

# R-CodingStandard

A coding standard for R that combines:

- The book ‘Advanced R’ by Hadley Wickham, CRC Press, 2014
- The book ‘R packages’ by Hadley Wickham, O’Reilly Media, Inc., 2015
- [Google’s R Style Guide](#)
- The document ‘Rchaeology: Idioms of R Programming’ by Paul E. Johnson, January 28, 2015, [PDF](#)
- The document ‘Rtips. Revival 2014!’ by Paul E. Johnson, March 24, 2014, [PDF](#)
- The document ‘R Style. An Rchaeological Commentary.’ by Paul E. Johnson, February 13, 2015, [PDF](#)
- The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>
- The C++ Core Guidelines, <https://github.com/isocpp/CppCoreGuidelines>

This coding standard does not say what to do. It does inventorise what other references tell what to do. The reader is free to weigh these references after his/her taste.

New references are always welcome. Feel free to add an Issue or a Pull Request.

## Sections

- G: General
- WS: Whitespace
- BR: Braces
- O: Operator use
- I: Identifiers
- FU: Functions
- FI: File management
- CO: Comments
- TO: TODO’s
- PA: Package management
- NR: Non-rules

## G: General

### G.1: Use a coding standard [1]

#### References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 6 Conclusion [...] Coding style is not about making things "work", is is about making them works in a way that is understood by the widest possible audience

### G.2: Be consistent [1-3]

Whatever you do, do it consistently.

#### References

- [1] Google's R Style Guide: Use common sense and BE CONSISTENT.
- [2] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, [...] whatever style you use, please use it consistently, since a mixture of styles within one program tends to look ugly.
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Code Style': You don't have to use my style, but I strongly recommend that you use a consistent style and document it

### G.3: Use common sense [1]

Not all things are caught in a coding standard.

#### References

- [1] Google's R Style Guide: Use common sense and BE CONSISTENT.

### G.4: Prefer creating re-usable tools over cutting and pasting [1]

#### References

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, Make re-usable tools (rather than cutting and pasting\_

## G.5: Consider using `formatR` for formatting your code [1,2]

`formatR` is an R package, by Yihui Xie, that can clean up poorly formatted code. It delivers a consistent style, but it does not always work as intended by you, so check first

```
install.packages("formatR")
formatR::tidy_dir("R")
```

### References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, Try `formatR::tidy.source()`
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Code Style': The `formatR` package, by Yihui Xie, makes it easier to clean up poorly formatted code

## G.6: Consider using `lintr` for formatting your code [1]

`lintr` is an R package, by Jim Hester, that warns you when your code does not follow its preferred style guidelines, but it does not fix these.

```
install.packages("lintr")
lintr::lint_package()
```

### References

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Code Style': The `lintr` package, by Jim Hester, checks for compliance with this style guide and lets you know if you've missed something

## G.7: Prefer lines shorter than 80 characters [1-3]

[EXAMPLE HERE]

### References

- [1] Google's R Style Guide: The maximum line length is 80 characters

- [2] The GNU coding standards: Please keep the length of source lines to 79 characters or less, for maximum readability in the widest range of environments.
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Line length': Strive to limit your code to 80 characters per line

## G.8: When modifying other people's code, follow their coding standard [1-3]

[Example here]

### References

- [1] Google's R Style Guide: Use common sense and BE CONSISTENT. If you are editing code, take a few minutes to look at the code around you and determine its style. If others use spaces around their if clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them, too. The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you are saying, rather than on how you are saying it. [...]. But local style is also important. If code you add to a file looks drastically different from the existing code around it, the discontinuity will throw readers out of their rhythm when they go to read it. Try to avoid this.
- [2] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, If you are contributing changes to an existing program, please follow the style of that program.
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Code Style': If you're working on someone else's code, don't impose your own style. Instead, read their style documentation and follow it as closely as possible

## G.9: Do not put more than one command on the same line [1]

```
x <- 42; y <- 314 # Bad
```

```
# Good:
x <- 42
```

y <- 314

## References

- [1] Google's R Style Guide: Do not terminate your lines with semicolons or use semicolons to put more than one command on the same line

## G.10: Prefer using S3 over S4 [1]

[Example here]

## Exceptions

Justifications for an S4 object would be:

- to use objects directly in C++ code
- to dispatch on two arguments

## References

- [1] Google's R Style Guide: Use S3 objects and methods unless there is a strong reason to use S4 objects or methods. A primary justification for an S4 object would be to use objects directly in C++ code. A primary justification for an S4 generic/method would be to dispatch on two arguments.

## G.11: Avoid mixing S3 and S4 [1]

[Example here]

## References

- [1] Google's R Style Guide: Avoid mixing S3 and S4: S4 methods ignore S3 inheritance and vice-versa

## WS: Whitespace

### WS.1: Prefer placing spaces around all binary operators [1-3, 5]

```
x <- 42
y <- x + 314
z <- y - 42

if (a == b) {
if (a != b) {
if (a < b) {
if (a <= b) {
if (a > b) {
if (a >= b) {

sub %in% superset # Is a subset with a superset?
m %*% n # Matrix multiplication

Show <- function(s = "Hello World") { # Note the spaces around the =
  # ...
}
```

### Exception

Spaces around =s are preferred [2,3,5] or optional [1,4] when passing parameters in a function call.

```
rep(x = 314, times = 42) # [1,2,3,4,5] Good
rep(x=314, times=42) # [1,4] Okay [2,5] Bad
```

Spaces :, :: and :::s are not needed [6]:

```
x <- 3:14 # Good [6]
x <- 3 : 14 # Bad [6]
```

```
base::get # Good [6]
base :: get # Bad [6]
```

### References

- [1]Google's R Style Guide: Place spaces around all binary operators (=, +, -, <-, etc.). Exception: Spaces around ='s are optional when passing parameters in a function call.

- [2] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 5.2.1: Place spaces around all infix operators (=, +, -, <-, etc.). The same rule applies when using = in function calls. Always put a space after a comma, and never before (just like in regular English).
- [3] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.3 (SEA .98). Blank spaces around symbols are required. [...] 1. Insert spaces before and after a) mathematical symbols like "=", "<-", "<", "\*", "+" b) R binary operations like "%\*%", "%o%", and "%in%"
- [4] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, Is there an "argument exception" to the space rule for equal signs? [...] Spaces may sometimes be omitted in an effort to keep code on one line. Especially where publishers are concerned about the use of scarce paper
- [5] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Spacing': Put spacing around all infix operators (=, +, -, <-, ett.). The same rule applies when using = in function calls
- [6] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Spacing': There's a small exception to this rule: :, ::, and ::: don't need spaces around them.

## WS.2: Avoid placing spaces around code in parentheses or square brackets [1-3]

```
if (debug) # Good [1-3]
if ( debug ) # Bad [1-3]
if (debug ) # Bad [1-3]
if ( debug) # Bad [1-3]
```

```
x[1] #Good [1-3]
x[ 1 ] #Bad [1-3]
x[1 ] #Bad [1-3]
x[ 1] #Bad [1-3]
```

### Exception

Always place a space after a comma [1-3]:

```
x[1, ] #Good [1-3]
x[1,] #Bad [1-3]
```

## References

- [1] Google's R Style Guide: Do not place spaces around code in parentheses or square brackets. Exception: Always place a space after a comma.
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Spacing': Always put a space after a comma, and never before (just like in regular English)
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Spacing': Do not put spaces around code in parentheses or square brackets (unless there's a comma, in which case refer to the previous rule)

## WS.3: Use indentation to improve readability [1-3]

```
# Good [1-3]
if (x == 0) {
  message("X equals zero")
}

# Bad [3]
if (x == 0) {
message("X equals zero")
}
```

## References

- [1] Google's R Style Guide: When indenting your code, use two spaces. Never use tabs or mix tabs and spaces. Exception: When a line break occurs inside parentheses, align the wrapped line with the first character inside the parenthesis.
- [2] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.1 (SEA 1.0). Indentation of code sections is required
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Curly braces': Always indent code inside curly braces

## WS.4: Prefer indenting with a consistent amount of spaces [1,3]

The amount varies between two [1,5] and four [3] spaces.



```

if (x == 42) {
  print("Yes!") # Good [1,5] Bad [3]
}

if (x == 42) {
  print("Yes!") # Good [3] Bad [1,5]
}

```

## Exception

Extra spacing is okay if it improves readability due to the alignment of equals signs = or arrows <- [2,4].

```

plot(
  x      = xs,
  y      = ys,
  xlab = "X",
  ylab = "Y",
  main = "Title"
)

```

Example from [6]:

```

do_it <- function(a = "a long argument,
                    b = "another long argument,
                    c = "another another long argument) {
  # Business as usual with two spaces
}

```

## References

- [1] Google's R Style Guide: When indenting your code, use two spaces. Never use tabs or mix tabs and spaces. Exception: When a line break occurs inside parentheses, align the wrapped line with the first character inside the parenthesis.
- [2] Google's R Style Guide: Extra spacing (i.e., more than one space in a row) is okay if it improves alignment of equals signs or arrows (<-).
- [3] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.1 (SEA 1.0). Indentation of code sections is required
- [4] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Spacing': Extra spacing (i.e., more than one space

in a row) is OK if it improves alignment of equals signs or assignments (<-)

- [5] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Indentation': When indenting your code, use two spaces
- [6] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Indentation': The only exception is if a function definition runs over multiple lines. In that case indent the second line to where the definition starts

### WS.5: Avoid using tabs [1-3]

```
if (x == 0) {  
  # Indented by spaces, not by tabs [1-3]  
}
```

When using R studio, consider setting this in the code preferences pane [4].

#### References

- [1] Google's R Style Guide: When indenting your code, use two spaces. Never use tabs or mix tabs and spaces.
- [2] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.1 (SEA 1.0). Indentation of code sections is required. This is explicitly spelled out in the R documentation. No tabs!
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Indentation': Never use tabs or mix tabs and spaces
- [4] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Indentation': Change these options in the code preferences pane

### WS.6: Avoid placing a space before a comma [1,4]. Prefer to place a space after a comma [1-4]

```
t <- sum(x[, 1]) # Good [1-4]  
t <- sum(x[1, ]) # Good [1-4]  
t <- sum(x[,1]) # Bad [1,4]  
t <- sum(x[1,]) # Bad [1,4]
```

#### References

- [1] Google's R Style Guide: Do not place a space before a comma, but always place one after a comma.

- [2] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.3 (SEA .98). Blank spaces around symbols are required. [...] 2. Put one space after commas
- [3] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, We find it easier to read a program when it has spaces before the open-parentheses and after the commas.
- [4] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Spacing': Always put a space after a comma, and never before (just like in regular English)

**WS.7: Prefer placing a space before a left parenthesis [1,3], except when calling a function [1,3]**

```
if (debug) # Good [1-3]
if(debug)  # Bad [1,2]
print("Hello") #Good [1,3]
print ("Hello") #Bad [1]
```

#### References

- [1] Google's R Style Guide: Place a space before left parenthesis, except in a function call.
- [2] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.3 (SEA .98). Blank spaces around symbols are required. [...] For me, that settles the question. For R code, as in C, "if" and "for" should be treated as keywords, and there would be a space after them, as in "if (x < 7)"
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Spacing': Place a space before left parentheses, except in a function call

**WS.8: Avoid putting extra spaces inside parentheses [1]**

```
if (debug) # Good [1]
if ( debug ) # Bad [1]
print("Hello") #Good [1]
print( "Hello" ) #Bad [1]
```

#### References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.3 (SEA .98). Blank spaces around symbols are

```
required. [...] 3. Do not insert "extra spaces" inside
parentheses
```

### **WS.9: Prefer placing a space before the opening curly brace [1]**

```
if (debug) { # OK [1]
if (debug){ # Bad [1]
```

#### **References**

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.3 (SEA .98). Blank spaces around symbols are required. [...] 3. Insert one space before the opening squiggly braces "{"

### **WS.10: Prefer placing a space after a closing parenthesis [1] and closing squiggly brace [1]**

```
if (debug) { # OK: space after ")" [1]
# ...
} else { # OK: space after "}" [1]
# ...
}
```

#### **References**

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.3 (SEA .98). Blank spaces around symbols are required. [...] 4. Put one space after the closing parenthesis ")" and the closing squiggly brace "}"

### **WS.11: Prefer to be consistent in placing a space between a function name and its opening parenthesis [1,2]**

```
x <- c(1,2) # Good [1] Bad [2]
x <- c (1,2) # Bad [2] Good [1]
```

## References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.3 (SEA .98). Blank spaces around symbols are required. [...] 1. Do not insert spaces between function names and their opening parenthesis
- [2] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, We find it easier to read a program when it has spaces before the open-parentheses and after the commas.

## WS.12: Prefer to placing a space after an if or for statement [1,2]

```
if (x == 42) # Good [1]
if(x == 42) # Bad [1]
for (i in v) # Good [1]
for(i in v) # Bad [1]
```

## References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.3 (SEA .98). Blank spaces around symbols are required. [...] For me, that settles the question. For R code, as in C, "if" and "for" should be treated as keywords, and there would be a space after them, as in "if (x < 7)"
- [2] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, We find it easier to read a program when it has spaces before the open-parentheses and after the commas.

## WS.13: When breaking an expression into multiple lines, split it before an operator [1]

```
if (x == 42
    && y == 314) {
  # ...
}
```

## References

- [1] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, When you split an expression

into multiple lines, split it before an operator, not after one.

## BR: Braces

### BR.1: Prefer a consistent brace use [1-4]

Prefer putting an opening curly brace:

- always at the end of a line [1,4]
- at the end of a line, when not starting a function [2]

Prefer to put a closing curly brace on its own line [1,2,4], except when followed by else [5].

```
# Good [1,4] Bad [2]
f <- function(x) {
  # ...
}
```

```
# Good [2] Bad [1,4]
f <- function(x)
{
  # ...
}
```

```
# [1,2,4] Good
if (x == 42) {
  # ...
}
```

```
# [5] Bad
if (x == 42) {
  # ... }
```

```
# [5] Good
if (x == 42) {
  # ...
} else {
  # ...
}
```

```
# [1,2,4] Bad
```

```

if (x == 42)
{
  # ...
}

# [1,2,4] Bad
if (x == 42)
{
  # ...
}

```

## References

- [1] Google's R Style Guide: An opening curly brace should never go on its own line; a closing curly brace should always go on its own line.
- [2] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, It is important to put the open-brace that starts the body of a C function in column one
- [3] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Avoid putting open-brace, open-parenthesis or open-bracket in column one when they are inside a function
- [4] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Curly braces': An opening curly brace should never go on its own line and should always be followed by a new line
- [5] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Curly braces': A closing curly brace should always go on its own line, unless it's followed by else

## BR.2: Be consistent in ommitting curly braces when they can be ommitted [1]

You may omit curly braces when a block consists of a single statement:

```

# Good: keeping the curly braces [1]
if (x == 42) {
  print("OK")
}

if (y == 314) {
  print("OK")
}

```

```

}

# Good: omitting the curly braces [1]
if (x == 42)
  print("OK")

if (y == 314)
  print("OK")

# Good: omitting the curly braces and putting the statements on the same line[2]
if (x == 42) print("OK")

if (y == 314) print("OK")

# Bad: inconsistency in omitting the curly braces [1]
if (x == 42)
  print("OK")

if (y == 314) {
  print("OK")
}

```

## References

- [1] Google's R Style Guide: You may omit curly braces when a block consists of a single statement; however, you must consistently either use or not use curly braces for single statement blocks.
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Curly braces': It's OK to leave very short statements on the same line

## BR.3: Prefer to begin the body of a block on a new line [1,2]

```

# Good [1,2]
if (x == 42) {
  print("OK")
}

# Bad [1,2]
if (x == 42)
{
  print("Bad")
}

```



```

}

# Bad?
if (x == 42) { print("Bad") }

```

## References

- [1] Google's R Style Guide: Always begin the body of a block on a new line.
- [2] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Avoid putting open-brace, open-parenthesis or open-bracket in column one when they are inside a function

## BR.4: Prefer to surround `else` with braces [1,2]

```

# OK [1-3]
if (x == 42) {
  # ...
} else {
  # ...
}

```

```

# Bad [1-3]
if (x == 42) {
  # ...
}
else {
  # ...
}

```

```

# Bad [1-3]
if (x == 42)
  # ...
else
  # ...

```

## References

- [1] Google's R Style Guide: Surround `else` with braces. An `else` statement should always be surrounded on the same line by curly braces.

- [2] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.4 (SEA .70) The "}" else {" policy
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Curly braces': A closing curly brace should always go on its own line, unless it's followed by else

## O: Operator use

### O.1: Prefer using <- over = for assignment [1-3]

```
x <- 42 # Good [1-3]
x = 42 # Bad [1-3]
```

#### References

- [1] Google's R Style Guide: Use <-, not =, for assignment
- [2] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 3.2 (SEA .95). Use "<-" not "=" for assignments
- [3] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Assignment': Use <-, not =, for assignments

### O.2: Avoid using semicolons ; [1]

```
x <- 42 # Good
x <- 42; # Bad
```

#### References

- [1] Google's R Style Guide: Do not terminate your lines with semicolons or use semicolons to put more than one command on the same line

### O.3: Prefer using && over & in an if-statement [1]

```
if (x == 42 && y == 42) {
  print("OK")
}
```

According to [3], & is a logical and (page 19), and && is described as an AND with IF.

The C and C++ languages call `&` a bitwise AND, where `&&` is called a logical AND. In those languages, `&&` is preferred as it invokes a short-circuit evaluation: `if (a && b)` will terminate directly if `a` is false, where in `if (a & b)` both `a` and `b` are evaluated even if `a` is false:

```
say_hello <- function()
{
  print("Hello")
  return (1)
}

# say_hello should not be called
# operator& evaluates all arguments, even if the first one is false
# operator&& evaluates the arguments until it finds a false (and thus can never be true any
if (1 + 1 == 3 & say_hello()) { }
if (1 + 1 == 3 && say_hello()) { }

# Also for more arguments
if (1 + 1 == 3 & 1 + 1 == 4 & say_hello()) { }
if (1 + 1 == 3 && 1 + 1 == 4 && say_hello()) { }
```

Note that [3] argues the contrary of this advice and even wonders on the use of `&&` for doing conditional *vector* operations, which is something else this advice is about.

## References

- [1] Style Guide from Hadley Wickham, <http://r-pkgs.had.co.nz/style.html>
- [2] The R Book, 2nd Edition, Michael j. Crawley
- [3] Blog from csgillespie: <http://www.r-bloggers.com/logical-operators-in-r/>

## I: Identifiers

### I.1: Prefer English variable names [1]

#### References

- [1] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, In a GNU program, names should be English, like other comments

## I.1: Strive for names that are concise and meaningful [1]

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Object names': Strive for names that are concise and meaningful (this is not easy!)

## I.2: Prefer to name non-constant variables in a consistent way [1-6]

```
my_value # Good [3,4,5,6] Bad [2]
my.value # Good [1] Bad [3,4,6]
myVlicks # Okay [1,5,7] Bad [3,4]
my-value # Bad [2,3,4,6]
```

## References

- [1] Google's R Style Guide: Identifiers should be named according to the following conventions. The preferred form for variable names is all lower case letters and words separated with dots (variable.name), but variableName is also accepted.
- [2] Google's R Style Guide: Don't use underscores ( \_ ) or hyphens ( - ) in identifiers
- [3] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 5.1.2: Variables and function names should be lowercase. Use an underscore to separate words within a name. Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful
- [4] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Please use underscores to separate words in a name [...] For example, you should use names like ignore\_space\_change\_flag; don't use names like iCantReadThis.
- [5] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Object names': Variable and function names should be lowercase
- [6] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Object names': Use an underscore to separate words within a name (reserve . for S3 methods)
- [7] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Object names': Camel case is a legitimate alternative, but be consistent!

### I.3: Prefer a variable being a noun [1,2]

```
variance # Good
working  # Bad
```

#### References

- [1] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 5.1.2: Variables and function names should be lowercase. Use an underscore to separate words within a name. Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Object names': Generally, variable names should be nouns and function names should be verbs

### I.4: Avoid naming a variable T of F [1]

Because T and F are also shorthands for TRUE and FALSE.

```
T <- 42 # Avoid
F <- 42 # Avoid
```

#### References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 5.2 (1.0 SEA) Never name a variabel T or F

### I.5: Consider variable name length to be proportional to their locality [1]

Long-living variables are preferred to be long. Short-living variables may be short, presumably with a comment that explain their purpose

#### References

- [1] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Local variable names can be shorter, because they are used only within one context, where (presumably) comments explain their purpose.

## I.6: Consider making variable name length inversely proportional to their use [1]

Infrequently used variables are preferred to be long. Frequently used variables may be short.

### References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 5.4 (0.50 SEA) Use long names for infrequently used variables. And use short names for variables that will be used often

## I.7: Consider limiting abbreviations [1]

```
n_species <- 42 # Number of species
avg_n_species <- 42 # Average number of species
```

### References

- [1] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Try to limit your use of abbreviations in symbol names. It is ok to make a few abbreviations, explain what they mean, and then use them frequently, but don't use lots of obscure abbreviations.
- [2] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 5.4 (0.50 SEA) [...] For abbreviations, include a comment to remind the reader what the thing stands for

## I.8: Avoid declaring variables that have the same names as widely used functions [1,2]

```
T <- false # Avoid [2]
c <- 42 # Avoid [1,2]
rep <- 314 # Avoid [1]
mean <- function(x) { sum(x) } # Avoid [2]
```

## References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 5.3 (.75 SEA) Avoid declaring variables that have the same names as widely used functions
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Object names': Where possible, avoid using names of existing functions and variables

### I.9: consider naming constant variables differently [1]

This idea is suggested by [1], it is unknown what [2] thinks about this.

When embracing the idea, I add the naming schemes for both [1] and [2]:

```
kPi <- 3.14 # Good [1] Bad [2]
k_pi <- 3.14 # Bad [1]
```

## References

- [1] Google's R Style Guide: constants are named like functions but with an initial k
- [2] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 5.1.2: Variables and function names should be lowercase. Use an underscore to seperate words within a name. Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful

### I.10: Prefer argument names that are clear, unambiguous, convenient and short [1]

## References

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, 4. Function arguments. [...] To state the obvious, argument names should be clear, unambiguous, convenient, short and minimally necessary

### I.11: Prefer lowercase function argument names [1]

## References

- [1] The GNU coding standards, by Free Software Foundation, Inc.,

<http://www.gnu.org/prep/standards>,      The variable name itself  
should be lower case

## I.12: Prefer naming functions in a consistent way [1]

Either:

- CamelCase (first letter upper case) [1]
- camelCase (first letter lower case) [3]
- lower\_case\_with\_underscores [2,4]

```
CalculateVariance # Good [1] Bad [2,3,4]
calculateVariance # Good [3,5] Bad [1,2,4]
calculate_variance # Good [2,4,5] Bad [1,3]
calculate.variance # Bad [1,2,3]
```

## References

- [1] Google's R Style Guide: function names have initial capital letters and no dots
- [2] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 5.1.2: Variables and function names should be lowercase. Use an underscore to separate words within a name. Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful
- [3] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 4.2 (.65 SEA) Use periods to indicate classes, otherwise don't use periods in function names. Instead, use camel case to name functions
- [4] [C++ Core Guidelines](<https://github.com/isocpp/CppCoreGuidelines>). NL.10: Avoid CamelCase
- [5] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Object names': Variable and function names should be lowercase

## I.13: Prefer the first word of a function being a verb [1,3]

```
MeasureSpeed # Good [1], but should should start with lowercase [3,4]
measure_speed # Good [3,4], but should should start with uppercase [1]
Speed        # Bad [1,3,4]
speed        # Bad [1,3,4]
```



## Exception

When creating a class object, the function name (constructor) and class name should match [2]

## References

- [1] Google's R Style Guide: Make function names verbs
- [2] Google's R Style Guide: When creating a class object, the function name (constructor) and class name should match
- [3] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 5.1.2: Variables and function names should be lowercase. Use an underscore to separate words within a name. Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful
- [4] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Object names': Generally, variable names should be nouns and function names should be verbs

## FU: Functions

### FU.1: Prefer short functions [1]

#### References

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, 6 Suggested chores [...] Functions should not be HUGE

### FU.2: Avoid naming function the same as those already present in R [1]

```
c <- function(x) #Bad, c is already a common R function
Concatenate <- function(x) #Good, this function name does not exist
```

#### References

- [1] 'R Style. An Rchaeological Commentary.' by Paul E. Johnson, February 13, 2015, 4.1 (SEA .98) Avoid using names that are already in R, especially common ones

### **FU.3: Prefer to start function definition argument lists with the non-defaultable values [1]**

```
# Good
Transmogrify <- function(
  foo,
  bar = 42
)
```

```
# Bad
Transmogrify <- function(
  foo = 314,
  bar
)
```

#### **References**

- Google's R Style Guide: Function definitions should first list arguments without default values, followed by those with default values.

### **FU.4: Check argument values [1]**

#### **References**

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, 4. Function arguments. [...] 2. Check argument values

### **FU.5: Prefer to optionally use line breaks in function definitions between assignments only [1]**

```
# Good
Transmogrify <- function(
  foo = 314,
  bar = 42
)
```

```
# Good
Transmogrify <- function(foo = 314, bar = 42)
```

```
# Bad
Transmogrify <- function(foo
  = 314, bar = 42)
```

## References

- Google's R Style Guide: In both function definitions and function calls, multiple arguments per line are allowed; line breaks are only allowed between assignments.

## FU.6: Prefer to optionally use line breaks in function calls between assignments only [1]

```
# Good
Transmogrify(
  foo = 314,
  bar = 42
)

# Good
Transmogrify(foo = 314, bar = 42)

# Bad
Transmogrify(foo
  = 314, bar = 42)
```

## References

- Google's R Style Guide: In both function definitions and function calls, multiple arguments per line are allowed; line breaks are only allowed between assignments.

## FU.7: Protect your function's calculations from the user's workspace [1]

## References

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, 4. Function arguments. [...] 1. Protect your function's calculations from the user's workspace

## **FU.8: Design the function so that it runs with a minimum number of arguments [1]**

### **References**

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, 4. Function arguments. [...] 3. Design the function so that it runs with a minimum number of arguments

## **FU.9: Prefer to specify default function arguments to reduce the numbers of function arguments needed [1]**

### **References**

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, 4. Function arguments. [...] 4.4.1 Specify defaults

## **FU.10: Prefer extracting or constructing information from the function arguments over requiring another function argument [1]**

### **References**

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, 4. Function arguments. [...] 4.4.2. Extract or construct what else is needed from the user input

## **FU.11: For loops are not necessarily bad [1]**

### **References**

- [1] 'Rchaeology: Idioms of R Programming' by Paul E. Johnson, January 28, 2015, 5 Do this, not that [...] One can write an lapply statement in one line, while a for loop can take 3 lines. The code is shorter, but won't necessarily run more quickly

## **FU.12: Prefer using `typeof()` over `mode()` [1]**

`mode()` is an alias of `typeof()` that only exists for S compatibility.

```
typeof(x) # Good [1]  
mode(x)  # Bad [1]
```

## References

- [1] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 7.1, page 102: You may have heard of `mode()` and `storage.mode()`. I recommend ignoring these functions because they are just aliases of the names returned by `typeof()`, and exist solely for S compatibility

## FU.13: Prefer using `typeof()` over `storage.mode()` [1]

`storage.mode()` is an alias of `typeof()` that only exists for S compatibility.

```
typeof(x) # Good [1]
storage.mode(x) # Bad [1]
```

## References

- [1] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 7.1, page 102: You may have heard of `mode()` and `storage.mode()`. I recommend ignoring these functions because they are just aliases of the names returned by `typeof()`, and exist solely for S compatibility

## FU.14: Avoid using `attach` [1]

[Example here]

## References

- [1] Google's R Style Guide: The possibilities for creating errors when using `attach` are numerous. Avoid it.

## FI: File

### FI.1: Prefer meaningful files names ending in `.R` [1]

- Good filenames: `predict_ad_revenue.R`, `fit-models.R`
- Bad filenames: `predict_ad_revenue.r`, `foo.R`, `utility-functions.R`

## References

- Google's R Style Guide: File names should end in `.R` and, of course, be meaningful
- Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 5.1.1: Filenames should be meaningful and end in `.R`

## FI.2: If files need to be run in sequence, consider prefixing them with numbers [1]

```
0-download.R
1-parse.R
2-explore.R
```

## References

- [1] Wickham, Hadley. Advanced R. CRC Press, 2014. Chapter 5.1.1: If files need to be run in sequence, prefix them with numbers

## FI.3: Prefer naming unit test files ending in `_test.R` [1]

- Good filenames: `predict_ad_revenue_test.R`
- Bad filames: `test_predict_ad_revenue.R`

## References

- [1] Google's R Style Guide: Unit tests should go in a separate file named `originalfilename_test.R`.

## FI.4: Be consistent in the ordering of a file [1]

An example ordering: `* Copyright statement comment` `* Author comment`  
`* File description comment, including purpose of program, inputs, and out-`  
`puts` `* source() and library() statements` `* Function definitions` `* Executed`  
`statements, if applicable, for example print and plot`

## References

- [1] Google's R Style Guide: If everyone uses the same general ordering, we'll be able to read and understand each other's scripts faster and more easily.

## CO: Comments

### CO.1: Prefer to add comments to your code [1,2]

```
# Determine if expensive calculation can be avoided
```

#### References

- [1] Google's R Style Guide: `Comment your code`
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Commenting guidelines': `Comment your code`

### CO.2: Prefer to add comments in English [1]

#### References

- [1] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Please write the comments in a GNU program in English, because English is the one language that nearly all programmers in all countries can read.

### CO.3: Prefer to use complete sentences that start with a capital [1]

```
# Calculate the variance after extracting all values from the matrices
```

#### Exceptions

When the sentence starts with a lower-case identifier [1]

```
# i is used to calculate the variance
```

But this can also be rewritten to:

```
# To calculate the variance, i is used
```

## References

- [1] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Also, please write complete sentences and capitalize the first word. If a lower-case identifier comes at the beginning of a sentence, don't capitalize it! Changing the spelling makes it a different identifier. If you don't like starting a sentence with a lower case letter, write the sentence differently (e.g., "The identifier lower-case is ...").

## CO.4: Comments should explain the why, not the what

```
# Good
x <- c(1,2) # omit values that cannot be used for primality testing

# Bad
x <- c(1,2) # combine a one and a two and store it in x
```

## References

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Commenting guidelines': Comments should explain the why, not the what

## CO.5: Consider starting a file with a comment briefly stating what it is for [1]

## References

- [1] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Every program should start with a comment saying briefly what it is for

## CO.6: Prefer to start an entire-line-comment with # and one space [1,2]

```
# Determine if expensive calculation can be avoided
```



## References

- [1] Google's R Style Guide: Entire commented lines should begin with # and one space.
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Commenting guidelines': Each line of a comment should begin with the comment symbol and a single space

**CO.7: For a comment directly after some code, add two spaces, # and then one space [1,2]**

```
if (x == 42) { # Are we lucky?
  # ... # Good [1,2]
}
```

## References

- [1] Google's R Style Guide: Short comments can be placed after code preceded by two spaces, #, and then one space.
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Commenting guidelines': Each line of a comment should begin with the comment symbol and a single space

**CO.8: Prefer to describe a function in comments in the line(s) directly below the function definition [1,2]**

These comments may consist of:

- a one-sentence description of the function
- a list of the function's arguments, denoted by Args:, with a description of each (including the data type)
- and a description of the return value, denoted by Returns:

The comments should be descriptive enough that a caller can use the function without reading any of the function's code.

```
SumFloats <- function(x) {
  # Calculates the sum of a vector of floats
  # Args: x: a vector containing zero or more floating point values
  # Returns: the sum of the vector as a floating point
}
```

## References

- [1] Google's R Style Guide: Functions should contain a comments section immediately below the function definition line. These comments should consist of a one-sentence description of the function; a list of the function's arguments, denoted by `Args:`, with a description of each (including the data type); and a description of the return value, denoted by `Returns:`. The comments should be descriptive enough that a caller can use the function without reading any of the function's code.
- [2] The GNU coding standards, by Free Software Foundation, Inc., <http://www.gnu.org/prep/standards>, Please put a comment on each function saying what the function does, what sorts of arguments it gets, and what the possible values of arguments mean and are used for. [...] Also explain the significance of the return value, if there is one.

## CO.9: Use commented lines of - and = to break up files in easily readable chunks [1]

```
# Load data -----  
  
# Plot data =====
```

## References

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Commenting guidelines': Use commented lines of - and = to break up files in easily readable chunks

## ER: Error handling

### ER.1: Errors should be raised using stop() [1]

```
if (x < 0) stop()
```

## References

- [1] Google's R Style Guide: Errors should be raised using stop()

## TE: Testing

### TE.1: Prefer to put unit tests in seperate files [1]

See also ‘prefer naming unit test files ending in `_test.R`’

#### References

- [1] Google’s R Style Guide: Unit tests should go in a separate file named `originalfilename_test.R`.

### TE.2: Prefer to let your unit tests serve as example function calls [1]

[Example]

#### References

- [1] Google’s R Style Guide: Ideally, unit tests should serve as sample function calls (for shared library routines).

## TO: TODO’s

### TO.1: Prefer to use a consistent style for TODOs [1]

For example:

```
TODO([username]): [Explicit description of action to be taken]
```

Like:

```
TODO(richelbilderbeek): Check if this really works
```

#### References

- [1] Google’s R Style Guide: Use a consistent style for TODOs throughout your code. `TODO(username):` Explicit description of action to be taken

## PA: Package management

These guidelines apply only when writing a package.

### PA.1: Never run code at the top-level of a package

Never run code at the top-level of a package [1].

#### References

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Top-level code': This means you should never run code at the top-level of a package: package code should only create objects, mostly functions

### PA.2: Don't use `library()` or `require()` [1,2]

Add the needed package to the DESCRIPTION file, instead of using `library` or `require` within a function.

```
f <- my_function() {  
  library(some_package) # Bad [1,2], add 'some_package' to the DESCRIPTION file  
}  
  
g <- my_function() {  
  require(some_package) # Bad [1,2], add 'some_package' to the DESCRIPTION file  
}
```

#### References

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'Top-level code': This means you should never run code at the top-level of a package: package code should only create objects, mostly functions
- [2] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'The R landscape': Don't use `library()` or `require()`

### PA.3: Never use `source()` to load code from a file [1]

Rely on `devtools::load_all()`, which automatically sources all files in R/. If you are using `source()` to create a dataset, instead switch to `data/`

```
f <- my_function() {
  source("my_file.R") # Bad [1]
}
```

## References

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'The R landscape': Never use `source()` to load code from a file

## PA.4: If you modify global options() or graphics par(), save the old values and reset when done [1]

Example from [1]:

```
old <- options(stringsAsFactors = FALSE)
on.exit(options(old), add = TRUE)
```

## References

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'The R landscape': If you modify `global options()` or `graphics pa()`, save the old values and reset when you're done

## PA.5: Avoid modifying the working directory [1]

Example from [1]:

```
old <- setwd(tempdir())
on.exit(setwd(old), add = TRUE)
```

## References

- [1] 'R packages' by Hadley Wickham, O'Reilly Media, Inc., 2015, Chapter 3, paragraph 'The R landscape': Avoid modifying the working directory

## Non-rules

### **NR.1: Every file should be self-sufficient** [1]

A file that uses libraries and source files, should call `library` and `source` to add these. Otherwise, the caller of the function gets the responsibility of doing this him/herself.

This is a non-rule however, see the PA section.

[1] Suggested by Richel Bilderbeek