# R Style

Paul E. Johnson[1] [2]

[1]Department of Political Science

[2]Center for Research Methods and Data Analysis, University of Kansas

2015

# Outline

# Outline

# Overview

- This presentation summarizes the vignette R Style that is distributed with the rockchalk package
- Primary emphasis: write code that sophisticated users will be able to read
- Suggestions:
  - Write code in the same style used by R Core Team members as exemplified in the R source code
  - Write code in the style that R uses to display itself

# Outline

# 1. Indentation

- Use 4 spaces to indent sections
- A good editor will convert a TAB you type into 4 spaces
- If you use TAB symbols, set your editor to display them as 4 spaces

# 2. The assignment symbol "<-"

- Not the equal sign

# 3. Blank spaces

This is a GNU coding standard.

- Around symbols! Spaces required on both sides of
  - math operators: - + * /
  - logical operators: = == | || & &&
  - R symbols <- %*% %o% %in%
- One space before opening "(" or "{" in if and for statements
- One space after
  - comma
  - closing ")" or "}"

# 3. Blank spaces

- Unnecessary blank spaces are considered harmful, such as

```
( x ≤ y )
```

- Spaces between function name and parens

```
lm (y ~ x)
```

# 3. Blank spaces

- There is a great deal of variety about equal signs in function calls.
- Once you get used to the GNU way of writing, this looks best

```
m1 <- lm(y ~ x, data = red, subset = x < 77, model =
    FALSE)
```

- but if you read much code, you find many authors "snug up" around the equal signs.

```
m1 <- lm(y~x, data=red, subset=x < 77, model=FALSE)
```

- The latter seems horrible to me now, but I can't deny
  - it is widely used by authors smarter than I, and
  - publishers may insist on it to keep program code on the page.

# 4. Squiggly braces

- For loops, if statements, and the like generally have the opening squggly brace on the same line as the language construct

```
if (some statement) {
    y <- 1
...
}
```

Note the "{" and "}" are not vertically aligned, as they would be in much C, C++ or Java code.
Main difficulty: indentation eats up "empty space" on left side of page. Some text editors do this

# 4. Squiggly braces ...

```
myFn <- function(x = 31, y = 44, z = NULL){
                                                    j <- 1
                                                    i <- 5


...
```

- To avoid that, you'll generally see functions in R Core code declared like so:

```
myGiantLongBoringFunctionName <- function(x = 31, y = 44, z
    = NULL)
{
    j <- 1
...
```

- Or Possibly

# 4. Squiggly braces ...

```
myGiantLongBoringFunctionName <-
function(x = 31, y = 44, z = NULL)
{
    j <- 1
...
```

- These were used to work around indentation styles of various editors

# 5. Evolving Indentation Standards

- Emacs ESS now has a family of indentation styles, the default is their interpretation of the R standard.
  - TAB key inserts 4 spaces
- Consider plot.lm, which appears in R source code as:

```
plot.lm <-
function (x, which = c(1L:3L,5L), ## was which = 1L:4L,
    caption = list("Residuals vs Fitted", "Normal Q–Q",
    "Scale–Location", "Cook's distance",
    "Residuals vs Leverage",
    expression("Cook's dist vs Leverage  " * h[ii] / (1 - h[
        ii]))),
    panel = if(add.smooth) panel.smooth else points,
    sub.caption = NULL, main = "",
    ask = prod(par("mfcol")) < length(which) &&
        dev.interactive(), ...,
```

# 5. Evolving Indentation Standards ...

```
    id.n = 3, labels.id = names(residuals(x)), cex.id = 0.75
        ,
    qqline = TRUE, cook.levels = c(0.5, 1.0),
    add.smooth = getOption("add.smooth"),
    label.pos = c(4,2), cex.caption = 1)
{
```

- Emacs 24.4 with ESS 15.09 turns that into this:

```
plot.lm <-
    function (x, which = c(1L:3L,5L), ## was which = 1L:4L,
              caption = list("Residuals vs Fitted", "Normal Q–Q",
                             "Scale–Location", "Cook's distance",
                             "Residuals vs Leverage",
                             expression("Cook's dist vs Leverage  " * h[ii] / (1
                                 - h[ii]))),
              panel = if(add.smooth) panel.smooth else points,
              sub.caption = NULL, main = "",
              ask = prod(par("mfcol")) < length(which) && dev.interactive(), ...
                  ,
              id.n = 3, labels.id = names(residuals(x)), cex.id = 0.75,
              qqline = TRUE, cook.levels = c(0.5, 1.0),
              add.smooth = getOption("add.smooth"),
```

# 5. Evolving Indentation Standards ...

```
                label.pos = c(4,2), cex.caption = 1)
    {
```

Note that Emacs-ESS will not let the opening squiggly brace go flush left, it is always indented 4 spaces.

- Should coders insert lots of line breaks within function declarations? Some coders like a line break after each argument is defined:

```
plot.lm <-
function (x,
    which = c(1L:3L,5L), ## was which = 1L:4L,
    caption = list("Residuals vs Fitted", "Normal Q-Q", "
        Scale-Location",
        "Cook's distance", "Residuals vs Leverage",
        expression("Cook's dist vs Leverage  " * h[ii] / (1
            - h[ii]))),
    panel = if(add.smooth) panel.smooth else points,
    sub.caption = NULL,
```

# 5. Evolving Indentation Standards ...

```
    main = "",
    ask = prod(par("mfcol")) < length(which) &&
        dev.interactive(),
    ...,
    id.n = 3,
    labels.id = names(residuals(x)),
    cex.id = 0.75,
    qqline = TRUE,
    cook.levels = c(0.5, 1.0),
    add.smooth = getOption("add.smooth"),
    label.pos = c(4,2),
    cex.caption = 1)
    {
```

You don't generally see that in the R code prepared by R Core Team.

# 6. I suggest "} else {"

- It is possible to write code that runs if it is inside a closure (function)
  - but it does not run from the command line.
- Example

```
i <- 7
if (i < 5) {
      j <- 1
}
else {
   j <- 12
}
```

- will fail at the command prompt

# 6. I suggest "} else {" ...

```
> if (i < 5) {
+     j <- 1
+ }
> else {
Error: unexpected 'else' in "else"
```

- But inside a function it will succeed:

```
myfn <- function(i){
    if (i < 5) {
        j <- 1
    }
    else {
        j <- 12
    }
j
}
```

- Try that:

# 6. I suggest "} else {" ...

```
> myfn(99)
[1] 12
> myfn(1)
[1] 1
```

- Why does it fail in the command line, but succeed inside the function?
- Why do you care?
  - While preparing a function, you may want to run the commands "line by line" in a session, to find out what they do!
- Alternatives, if you don't like "} else {"
  - Write your function then run it in the debugger.

# Outline

# Install the formatR package

- The "tidy.source" function

```
> myfn <- function(x){ if (x < 7) {i = 77; print(paste("x is
    less than 7 but i is", i))} else {print("x is excessive
  ") }}
> library(formatR)
> tidy.source(source = "clipboard", replace.assign = TRUE)
function(x) {
    if (x < 7) {
        i <- 77
        print(paste("x is less than 7 but i is", i))
    } else {
        print("x is excessive")
    }
}
```

- Will fail with errors if you have comments inserted in middle of lines.

# Outline

# 1. Names to avoid

- Don't create confusion by creating new functions with names like "seq()", "rep()", "lm()", or such
  - would obscure access to functions from R base
- Now R Core Namespace policy has "defended" many functions from that accidental abuse
  - stats::lm() can find the lm function in the stats package, even if you have lm in your packages
- Still wise to avoid creating new functions with same name because
  - Confuse/frustrate experts who might read your code and help you with it
  - Confuse yourself during your session

# 2. Suggest Camel Case Function Names

- If you are naming a new function, don't use periods for punctuation
- Better to write

```
getParms <- function(x, y, z) {
```

than

```
get.parms <- function(x, y, z) {
```

- Why?
  - The R object framework has "generic functions" like "plot" and "summary"
  - Which have customized "methods" (implementations) like "plot.lm", "plot.glm", etc.
    - A class name follows the period

# 2. Suggest Camel Case Function Names ...

- In the R runtime system, calculations are sent among functions by parsing the last part of the method name

- Your "get.parms" function makes a reader think there is an object of type "parms" and a generic function named "get".

# 3. Think Carefully on Function Names

- Short names for frequently used functions & arguments
- Think of R's common pieces. When you create your own classes, name your functions similarly.

# Outline

# 1. No funny symbols

- Variable names
  - begin with letters, generally SMALL letters
  - include only letters, numbers, as well as “_”, “.”
  - AND NO math symbols like “-”and “+?” or “%” “^” “&”!
- See R base function “make.names” which can clean up name vectors.

# 2. Variable names to Avoid

- "T" or "F". Cause confusion with abbreviated TRUE and FALSE
- function names in R.
  - Previously, was possible to obliterate R base functions by declaring variables like "seq" and "rep"
  - Now still confusing to readers if you name a variable "c" or "rep".

# 3. Long and Short: When to be terse?

- Long name OK for something you use once or twice
- If used often, create a 1-5 letter name.

# 4. Append variations on end of name

- Given a variable

```
uranium
```

- don't do this

```
y <- uranium
```

  - or this

```
logu <- log(uranium)
```

- Please consider this:

```
uraniumlog <- log(uranium)
```

# any of these (which are all better than y or logu

```
uraniumln <- log(uranium)
```

```
uranium.log <- log(uranium)
```

```
uranium_log <- log(uranium)
```

```
ulog <- log(uranium)
```

- Why? related things stay together alphabetically! run "ls()"

# Outline

# 1. Work with a fixed width font

- If you have a programmer's file editor that uses a proportionally spaced font, get a different font, or editor

# 2. Use the ###, ##, and # style for indentations

- ## means a comment indented to match the context
- ### means flush left comment
- # means comment at far right
- Advice:
  1. don't append comments at end of lines (no matter how tempting to 'save space')
  2. develop a style to insert your comments either before or after lines they address. Be consistent! I'm trying to remember to use the BEFORE strategy

# 3. Keep Short line lengths

- Suggestion: 80 characters or less per line
- While writing code, I'll often have very long lines that take advantage of the wide screen. Sometimes I forget, but I try to go back and cut lines into 80-100 character widths.
- Some evidence suggests humans read badly with long lines
- Long lines don't translate well into documents, and they either
  - go off the right edge of the page, or
  - have "line breaks" at bad spots
- This is required in R documentation, where packages with very long lines in Rd files are rejected.
- Relates to problem of "multi line strings", which are discouraged in the wider programming arena, but tolerated in R.

# 3. Keep Short line lengths ...

- An R user (me) might write all on one line:

```
if (!c("rcreg") %in% class(object)) stop("predict.rcreg is
    intended for rcreg objects, which are created by
    residualCenter in the rockchalk package")
```

- But it is certainly better to write 3 lines, using paste to connect them together:

```
if (!c("rcreg") %in% class(object)) stop(paste("predict.rcreg",
   "is intended for rcreg objects, which are created",
   "by residualCenter in the rockchalk package"))
```