查看: 91 | 回复: 7

[程序分享] 来自 Google 的 R 语言编码风格指南 [复制链接]

famousid

加关注

发表于 昨天 13:50 | 只看该作者 | 倒序浏览 分享到： 楼主 电梯直达

来自 **Google** 的 **R** 语言编码风格指南R 语言是一门主要用于统计计算和绘图的高级编程语言. 这份 R 语言编码风格指南旨在让我们的 R 代码更容易阅读、分享和检查. 以下规则系与 Google 的 R 用户群体协同设计而成.

- **概要: R**编码风格约定
  1. 文件命名: 以 .R (大写) 结尾
  2. 标识符命名: variable.name, FunctionName, kConstantName

0
关注
4
粉丝
大专生
论坛币 2838012
经验 1762 点
威望 0 级
帖子 69
精华 0
学术水平 37 点
热心指数 39 点
信用等级 27 点
在线时间 18 小时
注册时间 2009-12-9
最后登录 2012-4-7
串个门 加好友
打招呼 发消息

3. 单行长度: 不超过 80 个字符
4. 缩进: 两个空格, 不使用制表符
5. 空白
6. 花括号: 前括号不折行写, 后括号独占一行
7. 赋值符号: 使用 <-, 而非 =
8. 分号: 不要用
9. 总体布局和顺序
10. 注释准则: 所有注释以 # 开始, 后接一个空格; 行内注释需要在 # 前加两个空格
11. 函数的定义和调用
12. 函数文档
13. 示例函数
14. TODO 书写风格: TODO(您的用户名)

- **概要: R**语言使用规则
  1. attach: 避免使用
  2. 函数: 错误 (error) 应当使用 stop() 抛出
  3. 对象和方法: 尽可能避免使用 S4 对象和方法; 永远不要混用 S3 和 S4

1. 表示和命名

   1. 文件命名
      文件名应以 .R (大写) 结尾, 文件名本身要有意义.
      正例: predict_ad_revenue.R
      反例: foo.R
   2. 标识符命名
      在标识符中不要使用下划线 (_) 或连字符 (-). 标识符应根据如下惯例命名. 变量名应使用点 (.) 分隔所有的小写字母或单词; 函数名首字母大写, 不用点分隔 (所含单词首字母大写); 常数命名规则同函数, 但需使用一个 k 开头.
      1. variable.name
         正例: avg.clicks
         反例: avg_Clicks , avgClicks
      2. FunctionName
         正例: CalculateAvgClicks

反例: calculate_avg_clicks , calculateAvgClicks
函数命名应为动词或动词性短语.
例外: 当创建一个含类 (class) 属性的对象时, 函数名 (也是constructor) 和类名 (class) 应当匹配 (例如, lm).

3. kConstantName

2. 语法

1. 单行长度
   最大单行长度为 80 个字符.
2. 缩进
   使用两个空格来缩进代码. 永远不要使用制表符或混合使用二者.
   例外: 当括号内发生折行时, 所折行与括号内的第一个字符对齐.

3. 空白
   在所有二元操作符 (=, +, -, <-, 等等) 的两侧加上空格.
   例外: 在函数调用中传递参数时 = 两边的空格可加可不加.
   不可在逗号前加空格, 逗号后总须加空格.

   正例:
   tabPrior <- table(df[df$daysFromOpt < 0, "campaignid"])total <- sum(x[, 1])total <- sum(x[1,])
   反例:
   tabPrior <- table(df[df$daysFromOpt<0, "campaignid"]) # 在 '<' 两侧需要增加空格tabPrior <- table(df[df$daysFromOpt < 0,"campaignid"]) # 逗号后需要一个空格tabPrior<- table(df[df$daysFromOpt < 0, "campaignid"]) # 在 <- 前需要一个空格tabPrior<-table(df[df$daysFromOpt < 0, "campaignid"]) # 在 <- 两侧需要增加空格total <- sum(x[,1]) # 逗号后需要一个空格total <- sum(x[,1]) # 逗号后需要一个空格, 而非逗号之前
   在前括号前加一个空格, 函数调用时除外.
   正例:
   if (debug)
   反例:
   if(debug)
   多加空格 (即, 在行内使用多于一个空格) 也是可以的, 如果这样做能够改善等号或箭头 (<-) 的对齐效果.
   plot(x = xCoord, y = dataMat[, makeColName(metric, ptiles[1], "roiOpt")], ylim = ylim, xlab =

"dates", ylab = metric, main = (paste(metric, " for 3 samples ", sep=""))))不要向圆括号或方括号中的代码两侧加入空格.

例外: 逗号后总须加空格.

正例:

```
if (debug)x[1, ]
```

反例:

```
if ( debug ) # debug 的两边不要加空格x[1,] # 需要在逗号后加一个空格
```

4. 花括号

前括号永远不应该独占一行; 后括号应当总是独占一行. 您可以在代码块只含单个语句时省略花括号; 但在处理这类单个语句时, 您必须 *前后一致地* 要么全部使用花括号, 或者全部不用花括号.

```
if (is.null(ylim)) {  ylim <- c(0, 0.06)}或 (不可混用)
if (is.null(ylim))  ylim <- c(0, 0.06)总在新起的一行开始书写代码块的主体.
```

反例:

```
if (is.null(ylim)) ylim <- c(0, 0.06)
if (is.null(ylim)) {ylim <- c(0, 0.06)}
```

5. 赋值

使用 <- 进行赋值, 不用 = 赋值.

正例:

```
x <- 5
```

反例:

```
x = 5
```

6. 分号

不要以分号结束一行, 也不要利用分号在同一行放多于一个命令. (分号是毫无必要的, 并且为了与其他Google编码风格指南保持一致, 此处同样略去.)

3. 代码组织

1. 总体布局和顺序

如果所有人都以相同顺序安排代码内容, 我们就可以更加轻松快速地阅读并理解他人的脚本了.

1. 版权声明注释
2. 作者信息注释
3. 文件描述注释, 包括程序的用途, 输入和输出
4. source() 和 library() 语句
5. 函数定义

6. 要执行的语句, 如果有的话 (例如, print, plot)

单元测试应在另一个名为 原始的文件名_unittest.R 的独立文件中进行.

2. 注释准则

注释您的代码. 整行注释应以 # 后接一个空格开始.

行内短注释应在代码后接两个空格, #, 再接一个空格.

Create histogram of frequency of campaigns by pct budget spent.hist(df$pctSpent, breaks = "scott", # method for choosing number of buckets main = "Histogram: fraction budget spent by campaignid", xlab = "Fraction of budget spent", ylab = "Frequency (count of campaignids)")

3. 函数的定义和调用

函数定义应首先列出无默认值的参数, 然后再列出有默认值的参数.

函数定义和函数调用中, 允许每行写多个参数; 折行只允许在赋值语句外进行.

正例:

PredictCTR <- function(query, property, numDays, showPlot = TRUE)反例:PredictCTR <- function(query, property, numDays, showPlot = TRUE)理想情况下, 单元测试应该充当函数调用的样例 (对于包中的程序来说).

4. 函数文档

函数在定义行下方都应当紧接一个注释区. 这些注释应当由如下内容组成: 此函数的一句话描述; 此函数的参数列表, 用 Args: 表示, 对每个参数的描述 (包括数据类型); 以及对于返回值的描述, 以 Returns: 表示. 这些注释应当描述得足够充分, 这样调用者无须阅读函数中的任何代码即可使用此函数.

5. 示例函数

CalculateSampleCovariance <- function(x, y, verbose = TRUE) { # Computes the sample covariance between two vectors. # # Args: # x: One of two vectors whose sample covariance is to be calculated. # y: The other vector. x and y must have the same length, greater than one, # with no missing values. # verbose: If TRUE, prints sample covariance; if not, not. Default is TRUE. # # Returns: # The sample covariance between x and y. n <- length(x) # Error handling if (n <= 1 || n != length(y)) { stop("Arguments x and y have invalid lengths: ", length(x), " and ", length(y), ".") } if (TRUE %in% is.na(x) || TRUE %in% is.na(y)) { stop(" Arguments x and y must not have missing values.") } covariance <- var(x, y) if (verbose) cat("Covariance = ", round(covariance, 4), ".\n", sep = "") return(covariance)}

6. **TODO** 书写风格

编码时通篇使用一种一致的风格来书写 TODO.

TODO(您的用户名): 所要采取行动的明确描述

4. 语言

    1. **Attach**
       使用 attach 造成错误的可能数不胜数. 避免使用它.
    2. 函数
       错误 (error) 应当使用 stop() 抛出.
    3. 对象和方法
       S 语言中有两套面向对象系统, S3 和 S4, 在 R 中这两套均可使用. S3 方法的可交互性更强, 更加灵活, 反之, S4 方法更加正式和严格. (对这两套系统的说明, 参见 Thomas Lumley 的文章 "Programmer's Niche: A Simple Class, in S3 and S4", 发表于 R News 4/1, 2004, 33 - 36 页:http://cran.r-project.org/doc/Rnews/Rnews_2004-1.pdf.)
       这里推荐使用 S3 对象和方法, 除非您有很强烈的理由去使用 S4 对象和方法. 使用 S4 对象的一个主要理由是在 C++ 代码中直接使用对象. 使用一个 S4 泛型/方法的主要理由是对双参数的分发.
       避免混用 S3 和 S4: S4 方法会忽略 S3 中的继承, 反之亦然.

5. 例外
   除非有不去这样做的好理由, 否则应当遵循以上描述的编码惯例. 例外包括遗留代码的维护和对第三方代码的修改.
6. 结语
   遵守常识, 前后一致.如果您在编辑现有代码, 花几分钟看看代码的上下文并弄清它的风格. 如果其他人在 if 语句周围使用了空格, 那您也应该这样做. 如果他们的注释是用星号组成的小盒子围起来的, 那您也要这样写。
   遵循编码风格准则的意义在于, 人们相当于有了一个编程的通用词汇表, 于是人们可以专注于您在 *说什么* , 而不是您是 *怎么说* 的. 我们在这里提供全局的编码风格规则以便人们了解这些词汇, 但局部风格也很重要. 如果您加入文件中的代码看起来和周围的已有代码截然不同, 那么代码阅读者的阅读节奏就会被破坏. 尽量避免这样做. OK, 关于如何写代码已经写得够多了; 代码本身要有趣的多. 编码愉快!
7. 参考文献
   http://www.maths.lth.se/help/R/RCC/ - R语言编码惯例
   http://ess.r-project.org/ - 为 emacs 用户而生. 在您的 emacs 中运行 R 并且提供了一个 emacs mode.

已有 3 人评分	经验	学术水平	热心指数	信用等级	理由	收起
ryusukekenji		+ 1	+ 1		鼓励积极发帖讨论	
kk22boy		+ 5	+ 5	+ 5	精彩帖子	
qoiqpwqr	+ 40		+ 1		鼓励积极发帖讨论	

总评分: 经验 + 40 学术水平 + 6 热心指数 + 7 信用等级 + 5 查看全部评分

offandon

发表于 昨天 14:06 | 只看该作者 沙发

什么。。。。。。

加关注

1	1
关注	粉丝

VIP3

论坛币	10249 个
经验	53045 点
威望	0 级
帖子	1603
精华	0
学术水平	6 点
热心指数	13 点
信用等级	3 点
在线时间	992 小时
注册时间	2008-6-17
最后登录	2012-4-7

串个门 加好友

打招呼 发消息

举报

jiao_taishan

发表于 昨天 20:25 | 只看该作者

藤椅

加关注

22	3
关注	粉丝

副教授

论坛币	1971 个
经验	13147 点
威望	0 级
帖子	328
精华	0
学术水平	2 点
热心指数	5 点
信用等级	2 点
在线时间	1363 小时
注册时间	2009-9-21
最后登录	2012-4-7

串个门 加好友

打招呼 发消息

很感兴趣~~楼主多介绍一下呗

SIGNATURE

常用统计数据来源：中国统计月报，中国经济景气月报

举报

qoiqpwqr

发表于 昨天 21:39 | 只看该作者

板凳

加关注

0	48
关注	粉丝

版主

论坛币	53319 个
经验	65287 点
威望	1 级
帖子	1606
精华	1
学术水平	353 点
热心指数	432 点
信用等级	285 点
在线时间	1292 小时
注册时间	2009-7-18
最后登录	2012-4-7

串个门 加好友

打招呼 发消息

个人喜好，一致就好。

举报

kk22boy

发表于 昨天 22:30 | 只看该作者

报纸

R Coding Conventions (RCC)
- a draft

Version 0.9, January 2009
(since 2002)

Henrik Bengtsson

Dept of Statistics, University of California, Berkeley

Table of Content

References

Introduction

Please note that this document is under construction since mid October 2002 and should still be seen as a first rought draft. There is no well defined coding recommendations for the R language [1] and neither is there a de facto standard. This document will give some recommendations, which are very similar to the ones in the Java programming style [2][3], which have found to be helpful for both the developer as well as the end user of packages and functions written in R.

Layout of the Recommendations

The recommendations are grouped by topic and each recommendation is numbered to make it easier to refer to during reviews. Layout for the recommendations is as follows:

Guideline short description
Example if applicable
Motivation, background and additional information.

The motivation section is important. Coding standards and guidelines tend to start "religious wars", and it is important to state the background for the recommendation.

Recommendation Importance

In the guideline sections the terms must, should and can have special meaning. A must requirement must be followed, a should is a strong recommendation, and a can is a general guideline.

General Recommendations

Any violation to the guide is allowed if it enhances readability.

The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer

should be flexible.

Naming Conventions

General Naming Conventions

Names representing classes must be nouns and written in mixed case starting with upper case ("CamelCase").
Line
FilePrefix # NOT: File.Prefix
Even if it is legal to have . (period) in a class name, it is highly recommended not to have it, since declaration of S3 methods are separating the method name from the class name where a . (period) occurs, cf. the following ambigous method definition:

```
a.b.c <- function(x) {
  :
}
```

Is the method above meant to be method a.b() for class c or method a() for class b.c?

Variable (field, attribute) names must be in mixed case starting with lower case ("camelCase").
line
filePrefix # NOT: file.prefix
Makes variables easy to distinguish from types, e.g. Line vs line.
Avoid using . (period) in variable names to make names more consistent with other naming conventions.

Names representing constants must be all uppercase using period to separate words.
MAX.ITERATIONS, COLOR.RED
Since the R language does not support (final) constants, but regular variables must be used, it is up to the programmer to make sure that such variables keeps the same value throughout its life time. This rule will help the programmer to identify which variables can be modified and which can be not.
Note that this does not follow the general suggestions of avoiding . (period) in names. In other languages, it is common to use _, but since this was previously used as a "shortcut" for assignment in R, we choose not to use this

in order to avoid problems. In the future, we might update this guideline to make use of _ instead/also.

Names representing methods (functions) must be verbs and written in mixed case starting with lower case ("camelCase").

```
getName()                # NOT:  get.name()
computeTotalWidth()      # NOT:  compute.total.width()
```

This is identical to variable names, but methods in R are already distinguishable from variables by their specific form.

Do not use . (period) in the method name as it is ambigous in the context of object oriented code, cf. "Names representing classes must be nouns and written in mixed case starting with upper case." above.

Names representing arguments should be in mixed case starting with lower case.

```
normalizeScale <- function(x, newSd=1) {
  :
}
```

For backward compatibility with historical functions, it is alright to also use . for separating words, e.g.

```
normalizeScale <- function(x, new.sd=1) {
  :
}
```

Names representing constructors should be identical to the class name.

```
Line <- function(x0, y0, x1, y1) {
  line <- list(x=c(x0,y0), y=(x1,y1));
  class(line) <- "Line";
  line;
}
```

This makes it easy to remember the name of a function for creating a new instance of a class. It also makes the constructor to stand out from the methods.

Abbreviations and acronyms should not be uppercase when used as name.

```
exportHtmlSource();  # NOT: exporthtmlSource();
openDvdPlayer();     # NOT: openDVDPlayer();
```

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type whould have to be named dVD, hTML etc. which obviously is not very readable. Another problem is illustrated in the examples above; When the name is connected to another, the readability is seriously reduced; The word following the acronym does not stand out as it should.

Private variables and class fields should have . prefix.

```
.lastErrorValue <- 0;

SomeClass <- function() {
  object <- list(
    .length = NA;
  )
  class(object) <- "SomeClass";
  object;
}
```

Apart from its name and its type, the scope of a variable is its most important feature. Indicating class scope by using . makes it easy to distinguish class variables from local scratch variables. This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer.

This naming convention makes it easy to exclude these objects from the ones exported in the name spaces, e.g.

```
# NAMESPACE file for package not exporting private objects:
exportPattern("^[^\\.]")
```

A side effect of the . naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods:

```
setDepth.SomeClass <- function(this, depth) {
  this$.depth <- depth;
  this;
```

```
}
```

An issue is whether the . should be added as a prefix or as a suffix. Both practices are commonly used, but the former is recommended because it is also consistent with how ls() works, which will only list R object with . as a prefix if and only if the argument all.names=TRUE.

It should be noted that scope identification in variables have been a controversial issue for quite some time. It seems, though, that this practice now is gaining acceptance and that it is becoming more and more common as a convention in the professional development community.

Private functions and class methods should have . prefix.

```
.anInternalUtilityFunction <- function(x, y) {
  # ...
}

.calculateIntermediateValue.SomeClass <- function(this) {
  # ...
}
```

The rational for this rule is the same as the one for the above rule about private variables and private class fields.

Arguments and generic variables should have the same name as their type.

```
setTopic(topic)       # NOT: setTopic(value)
                      # NOT: setTopic(aTopic)
                      # NOT: setTopic(x)

connect(database)     # NOT: connect(db)
                      # NOT: connect(oracleDB)
```

Reduce complexity by reducing the number of terms and names used. Also makes it easy to deduce the type given a variable name only.

If for some reason this convention doesn't seem to fit it is a strong indication that the type name is badly chosen.

Non-generic variables have a role. These variables can often be named by combining role and type:

Point startingPoint, centerPoint;
Name loginName;

All names should be written in English.
fileName; # NOT: filNamn
English is the preferred language for international development.

Variables with a large scope should have long names, variables with a small scope can have short names [2].

Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are i, j, k (or ii, jj, kk), m, n and for characters ch (c is not recommended since it used for concatenating vectors, e.g. c(1,2)).

The name of the object is implicit and should be avoided in a method name.
getLength(line); # NOT: getLineLength(line);
The latter seems natural in the class declaration, but proves superfluous in use, as shown in the example.

举报

kk22boy

发表于 昨天 22:30 | 只看该作者

地板

Specific Naming Conventions

The is prefix should be used for boolean variables and methods.
isSet, isVisible, isFinished, isFound, isOpen
Using the is prefix solves a common problem of choosing bad boolean names like status or flag. isStatus or isFlag

simply doesn't fit, and the programmer is forced to chose more meaningful names.
There are a few alternatives to the is prefix that fits better in some situations. These are has, can, should, and was prefixes:
hasLicense();
canEvaluate();
shouldAbort <- FALSE;

The term find can be used in methods where something is looked up.
findNearestVertex(vertex); findMinElement(matrix);
In cases where get is inappropriate or awkward, find can be used as a prefix. It gives the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

Moreover, find may indicate that there is a lookup that may take some computing time, whereas get indicates a more directory action of low cost. Contrary to, say, searchFor, both find and get indicate that there will be a unique answer.

The term initialize can be used where an object or a concept is established.
initializeFontSet(printer);
The American initialize should be preferred over the English initialise. Abbreviation init must be avoided.

Tcl/Tk (GUI) variables should be suffixed by the element type.
okButton, bgImage, mainWindow, leftScrollbar, nameEntry
Enhances readability since the name gives the user an immediate clue of the type of the variable and thereby the available resources of the object.

The List suffix can be used on names representing a list of objects.
vertex # one vertex
vertexList # a list of vertices

Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on the object.
Simply using the plural form of the base class name for a list (matrixElement (one matrix element), matrixElements (list of matrix elements)) should be avoided since the two only differ in a single character and are thereby difficult to distinguish.
A list in this context is the compound data type that can be traversed backwards, forwards, etc. (typically a Vector).
A plain array is simpler. The suffix Array can be used to denote an array of objects.

The n prefix should be used for variables representing a number of objects.
nPoints, nLines
The notation is taken from mathematics where it is an established convention for indicating a number of objects.
In addition to n the prefix nbrOf or the prefix numberOf can also be used. A num prefix must not be used.

The No suffix should be used for variables representing an entity number.
tableNo, employeeNo
The notation is taken from mathematics where it is an established convention for indicating an entity number.
An elegant alternative is to prefix such variables with an i: iTable, iEmployee. This effectively makes them named iterators.

Iterator variables should be called i, j, k etc.

```
for (i in seq(nTables)) {
  :
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators.
Variables named j, k etc. should be used for nested loops only.
Some prefer to use "doubled" variable names, e.g. ii, jj, kk, etc., because they are much easier to find using the editors search functions.

Complement names must be used for complement entities [2].

get/set, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down, min/max,
next/previous, old/new, open/close, show/hide
Reduce complexity by symmetry.

Abbreviations in names should be avoided.
computeAverage(); # NOT: compAvg();
There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated. Never write:
cmd instead of command
cp instead of copy
pt instead of point
comp instead of compute
init instead of initialize
etc.
Then there are domain specific phrases that are more naturally known through their acronym or abbreviations. These phrases should be kept abbreviated. Never write:
HypertextMarkupLanguage instead of html
CentralProcessingUnit instead of cpu
PriceEarningRatio instead of pe
etc.

Negated boolean variable names must be avoided.
isError; # NOT: isNotError
isFound; # NOT: isNotFound
The problem arise when the logical not operator is used and double negative arises. It is not immediately apparent what !isNotError means.

Associated constants (final variables) should be prefixed by a common type name.
COLOR.RED <- 1;

COLOR.GREEN <- 2;
COLOR.BLUE <- 3;
This indicates that the constants belong together, and what concept the constants represents.

Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do.

Increase readability. Makes it clear what the unit should do and especially all the things it is not supposed to do. This again makes it easier to keep the code clean of side effects.

举报

kk22boy

发表于 昨天 22:31 | 只看该作者

7楼

Files

R source files should have the extension .R.
Point.R
In [4] it says "the code files to be installed must start with a (lower or upper case) letter and have one of the extensions .R, .S, .q, .r, or .s. We recommend using .R, as this extension seems to be not used by any other software". Furthermore, these extensions are required for building R packages.

Classes should be declared in individual files with the file name matching the class name.

This improves the overview of the source directory of a package and simplifies editing and debugging.

File content must be kept within 80 columns.

精华 0
学术水平 117 点
热心指数 156 点
信用等级 96 点
在线时间 1085 小时
注册时间 2005-3-10
最后登录 2012-4-7
串个门 加好友
打招呼 发消息

80 columns is the common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several developers should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.

Special characters like TAB and page break must be avoided.

These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

The incompleteness of split lines must be made obvious [2].

```
totalSum <- a + b + c +
            d + e);
function(param1, param2,
         param3);
for (tableNo in seq(1, maxTable,
     by=tableStep))
```

Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint.
In general:
Break after a comma.
Break after an operator.
Align the new line with the beginning of the expression on the previous line.

Statements

Generic functions (under S3)

Generic functions must not specify the arguments [5].

```
foo <- function(...)        # NOT: foo <- function(object, arg1, arg2)
```

```
 UseMethod("foo")          #        UseMethod("foo")
                      # NOT:  foo <- function(object, ...)
                      #         UseMethod("foo")
```

Specify the object argument or the method arguments of a generic function will restrict any other methods with the same name to have the same argument. By also excluding the object argument, default functions such as search() will also be called if the generic function is called without any arguments.

Specify the generic method name in UseMethod() calls [5].

```
foo <- function(...)        # NOT:  foo <- function(...)
 UseMethod("foo")          #        UseMethod()
```

If one do not specify the method name in the UseMethod() the generic function can not be retrieved dynamically, e.g.

```
 fcn <- get("foo", mode="function")
 fcn(obj)
```

because the UseMethod() will try to call a function named "fcn" and not "foo".

Variables

Variables must never have dual meaning.

Enhances readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

Variables should be kept alive for as short a time as possible.

Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

Constants

Constants should never change value.

Since the R language does not support (final) constants it is up to the programmer to make sure they do not change values. If the RCC is followed, this is the same as saying that any variable with a name with all letters in upper case should never be modified.

Loops

Use seq(along=) in for statements if looping over a vector or a list.

```
for (kk in seq(along=x)) { # NOT:   for (kk in 1:length(x)) {
  :                        #         :
}                          #       }
                           #
                           # NOT:   for (kk in seq(length(x))) {
                           #         :
                           #       }
```

If x is empty, the use of 1:length(x) or seq(length(x)) results in an unwanted looping over 1:0, whereas seq(along=x) will loop over the empty set, i.e. no iteration.

Loop variables should be initialized immediately before the loop.

```
done <- FALSE;      # NOT:   done <- FALSE;
while (!done) {     #         :
  :                 #       while (!done) {
}                   #         :
                    #       }
```

The use of break and next in loops should be avoided.

These statements should only be used if they prove to give higher readability than their structured counterparts.

In general break should only be used in case statements and next should be avoided altogether.

The form repeat {} should be used for empty loops.

```
repeat {      # NOT:  while (TRUE) {
  :           #        :
}             #      }
```

This form is better than the functionally equivalent while (TRUE) since this implies a test against TRUE, which is neither necessary nor meaningful.

Conditionals

Complex conditional expressions must be avoided. Introduce temporary boolean variables instead [2].

```
if ((elementNo < 0) || (elementNo > maxElement) ||
    (elementNo == lastElement)) {
  :
}
```

should be replaced by:

```
isFinished      <- (elementNo < 0) || (elementNo > maxElement);
isRepeatedEntry <- (elementNo == lastElement);
if (isFinished || isRepeatedEntry) {
  :
}
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read and to debug.

The nominal case should be put in the if-part and the exception in the else-part of an if statement [2].

```
isError <- readFile(fileName);
if (!isError) {
  :
```

```
} else {
  :
}
```
Makes sure that the exceptions does not obscure the normal path of execution. This is important for both the readability and performance.

The conditional should be put on a separate line.
```
if (isDone)       # NOT:  if (isDone) doCleanup();
  doCleanup();
```
This is for debugging purposes. When writing on a single line, it is not apparent whether the test is really TRUE or not.

Side effects in conditionals must be avoided.
```
file <- openFile(fileName, "w");  # NOT: if (!is.null(file <- openFile(fileName, "w"))) {
if (!is.null(file)) {             #       :
  :                               #     }
}
```
Conditionals with side effects are simply very difficult to read. This is especially true for programmers new to R.

Miscellaneous

Assignments should be done using <-.
```
maxValue <- 23;  #  NOT:  maxValue _ 23
                 #  NOT:  23 -> maxValue
                 #  NOT:  maxValue = 23
```
Assignment using _ is considered really unsafe and unreadable, especially since it in many other languages is considered a valid character of a variable, e.g. MAX_VALUE. It has also been made deprecated in the latest versions of R.
Assignment using -> is neither considered a standard in a lot of programming languages and might confuse a newcomer.

Finally, assignment using = has recently be supported by R and is common in many other languages. However, by <- is still by far the most common assignment operator used and therefore also the recommended one.

The semicolon ; should be used whereever possible to emphasize the end of an expression.
sum <- sum + a[pos]; # NOT: sum <- sum + a[pos]
pos <- pos + 2;   # NOT: pos <- pos + 2
In R a newline will be synonyme with the ; if the expression ends at the end of the line. However, just using newlines is dangerous and might lead to errors, or worse, ambigious code, if a newline is deleted by misstake.
(This guideline has been subject to quite a few replies about how ugly the code looks with semicolons. We do not have any comments about this, but we do want to make it known that many disagrees with this guideline.)

The use of magic numbers in the code should be avoided. Numbers other than 0 and 1 should be considered declared as named constants instead.

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead.

Floating point constants should always be written with decimal point and at least one decimal.
total <- 0.0;   # NOT: total <- 0;
speed <- 3.0e8; # NOT: speed <- 3e8;

sum <- (a + b) * 10.0;
This empasize the different nature of integer and floating point numbers even if their values might happen to be the same in a specific case.
Also, as in the last example above, it emphasize the type of the assigned variable (sum) at a point in the code where this might not be evident.

Floating point constants should always be written with a digit before the decimal point.
total <- 0.5;   # NOT: total <- .5;

The number and expression system in R is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5 and there is no way it can be mixed with the integer 5.

SIGNATURE

如果该贴对您有些许帮助，希望你能回复一下或者评一下热心指数！谢谢！

举报

kk22boy

发表于 昨天 22:32 | 只看该作者

8楼

加关注

7	22
关注	粉丝

高级会员

一叶知秋

论坛币	63211 个
经验	40233 点
威望	1 级
帖子	1248
精华	0
学术水平	117 点
热心指数	156 点
信用等级	96 点
在线时间	1085 小时
注册时间	2005-3-10
最后登录	2012-4-7

Layout and Comments

Layout

Basic indentation should be 2.

```
for (ii in seq(nElements))
  a[ii] <- 0;
```

Indentation of 1 is too small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 is chosen to reduce the risk of having to wrap code lines.

Block layout should be as illustrated in example 1 below or example 2, and must not be as shown in example 3. Class, Interface and method blocks should use the block layout of example 2.

```
while (!isDone) {
  doSomething();
  isDone <- moreToDo();
}
while (!isDone)
{
  doSomething();
  isDone <- moreToDo();
```

串个门 加好友
打招呼 发消息

```
}
while (!isDone)
 {
   doSomething();
   isDone <- moreToDo();
 }
```

Example 1 and Example 2 are commonly used and some prefer the former whereas some the latter. Since it is more or less "impossible" to agree to use only one of these, both are recommended. In addition of being a matter of taste, both have there pros and cons.
Example 3 introduce an extra indentation level which doesn't emphasize the logical structure of the code as clearly as example 1 and 2.

S3 method declarations should have the following form:

```
someMethod.SomeClass <- function(object) {
  ...
}
```

The if-else class of statements should have the following form:

```
if (condition) {
  statements;
}

if (condition) {
  statements;
} else {
  statements;
}

if (condition) {
  statements;
```

```
} else if (condition) {
  statements;
} else {
  statements;
}
```

This follows partly from the general block rule above. Note that the else clause have to be on the same line as the closing bracket of the previous if (or else) clause. If not, the previous if clause is considered to be finished at previous line. When the R parse then reaches the else statement on the next line it complains (with an "Error: syntax error" message), because a else must have start with an if statement. Thus, the following is syntactically erroneous in R:

```
# This is...            # ...equivalent to this:
if (condition) {          if (condition) {
  statements;                 statements;
}                         };
else {                    else {
  statements;                 statements;
}                         }
```

A for statement should have the following form:

```
for (variable in sequence) {
  statements;
}
```

This follows from the general block rule above.

A while statement should have the following form:

```
while (condition) {
  statements;
}
```

This follows from the general block rule above.

A switch statement should have the following form:

```
switch(condition,
  "ABC" = {
    statements;
  },
  "DEF" = {
    statements;
  },
  "XYZ" = {
    statements;
  }
}
```

A tryCatch statement should have the following form:

```
tryCatch({
  statements;
}, error = function(error) {
  statements;
})

tryCatch({
  statements;
}, error = function(error) {
  statements;
}, finally = {
  statements;
})
```

This follows partly from the general block rule above.

Single statement if-else, for or while statements can be written without brackets.

```
if (condition)
  statement;

while (condition)
  statement;

for (variable in sequence)
  statement;
```

Use this only if the statement is short enough, and never use it when it spans multiple lines.

It is a common recommendation that brackets should always be used in all these cases. However, brackets are in general a language construct that groups several statements. Brackets are per definition superfluous on a single statement.

White Space

- Conventional operators should be surrounded by a space character.
- R reserved words should be followed by a white space.
- Commas should be followed by a white space.
- Colons should be surrounded by white space.
- Semicolons in for statements should be followed by a space character.

```
a <- (b + c) * d;          # NOT:   a<-(b+c)*d
while (TRUE) {             # NOT:   while(TRUE) ...
doSomething(a, b, c, d);   # NOT:   doSomething(a,b,c,d);
```

Makes the individual components of the statements stand out. Enhances readability. It is difficult to give a complete list of the suggested use of whitespace in R code. The examples above however should give a general idea of the intentions.

Logical units within a block should be separated by one blank line.

Enhances readability by introducing white space between logical units of a block.

Methods should be separated by 3-5 blank lines.

By making the space larger than space within a method, the methods will stand out within the class.

Statements should be aligned wherever this enhances readability.

```
value <- (potential        * oilDensity)   / constant1 +
         (depth            * waterDensity) / constant2 +
         (zCoordinateValue * gasDensity)   / constant3;

minPosition     <- computeDistance(min,     x, y, z);
averagePosition <- computeDistance(average, x, y, z);

# A function definition extending over multiple lines.
ll <- function(pattern=".*", ..., private=FALSE, properties="data.class",
               sortBy=NULL, envir=parent.frame()) {
  ...
}

# Alternatively, right alignment may be used.
ll <- function(pattern=".*", ..., private=FALSE, properties="data.class",
                                 sortBy=NULL, envir=parent.frame()) {
  ...
}
```

There are a number of places in the code where white space can be included to enhance readability even if this violates common guidelines. Many of these cases have to do with code alignment. General guidelines on code alignment are difficult to give, but the examples above should give some general hints. In short, any construction that enhances readability is allowed.

Comments

Tricky code should not be commented but rewritten. [2]

In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

All comments should be written in English.

In an international environment English is the preferred language.

Comments should be indented relative to their position in the code. [2]

```
while (TRUE) {        # NOT:    while (TRUE) {
  # Do something      #          # Do something
  something();        #          something();
}                     #        }
```

This is to avoid that the comments break the logical structure of the program.

Acknowledgments

I wish to thank (in alphabetic order) the following persons for valueable comments, sugestions, and improved guidelines:
Jan Kim, School of Computing Sciences, University of East Anglia, UK.
Gordon Smyth, Walter & Eliza Hall Institute of Medical Research, Australia.
Jason Turner, Inidigo Industrial Controls, New Zeeland.

References

[1] R Development Core Team, R Language Definition, ISBN 3-901167-56-0.
http://cran.r-project.org/manuals.html
[2] Java Code Conventions

http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
[3] Java Programming Style Guidelines v3.0, Geotechnical Software Services
http://geosoft.no/javastyle.html
[4] R Development Core Team, Writing R Extensions, ISBN 3-901167-54-4.
http://cran.r-project.org/manuals.html
[5] Henrik Bengtsson, Safely creating S3 generic functions using setGenericS3(), Division for Mathematical Statistics, Centre for Mathematical Sciences, Lund University, Sweden, 2002.
http://www.maths.lth.se/help/R/

SIGNATURE

如果该贴对您有些许帮助，希望你能回复一下或者评一下热心指数！谢谢！

举报

发帖

返回列表

高级模式

您需要登录后才可以回帖 登录 | 我要注册 用QQ帐号登录

发表回复 回帖后跳转到最后一页

论坛推荐 关闭

2012年上半年银行从业资格考试4月11日报名截止

2012年上半年银行从业资格考试4月11日报名截止 2012年上半年中国银行业从业人员资格认证考试定于6月2、3日在全国180个城市举行，本次考试科目包括公共基础、个人理财、风险管理、公司信贷和个人贷款,共五个科目。

查看 »