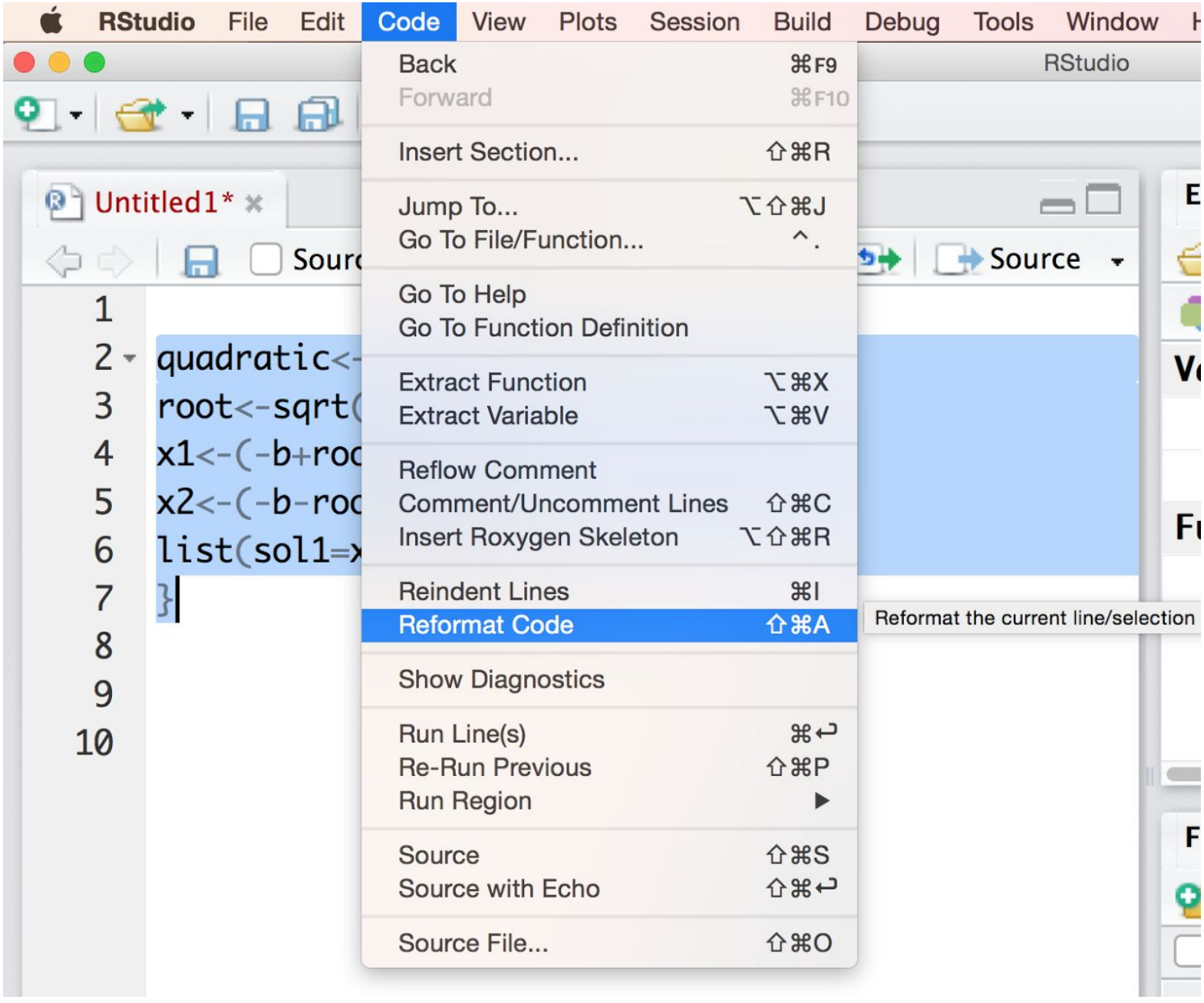
# Indentation

## Benefits of code indentation:

- Easier to read
- Easier to understand
- Easier to modify
- Easier to maintain
- Easier to enhance

# Reformat Code in RStudio

- RStudio provides *code reformatting* (use it!)
- Click **Code** on the menu bar
- Then click **Reformat Code**


RStudio File Edit Code View Plots Session Build Debug Tools Window
RStudio
Untitled1*
Source
1
2 quadratic<-
3 root<-sqrt(
4 x1<-(-b+roc
5 x2<-(-b-roc
6 list(sol1=x
7 }
8
9
10
Back ⌘F9
Forward ⌘F10
Insert Section... ⇧⌘R
Jump To... ⌥⇧⌘J
Go To File/Function... ^.
Go To Help
Go To Function Definition
Extract Function ⌥⌘X
Extract Variable ⌥⌘V
Reflow Comment
Comment/Uncomment Lines ⇧⌘C
Insert Roxygen Skeleton ⌥⇧⌘R
Reindent Lines ⌘I
Reformat Code ⇧⌘A
Reformat the current line/selection
Show Diagnostics
Run Line(s) ⌘↵
Re-Run Previous ⇧⌘P
Run Region ▶
Source ⇧⌘S
Source with Echo ⇧⌘↵
Source File... ⇧⌘O

# Reformat Code in RStudio

```
# unformatted code
quadratic<-function(a=1,b=1,c=0){
root<-sqrt(b^2-4*a*c)
x1<-(-b+root)/2*a
x2<-(-b-root)/2*a
list(sol1=x1,sol2=x2)
}
```

# Reformat Code in RStudio

```r
# unformatted code
quadratic<-function(a=1,b=1,c=0){
root<-sqrt(b^2-4*a*c)
x1<-(-b+root)/2*a
x2<-(-b-root)/2*a
list(sol1=x1,sol2=x2)
}
```

```r
# reformatted code
quadratic <- function(a = 1,b = 1,c = 0) {
  root <- sqrt(b ^ 2 - 4 * a * c)
  x1 <- (-b + root) / 2 * a
  x2 <- (-b - root) / 2 * a
  list(sol1 = x1,sol2 = x2)
}
```

# Meaningful Names

# 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

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

# Naming Style

If you're thinking about using names of R objects, prefer something like this

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

# Naming Style

Better to add meaning like this

- `mean_salary`
- `input_vector`
- `data_list`
- `data_table`
- `first_last`
- `some_colors`
- *etc*

# Naming Style

```
# what does getThem() do?
getThem <- function(values, y) {
  list1 <- c()

  for (i in values) {
    if (values[i] == y)
      list1 <- c(list1, x)
  }
  return(list1)
}
```

# Naming Style

```
# this is more meaningful
getFlaggedCells <- function(gameBoard, flagged) {
  flaggedCells <- c()

  for (cell in gameBoard) {
    if (gameBoard[cell] == flagged)
      flaggedCells <- c(flaggedCells, x)
  }
  return(flaggedCells)
}
```

# Meaningful Distinctions

```
# argument names 'a1' and 'a2'?
move_strings <- function(a1, a2) {
  for (i in seq_along(a1)) {
    a1[i] <- toupper(substr(a1, 1, 3))
  }
  a2
}
```

# Meaningful Distinctions

```r
# argument names 'a1' and 'a2'?
move_strings <- function(a1, a2) {
  for (i in seq_along(a1)) {
    a1[i] <- toupper(substr(a1, 1, 3))
  }
  a2
}
```

```r
# argument names
move_strings <- function(origin, destination) {
  for (i in seq_along(origin)) {
    destination[i] <- toupper(substr(origin, 1, 3))
  }
  destination
}
```

# Pronounceable Names

```r
# cryptic abbreviations
DtaRcrd102 <- list(
  nm = 'John Doe',
  bdg = 'Valley Life Sciences Building',
  rm = 2060
)
```

# Pronounceable Names

```r
# cryptic abbreviations
DtaRcrd102 <- list(
  nm = 'John Doe',
  bdg = 'Valley Life Sciences Building',
  rm = 2060
)
```

```r
# pronounceable names
Customer <- list(
  name = 'John Doe',
  building = 'Valley Life Sciences Building',
  room = 2060
)
```

# Your Turn

Which of the following is NOT a valid name:

- A) `x12345`
- B) `data_`
- C) `oBjEcT`
- D) `5ummary`
- E) `data.frame`

# 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

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

# White spaces

```
# Avoid this
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

```
# 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

- Spacing forms the second important part in code indentation and formatting.
- Spacing makes the code more readable
- Follow proper spacing through out your coding
- Use spacing consistently

# 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:

```
# 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:

```
# 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)
)
```

# White spaces

- All commas and semicolons must be followed by single whitespace
- All binary operators should maintain a space on either side of the operator
- Left parenthesis should start immediately after a function name
- All keywords like `if`, `while`, `for`, `repeat` should be followed by a single space.

# White spaces

All binary operators should maintain a space on either side of the operator

```
# NOT Recommended
a=b-c
a = b-c
a=b - c;

# Recommended
a = b - c
```

# White spaces

All binary operators should maintain a space on either side of the operator

```
# Not really recommended
z <- 6*x + 9*y

# Recommended (option 1)
z <- 6 * x + 9 * y

# Recommended (option 2)
z <- (7 * x) + (9 * y)
```

# White spaces

Left parenthesis should start immediately after a function name

```
# NOT Recommended
read.table ('data.csv', header = TRUE, row.names = 1)

# Recommended
read.table('data.csv', header = TRUE, row.names = 1)
```

# White spaces

All keywords like `if`, `while`, `for`, `repeat` should be followed by a single space.

```
# not bad
if(is.numeric(object)) {
  mean(object)
}

# much better
if (is.numeric(object)) {
  mean(object)
}
```

# 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

```
# 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

# 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

```
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

```
# ================================================
# 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)
```

# Line spaces and Comments

Different line styles:

```
##################################################

# ================================================

# ************************************************

# ------------------------------------------------
```

# Line spaces and Comments

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

# Line spaces and Comments

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

```
# ---- Preparing data and blocks indexification ----

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

# 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 = "*")
```

# About Comments

Be careful with your comments (you never know who will end up looking at your code, or where you'll be in the future)

```
# F***ing piece of code that drives me bananas

# wtf function

# best for loop ever
```

# 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

# 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:

```
# ================================================
# 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)
```

# 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

## Some rules of thumb (in order of preference)

- Ideal length between 2 and 4 lines of code
- No more than 10 lines
- No more than 20 lines
- Should not exceed the size of the text editor window

# 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

```
# 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

# DRY

# Don’t Repeat Yourself

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

# DRY

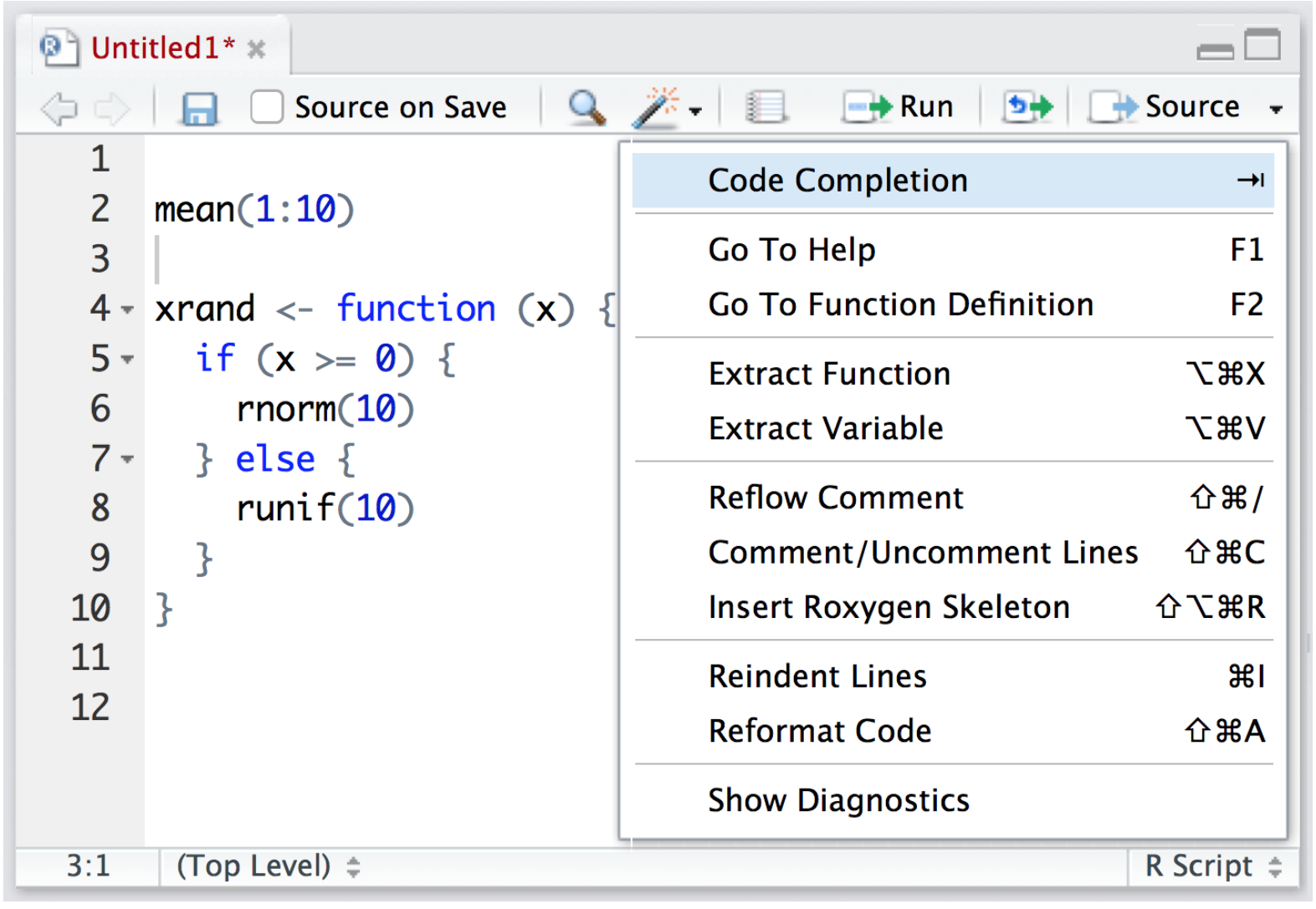
```
# avoid repetition
plot(x, y, type = 'n')
points(x[size == 'xsmall'], y[size == 'xsmall'], col = 'purple')
points(x[size == 'small'], y[size == 'small'], col = 'blue')
points(x[size == 'medium'], y[size == 'medium'], col = 'green')
points(x[size == 'large'], y[size == 'large'], col = 'orange')
points(x[size == 'xlarge'], y[size == 'xlarge'], col = 'red')
```

# DRY

```
# avoid repetition
plot(x, y, type = 'n')
points(x[size == 'xsmall'], y[size == 'xsmall'], col = 'purple')
points(x[size == 'small'], y[size == 'small'], col = 'blue')
points(x[size == 'medium'], y[size == 'medium'], col = 'green')
points(x[size == 'large'], y[size == 'large'], col = 'orange')
points(x[size == 'xlarge'], y[size == 'xlarge'], col = 'red')
```

```
# avoid repetition
size_colors <- c('purple', 'blue', 'green', 'orange', 'red')
plot(x, y, type = 'n')
for (i in seq_along(levels(size))) {
  points(x[size == i], y[size == i], col = size_colors[i])
}
```

# RStudio Code Tools



# RStudio Shortcuts

Rstudio

## Keyboard Shortcuts

### Console

| Description | Windows & Linux | Mac |
|---|---|---|
| Move cursor to Console | Ctrl+2 | Ctrl+2 |
| Clear console | Ctrl+L | Command+L |
| Move cursor to beginning of line | Home | Command+Left |
| Move cursor to end of line | End | Command+Right |
| Navigate command history | Up/Down | Up/Down |
| Popup command history | Ctrl+Up | Command+Up |
| Interrupt currently executing command | Esc | Esc |
| Change working directory | Ctrl+Shift+H | Ctrl+Shift+H |

# 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

# Your Own Style

- It takes time to develop a personal style
- Try different styles and see which one best fits you
- Sometimes you have to adapt to a company's style
- There is no one single best style

# Test Yourself

What's wrong with this function?

```
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?

```
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!