FAO SWS R Style Guide

Contents

1 R Style Rules

The following rules are designed for any modules uploaded to the SWS production server. However, some rules below are not strictly enforced, and those are written in italics.

1.1 File Names

Should end in .R and be meaningful.

1.2 Identifiers

Any style using underscores (my_function), dots (my.function), lower camel case (myFunction), or upper camel case (MyFunction) is acceptable. But, the style should be consistent across the entire R file. Also, "magic numbers" should be avoided. For example:

```
if(daysInMonth == 28)
    currentMonth <- "february"
```

should instead be replaced with

```
daysInFebruary <- 28
if(daysInMonth == daysInFebruary)
    currentMonth <- "february"
```

1.3 Line Length

Lines should generally wrap at 80 characters. 100 characters is allowable in certain scenarios, but lines should not exceed 100 characters.

1.4 Indentation

Indentation should always be with space characters, but may be two or four spaces. However, spaces should be consistent across the entire file. Using an editor such as R Studio will help you with this. Indentation should never be with tabs, only spaces, as tabs may be rendered differently in different editors. Indentation should occur whenever a set of braces occur, as well as whenever code is wrapped onto multiple lines. Within R Studio, code can be automatically indented by selecting all the code and pressing CTRL + I.

1.5 Spacing

The particular style of spacing that one uses is not important, but spaces should be in your code to keep it from being too cluttered. Additionally, the style followed should be consistent throughout the whole file. Below are some examples of different spacing styles, both of which are acceptable:

```
myFunc <- function(x = 1, y = 2){
    x + y
}
```

```
myFunc <- function (x=1, y=2) {
    x + y
}
```

Moreover, spacing can be placed within code to help with readability, but this is not required:

```
myComplexFunction <- function(longArgument  = 1,
                              y             = 3,
                              someArgument  = d[filter == TRUE, ],
                              otherArg      = data[, c("Test")] * 100){
    return(0)
}
```

1.6 Curly Braces

The preferred style is that an opening curly brace should never go on it's own line, and a closing curly brace should always go on it's own line. However, variations are allowed but must be consistent throughout the file:

```
f <- function(x, y, z){
    x + y + z
}
```

```
f <- function(x, y, z)
{
    x + y + z
}
```

Curly braces are required around one-line if, for, while, ... statements. Thus, the following code is poorly formatted (even though it runs ok):

```
if(x == 1)
    y <- 2
else
    y <- 1
```

Instead, this should be written as

```
if(x == 1){
    y <- 2
} else {
    y <- 1
}
```

1.7 Assignment

Both `<-` and `=` are allowed, but must be consistent throughout the entire file. If a user has no preference, then the `<-` is recommended, as this is in line with R standard recommendations.

1.8 Semicolons

These should never be used.

1.9 Commenting Guidelines

Thorough commenting should be in your code so that another programmer can easily understand what you're doing. Comments should begin with one or multiple #'s and a single space. Short comments can be placed at the end of lines, and should be preceded by at least one space, one #, and one space. Examples:

```
## This is a comment
x <- 1   # Assign the value of 1 to x
```

When editing or updating a file, it is good practice to annotate with comments the changes you made, or to explain confusing parts of the code. In such cases, you should also add your name and date.

```
## (JMB, 6/5/2015) The following code does ...
d[(filter) & x > lowerBound, newValue := oldValue + oldValueAdjustment]
```

1.10 Function Definitions and Calls

Function definitions should first list arguments without default values, followed by those with default values. In both function definitions and function calls, multiple arguments per line are allowed; line breaks are only allowed between assignments.

1.11 Function Documentation

Any non-trivial function should be documented following the roxygen2 coding style. Not all these sections are required, but the function should have at least a title (the first line), a short description (the next chunk), all parameters documented, and a description of the return value. Moreover, these descriptions should be meaningful, and not like:

```
##' My Function
##'
##' Runs the analysis.
##'
##' @param func The function
##'
##' @return The result
```

Instead, here is an example of a better documented function:

```
##' Mixed Model for Imputation
##'
##' This function imputes missing values with a linear mixed model.
##'
##' The default functionality of this model is to fit a linear mixed model to
##' the data.  time and intercept are assumed to be fixed effects, and the
##' random effects are specified by the byKey parameter of
##' imputationParameters.  So, for example, byKey may reference the variable
##' containing country data.  In that case, a model is fit which assumes a
##' linear relationship between production (or whatever dependent variable the
##' user has specified) and time.  However, the intercept and slope of this
##' fit varies from country to country, and so country is considered a random
##' effect.
##'
##' Moreover, the model fit is not a simple linear regression, but rather a
##' spline regression (using the bs function from the \pkg{splines} package).
##' The fit of this model will therefore depend on the number of degrees of
##' freedom of this spline model (and, if the degrees of freedom is 1, then the
##' simple linear regression model is used).
##'
##' @param data The data.table object containing the data.
##' @param df The number of degrees of freedom for the spline.
##' @param weights The weights for the observations.
##' @param modelFormula Formula specifying how the dependent variable (value)
##' depends on the other columns of data.  Should be a valid mixed model
##' formula, as it will be passed to lmer (R's mixed model function).
##' @param imputationParameters A list of the parameters for the imputation
##' algorithms.  See defaultImputationParameters() for a starting point.
##'
##' @return Returns a vector of the estimated/imputed values.  If a value
##' existed in the original data, then an NA is returned in that location.
##'
##' @export
##'

defaultMixedModel = function(data, df = 1, weights = NULL, modelFormula = NULL,
                             imputationParameters){
    Write code here...
}
```

Additionally, if the function has any side effects, these should be documented very clearly! Global variables should only be created in extreme scenarios, as they can create such side effects. However, data.table objects are passed by reference and hence modifications inside a function will change the data.table. These effects should be documented (and kept to a minimum, when possible).

Functions should also be kept reasonably short and should be split into coherent blocks.

Never define a function within a loop. This creates unnecessary overhead, and if that loop iterates many times, alot of computation is spent on redefining the same function.

1.12 Anonymous Function

An anonymous function is an unnamed function, usually something that is constructed as part of a *apply call. Such functions are useful for quick inline evaluation, and don't necessarily need to be documented.

However, if the function is complex enough, it should be defined and named outside of the *apply call with the usual function documentation (i.e. roxygen-style comments). Additionally, if it is used more than once, it should be defined outside of the *apply call (to avoid having to update the function in multiple locations).

2 R Language Rules

2.1 attach

Never use attach.

2.2 .Rprofile

Your code should run without any custom definitions in your .Rprofile. For example:

```
options("stringsAsFactors")
```

```
## $stringsAsFactors
## [1] TRUE
```

```
options("stringsAsFactors" = FALSE)
options("stringsAsFactors")
```

```
## $stringsAsFactors
## [1] FALSE
```

For example, suppose you place the second line of code above (that assigns stringsAsFactors to FALSE) in your .Rprofile. Then, when you read in a dataset with characters, that column will be treated as a character column. However, if a different user runs your same code, that column will be created as a factor. This can cause errors which are nearly impossible to debug, as the reason for the difference between the two behaviours is not clear.

This file can be useful, however, for changing some other options. For example:

```
## Don't ask me for my CRAN mirror every time
options("repos" = c(CRAN = "http://cran.rstudio.com/"))

## Penalize scientific notation (only affects display):
options(scipen=20)
```

2.3 Loading External Code

Libraries are external packages which are unavoidable when programming in R. Functions from libraries should be used in one of two ways:

```
library(data.table)
reshape2::cast(d, x ~ y)
```

In the first case, the entire data.table package is loaded, and this is the desired behavior if that package will be used extensively throughout the script. If, however, a package is to be used only a few times, the second approach is preferred: not loading the entire package helps prevent potential naming conflicts. Additionally, it makes the code more understandable. Never use require in place of library: it allows the function to continue even if the loading of the library failed, and this will likely cause errors later (that will presumably be harder to understand and debug).

2.4 data.table

Discuss complications with pass-by-reference...

3 General Layout and Ordering

The following sections should exist in the specified order. However, not all sections are required in a particular R file. The single hash comments are meant to separate different sections.

```r
# Beginning comment:
## This file does ...
## We assume that you have a connection to the SWS, that a linear model is
## reasonable, etc.
## We still need to add the following features: ...
## Author: Me

# All library() calls followed by all source() calls
library(ggplot2)
library(faosws)
source("myFunctions.R")
source("/path/to/file/myOtherFile.R")

# Function definitions
##' My Function
##'
##' This function doesn't do anything.
##'
##' @param x Numeric, an input value.
##'
##' @return Returns a numeric vector of the same length as x but with values
##' x + 1

myFun <- function(x){
    x + 1
}

# Executed statements, if applicable.
x <- rnorm(10)
y <- myFun(x)
```

4 SWS Specific Content

4.1 Repository Organization

How should repositories be organized? Where should we place the R files, the .xml file, and the .R file that runs the module? What about documentation files (man, vignettes)?

4.2 Local Testing Environment

All scripts uploaded to the SWS will likely need to be run in a local/test environment as well as on the SWS. To test if the script is currently being run on the server, use the following R code block:

```r
DEBUG_MODE <- Sys.getenv("R_DEBUG_MODE")
if(DEBUG_MODE == TRUE){ # Must test since DEBUG_MODE could be ""
    GetTestEnvironment(
        baseUrl = "https://hqlqasws1.hq.un.fao.org:8181/sws",
        token = "88ee026d-cd28-40ed-8266-284541d346b9"
    )
}
```

4.3 Code Organization

It is generally preferable to have functions in a separate folder rather than one long .R file. Each function can then be documented and easily accessed.

4.4 R Packages

For complex analyses related to SWS modules should be organized following the R package structure. For an example of such a structure, please see https://github.com/SWS-Methodology/sws_imputation/tree/master/faoswsImputation. For more details around the R practices, please see http://cran.r-project.org/doc/manuals/r-release/R-exts.html#Package-structure. Here's an overview of the structure:

- R: This should be a directory containing all R functions for the package.
- man: This should be a directory containing all the .Rd files (or documentation files). Creation of these files is easy by executing `devtools::document` on the package directory (R will scan the scripts in the R directory and automatically create documentation files based on the roxygen2 comments written by the programmer).
- data: Optional, but highly recommended. This directory can contain sample datasets that show how the package works. These datasets should generally be for reference only, and should not be required to actually run any of the code.
- vignettes: This directory contains .Rnw files which will become vignettes once the package is built. These vignettes provide extensive documentation of the package and examples of how to run code.
- tests: This directory contains unit tests that are ran when the package is compiled and provide a very good tool for checking the code written by developers. Additionally, the testthat package can be used to run these tests frequently while updating package code. This is an extremely good practice, but it's currently not implemented on any FAO SWS packages.
- DESCRIPTION: A file describing the package (i.e. version, author, etc.)
- NAMESPACE: This package is automatically created by `devtools::document`. It specifies function imports and exports, but shouldn't be edited by hand if using `devtools::document`.
- ...: Additional directories are also accepted and useful, but this document is meant to just discuss R coding standards.

The R modules should be implemented as packages for several reasons:

- The code, documentation, and manuals can be easily transferred to a new programmer and he/she can begin using them without much guidance (assuming the documentation is thorough).
- Checks are run while compiling packages to enforce certain constraints and warnings notify the compiler when certain types of documentation are missing (for example). Thus, building a package provides a useful way for checking your code.
- This structure forces programmers to follow a rigorous code structure and ensures consistency.

4.5 Additional Styles/Rules for SWS

- The code `rm(list = ls())` should never be used within any script uploaded to the working system, as this removes crucial SWS variables and will result in an error.

5 Additional Topics

- Package structure, checks to pass.
- Ensuring inputs: we should implement simple tests at the beginning of each function to ensure that the input parameters are valid/meet assumptions of the function.
- What rules should we have around which functions should be exported from a package?
- Avoid data.table::setnames in functions so as to prevent colnames changing unexpectantly when errors occur.
- roxygen: in order to ensure the NAMESPACE file is correctly written, functions from external packages should be referenced via :: in the FAO package. Or, the roxygen comment block should have @@import or @@importFrom.
- Some scripts make use of unlink and file.copy to delete all scripts within a directory and then copy them back in. I'd recommend we avoid this type of behavior, and the use of unlink, as much as possible.
- All repositories with FAO code on Github should have a disclaimer in the README.md file, something to the effect of " All work under this repository represents the latest status of development and made public for collaboration. It however, does not reflect the current state of the system and use of the program is at the discretionary of the users. "