

Customizing and extending Emacs Speaks Statistics

Frédéric Santos¹

January 18, 2021

¹frederic.santos@u-bordeaux.fr

A short tutorial for customizing ESS

Target audience

This tutorial is for R-users beginning to use ESS. We only assume that you have mild familiarity with Emacs, and that you know how to customize Emacs by filling your `.emacs` or `init.el` file.

Goals

- Helping newcomers and beginners to have a more friendly and efficient setup
- Helping users migrating from other IDEs to keep behaviours and habits they are used to
- Showcasing some other Emacs packages that play well with ESS and may improve your coding experience

These slides are a companion for the video tutorial available at ...

An opinionated tutorial?

Some parts of this tutorial (e.g., replacing Flymake by Flycheck for syntax checking, or using company instead of other solutions for auto-completion) might be seen as quite opinionated. However, when several options or approaches were available for a given task, the most widespread and efficient (I hope!) was presented.

If you prefer other approaches, or simply want to give a try to some other ones, the ESS manual remains the canonical and exhaustive documentation: <http://ess.r-project.org/Manual/ess.html>

A word about use-package

In this document, several pieces of Emacs Lisp code will be proposed so that you can use them in your init file.

It is assumed that you use use-package for your init file: the Emacs Lisp code can be adapted in a straightforward manner if you do not use it.

As a reminder, this is the minimal code to add in your init file so as to use use-package, once it has been installed:

```
;; Make sure that use-package is installed:  
(unless (package-installed-p 'use-package)  
  (package-refresh-contents)  
  (package-install 'use-package))  
;; Load use-package:  
(eval-when-compile  
  (require 'use-package))
```

Visibility of evaluation

From an R source file (i.e., an ESS[R] buffer), you can send code to the underlying R process with shortcuts like `C-c C-b` (`ess-eval-buffer`), `C-RET` (`ess-eval-region-or-line-and-step`), etc.

In the inferior R process buffer, ESS can display either both the commands sent and their results, or only the results. The choice between the two can be made through the variable `ess-eval-visibly`.

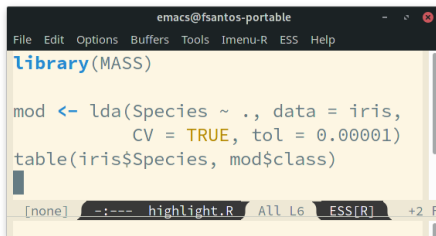
This variable can be `t` (eval visibly and wait for process) or `nil` (eval invisibly); but by default it is set to `'nowait'`, which means that ESS shows input commands in the process buffer, but allows for further editing in the source buffer even while the previous instructions sent are still running. This way, ESS is never freezing while evaluating code.

If you prefer to eval R code invisibly, add in your init file:

```
(setq ess-eval-visibly nil)
```

Syntactic highlighting in ESS R source buffers I

By default, ESS highlights literals, assignments, source functions and reserved words. An example of default syntactic highlighting is as follows:

A screenshot of the Emacs ESS R source buffer. The window title is 'emacs@fsantos-portable'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Imenu-R', 'ESS', and 'Help'. The code is as follows:

```
library(MASS)

mod <- lda(Species ~ ., data = iris,
           CV = TRUE, tol = 0.00001)
table(iris$Species, mod$class)
```

The code is syntactically highlighted: 'library' is blue, 'MASS' is black, 'mod' is black, '<-' is black, 'lda' is black, 'Species' is black, '~' is black, '.' is black, 'data' is black, '=' is black, 'iris' is black, 'CV' is black, '=' is black, 'TRUE' is orange, 'tol' is black, '=' is black, '0.00001' is black, 'table' is black, 'iris\$Species' is black, and 'mod\$class' is black. The status bar at the bottom shows '[none]', 'highlight.R', 'All L6', 'ESS[R]', '+2 F'.

Figure 1: Default patterns for syntactic highlighting.

You can add more patterns to be highlighted, or remove some of them, by customizing the variable `ess-R-font-lock-keywords`.

Syntactic highlighting in ESS R source buffers II

For instance, if you also want the numbers and the function calls to be highlighted:

```
;; Font lock keywords for syntactic highlighting:  
(setq ess-R-font-lock-keywords  
  '((ess-R-fl-keyword:keywords . t)  
    (ess-R-fl-keyword:constants . t)  
    (ess-R-fl-keyword:modifiers . t)  
    (ess-R-fl-keyword:fun-defs . t)  
    (ess-R-fl-keyword:assign-ops . t)  
    (ess-R-fl-keyword:%op% . t)  
    (ess-fl-keyword:fun-calls . t)  
    (ess-fl-keyword:numbers . t)  
    (ess-fl-keyword:operators)  
    (ess-fl-keyword:delimiters)  
    (ess-fl-keyword:=)  
    (ess-R-fl-keyword:F&T . t))))
```

Syntactic highlighting in ESS R source buffers III

The same code is now highlighted differently, and maybe somewhat more clearly:



The screenshot shows an Emacs window titled 'emacs@fsantos-portable'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Imenu-R', 'ESS', and 'Help'. The main text area displays R code with the following syntax highlighting: 'library' is blue, 'MASS' is black, 'mod' is black, '<-' is blue, 'lda' is blue, 'Species' is black, '~' is black, '.' is black, 'data' is black, '=' is black, 'iris' is black, 'CV' is black, '=' is black, 'TRUE' is yellow, 'tol' is black, '=' is black, '0.00001' is yellow, 'table' is blue, 'iris\$Species' is black, 'mod\$class' is black. The status bar at the bottom shows '[none]', '--- highlight.R', 'All L6', 'ESS[R]', '+2 F'.

```
library(MASS)

mod <- lda(Species ~ ., data = iris,
           CV = TRUE, tol = 0.00001)
table(iris$Species, mod$class)
```

Figure 2: More custom patterns for syntactic highlighting.

Seeing matching parentheses

Directly taken from the ESS manual

(<http://ess.r-project.org/Manual/ess.html#Parens>):

“Emacs has facilities for highlighting the parenthesis matching the parenthesis at point. This feature is very useful when trying to examine which parentheses match each other. This highlighting also indicates when parentheses are not matching.”

To activate parenthesis matching in ESS[R] (source) buffers, add this to your init file:

```
;; Activate global mode for parenthesis matching:  
(show-paren-mode)
```

Navigating through matching parentheses

Here are some convenient tricks for navigating through parenthetical groups (this can be useful when dealing with large paren groups, e.g. when developing a shiny UI):

Shortcut	Elisp function (Docstring)
C-M-p	<code>backward-list</code> (Move backward across one balanced paren group)
C-M-n	<code>forward-list</code> (Move forward across one balanced paren group)
C-M-SPC	<code>mark-sexp</code> (Set mark at the end of the paren group)
C-M-k	<code>kill-sexp</code> (Kill from point to end of paren group)

Table 1: Some useful shortcuts for dealing with parenthetical groups.

For instance, when the point is over a closing parenthesis, C-M-p brings you to the matching opening parenthesis. Then, C-M-k kills to whole paren group.

Syntax checking: Flycheck vs. Flymake

ESS has facilities for on-the-fly syntax checking. Instead of using Flymake, which is the default choice, using Flycheck appears to be a better and more stable option. The Flycheck documentation allows for a comparison between those two packages: <https://www.flycheck.org/en/latest/user/flycheck-versus-flymake.html>

To switch from Flymake to Flycheck, you can add the following in your init file:

```
;; Remove Flymake support:  
(setq ess-use-flymake nil)  
;; Replace it (globally) by Flycheck:  
(use-package flycheck  
  :ensure t  
  :init  
  (global-flycheck-mode t))
```

On-the-fly syntax checking with Flycheck I

Using Flycheck with ESS first requires you to install the R package `lintr`:

```
## Install stable CRAN version:  
install.packages("lintr", dep = TRUE)  
## OR  
## Install latest Github devel version:  
devtools::install_github("jimhester/lintr")
```

`lintr` is an R package that offers facilities for static code analysis. It integrates with the main IDEs and text editors (Emacs, Rstudio, vim, etc.). In particular, it has native support for ESS + Flycheck.

On-the-fly syntax checking with Flycheck II

Once both Flycheck and `lintr` are installed, your R code is analyzed “on-the-fly” while you are typing. Several checks are performed, including:

- R code style: correct use of `snake_case`, convenient spacing around all operators, etc.
- undeclared variables in function body
- bad use of `=` for variable assignment
- unmatched parentheses

The following screenshot (Fig. 3) gives some examples of such checks.

To display all syntax error in a dedicated buffer, use `M-x flycheck-list-errors` (bound to `C-c ! 1` by default).

On-the-fly syntax checking with Flycheck III

The screenshot shows an Emacs window titled 'emacs@fsantos-portable'. The main editor contains R code with several syntax errors. A Flycheck buffer is open at the bottom, displaying the errors detected by the 'lintr' checker.

Code in the editor:

```

1 MYvariable <- 2
2
3 myfun <- function(x) {
4   return(2*x + y)
5 }
6
7 z = c(1, 3)
8
9 mean(z)[]

```

Flycheck errors buffer:

File	Line	Col	Level	ID	Message (Checker)
lintr.R	1	1	info		Variable and function name style should be snake_case. (r-lintr)
lintr.R	4	13	info		Put spaces around all infix operators. (r-lintr)
lintr.R	4	18	warning		no visible binding for global variable 'y' (r-lintr)
lintr.R	7	3	info		Use <-, not =, for assignment. (r-lintr)
lintr.R	9	8	error		')' inattendu(e) (r-lintr)

The status bar at the bottom shows: [R] -:***- lintr.R All (U:%%- *Flycheck errors* for buffer lintr.R All (6,0) Flycheck errors orb Org-roam company

Figure 3: An example of syntax errors detected by lintr and displayed in a dedicated Flycheck buffer.

Completion I

As mentioned in the ESS manual, there are several completion frameworks for writing R code with ESS. The Emacs package `company` is an elegant solution, which also supports many other programming languages.

Here is a minimal piece of Emacs code to add in your init file to install and load `company`:

```
(use-package company
  :ensure t
  :config
  ;; Turn on company-mode globally:
  (add-hook 'after-init-hook 'global-company-mode)
  ;; Only activate company in R scripts, not in R console:
  (setq ess-use-company 'script-only))
```

Completion II

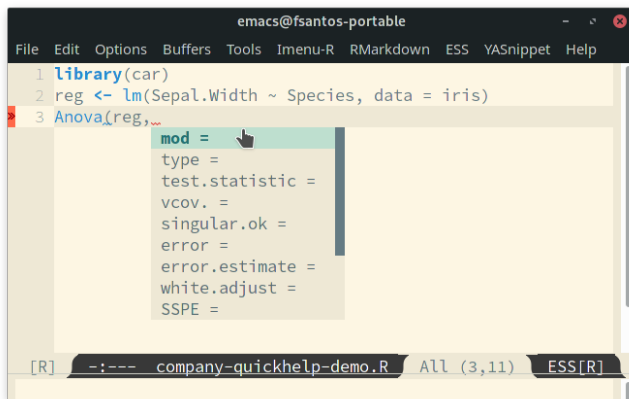


Figure 4: An example of code completion with `company`: various candidates are proposed for the arguments of the function `car::Anova()`.

Completion III

company offers completion candidates in various contexts: function name, argument name within a function call (as in Fig. 4), object name.

It may seem preferable to adopt a non-intrusive workflow. For functions or objects names, completion starts automatically after you type a few letters. For arguments names within a function call, it is suggested that you trigger manually the completion only when you need it. This can be done with M-x `company-complete`, or more conveniently, by binding this function to a convenient shortcut. For example, to bind it to F12, add the following to your init file:

```
;; Use F12 to trigger manually completion on R function args:  
(add-hook 'ess-r-mode-hook  
  '(lambda ()  
    (local-set-key (kbd "<f12>") #'company-R-args)))
```

Completion IV

Of course, further customization of `company` can be done in your `init` file. For instance:

```
;; More customization options for company:  
(setq company-selection-wrap-around t  
  ;; Align annotations to the right tooltip border:  
  company-tooltip-align-annotations t  
  ;; Idle delay in seconds until completion starts automatically  
  company-idle-delay 0.45  
  ;; Completion will start after typing two letters:  
  company-minimum-prefix-length 2  
  ;; Maximum number of candidates in the tooltip:  
  company-tooltip-limit 10)
```

Documentation popups I

company-quickhelp allows for documentation popups, e.g. to further describe function arguments.

The screenshot shows the Emacs editor interface with the title bar 'emacs@fsantos-portable'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Imenu-R', 'RMarkdown', 'ESS', 'YASnippet', and 'Help'. The left pane contains R code:

```
1 library(car)
2 reg <- lm(Sepal.Width ~ Species, data = iris)
3 Anova(reg,
  mod =
  type =
  test.statistic =
  vcov. =
  singular.ok =
  error =
  error.estimate =
  white.adjust =
  SSPE =
  error.df =
```

A documentation popup is displayed over the 'type =' line. It contains the following text:

```
1
2 R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
3 Copyright (C) 2020 The R Foundation for Statistical Computing
4
5 type of test, "II", "III", 2, or 3. Roman numerals are equivalent to
6 the corresponding Arabic numerals.
7 vous pouvez le redistribuer sous certaines conditions.
8 Tapez 'license()' ou 'licence()' pour plus de détails.
9
10 R est un projet collaboratif avec de nombreux contributeurs.
11 Tapez 'contributors()' pour plus d'information et
12 'citation()' pour la façon de le citer dans les publications.
13
14 Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
15 en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
16 Tapez 'q()' pour quitter R.
17
18 > setwd('/home/fsantos/Documents/')
19 > library(car)
20 Le chargement a nécessité le package : carData
21 > reg <- lm(Sepal.Width ~ Species, data = iris)
22 >
```

Figure 5: Documentation popups with company-quickhelp.

Documentation popups II

The minimal elisp code to add to your init file is straightforward:

```
(use-package company-quickhelp
  :ensure t
  :config
  ;; Load company-quickhelp globally:
  (company-quickhelp-mode)
  ;; Time before display of documentation popup:
  (setq company-quickhelp-delay 0.3))
```

By default, the documentation popup is shown automatically. You can adjust the time before the popup shows up by customizing the variable `company-quickhelp-delay`.

Code snippets

yasnippet is an Emacs package allowing for the expansion of whole pieces of code you often use (*snippets*) from one given abbreviation.

Key features of yasnippet

- All code snippets are stored as plain-text files in one given directory, so that they are easy to share with other people, and can be easily version controlled.
- As a corollary, it is also easy to retrieve and use large collection of snippets already available online. For instance, Andrea Crotti maintains a great collection available at <https://github.com/AndreaCrotti/yasnippet-snippets>.
- Although we only demonstrate its use within ESS and R here, note that yasnippet is not an R-specific solution, and that you can use it for any other programming language.

Setting up yasnippet |

To set up yasnippet, proceed through the following steps:

- 1 Create a directory `snippets/` at some convenient location, and add a subfolder `ess-r-mode/` in this directory.
- 2 Add the minimal following code in your init file:

```
(use-package yasnippet
  :ensure t
  :config
  ;; Indicate the directory containing your snippets:
  (setq yas-snippet-dirs '("path/to/your/snippets"))
  ;; Load your snippets on startup:
  (yas-reload-all)
  ;; Turn on yasnippet (minor) mode when editing R files:
  (add-hook 'ess-r-mode-hook #'yas-minor-mode))
```

Setting up yasnippet II

- ③ You can now fill your `snippets/ess-r-mode/` directory with your own snippets. For instance, create a file `function` (without any extension) in this directory, with the following contents:

```
#name : function
#key : fun
# --
${1:name} <- function(${2:args}) {
  ${3:body}
}
```

Each snippet has a unique name, and can be triggered by typing a given key (followed by TAB). As we will see later on, the present snippet allows for the expansion of a template for defining new R functions more easily. The yasnippet manual gives more details about the expected syntax to define your own code snippets:

<http://joaotavora.github.io/yasnippet/>.

Setting up yasnippet III

- ④ Now your snippets directory should look like:

```
| └─ snippets
|     └─ ess-r-mode
|         └─ function
```

Feel free to add or retrieve (a lot!) more snippets, i.e. to add more template files within the `ess-r-mode` sub-directory.

Using yasnippet in an ESS[R] buffer

While you are editing an R source file with ESS, each snippet can be triggered by typing its key and then pressing TAB. You can then navigate through the placeholders of the expanded template by pressing TAB again.

For instance, with our previously defined snippet, typing `fun` followed by TAB will expand the full `function` template; you will then be able to specify easily a value for each of the three placeholders (the function's name, its args and body).

Note that yasnippet has a short video tutorial, available at <https://www.youtube.com/watch?v=ZCGmZK4V7Sg>.