Consistent Banner Comments for R Scripts

Bill Venables

2016-12-05

Contents

Rationale

This tiny package is purely for the convenience of authors who wish to document their code with the occasional block of clearly marked comments in order to make the code more easily navigable.

A common practice is to use comment lines enclosed in some kind of band, or box of display characters. We call such a block of comments a **banner comment**, and by using them sparingly and judiciously code can be made much easier to navigate visually and hence to maintain.

Examples might include blocks such as the following to initiate a major code section:

```
################################################################################
################################################################################
###                                                                          ###
###                              SECTION 1                                   ###
###                     DATA INPUT AND INITIALIZATION                        ###
###                                                                          ###
################################################################################
################################################################################
```

Subsections might be flagged by less prominent comments such as one like this

```
##------------------------------------------------------------------
##                       Primary data input                        -
##------------------------------------------------------------------
```

or a more minor one like

```
##..............................
##  Some minor glitches in the data
##  need special treatment here
##..............................
```

Banner comments look much better if there is a consistent formatting throughout the script so that sections and sub-sections can be readily identified. This is quite easy to do, but to do it properly can take some editing and drafting time.

The simple tool we offer here aims to make the formatting tasks essentially no more work than typing the text itself. It presumes that while the R script is being drafted there will be a console window open as well, but this is almost *de rigeur* these days.

How to use it

The package `bannerCommenter` provides a single main function, `banner`, along with a few helpers.

To make a banner such as the first one above you could simply type the text into the console window via a call to the main function:

```
library(bannerCommenter)
banner("Section 1:", "Data input and initialization", emph = TRUE)
```

```
###############################################################################
###############################################################################
###                                                                         ###
###                              SECTION 1:                                 ###
###                    DATA INPUT AND INITIALIZATION                        ###
###                                                                         ###
###############################################################################
###############################################################################
```

This provides a formatted comment that can be copy-and-pasted into the script. However, if the operating system allows it, as well as displaying the comment in the console window the result is *also copied onto a* `clipboard` *file or pipe*, so the "copy" part of the copy-and-paste should not be required.

Two other conveniences are also provided.

1. Since this is likely to be a common form of banner comment, a simple front-end function `section()` is provided which simply called `banner(..., emph = TRUE)`, that is with the `emph` argument having a different default value.
2. Rather than typing strings with quite delimiters separated by commas, if the function is called with no string arguments, and in an interactive session of course, the strings are read from the terminal, line by line, with prompts issued as with the `scan` function. An empty line indicates the lines are complete.

An example, again with the same banner comment, is as follows

```
> section()
1: Section 1
2: Data input and initialization
3:

###############################################################################
###############################################################################
###                                                                         ###
###                              SECTION 1                                  ###
###                    DATA INPUT AND INITIALIZATION                        ###
###                                                                         ###
###############################################################################
###############################################################################

>
```

At this point the user should be able to paste the banner comment into the script in the usual way. If the automatic clipboard facility is not available the band displayed in the console window, essentially for checking, can easily be used for manual copy-and-paste.

In a series of similar comment banners the function would normally be invoked by command line recall making only the typing of the text itself necessary.

Arguments

At first sight `banner` seems to have a bewildering number of arguments, but most have sensible defaults and there are four front-end functions like `section` that handle simple special cases where the default values are slightly different.

The full list of arguments to `banner` and their default values are listed in the table below. Note that some argument defaults refer to the values of other arguments.

Argument	Meaning	Default
`x, ...`	One or more strings, which may be missing. (Single strings may be further broken by '`\n`'.)	`<none>`
`emph`	Do you want emphasis, i.e. a bigger, bolder banner?	`FALSE`
`snug`	Do you want any box to be close fitting?	`FALSE`
`upper`	Do you want the text to be in upper case?	`emph`
`centre`	Do you want the text lines to be centred?	`!snug && emph`
`leftSideHashes`	How many hash characters to the left?	`2 + emph`
`rightSideHashes`	How many hash characters to the right	`leftSideHashes`
`minHashes`	How long do you want the bands (at least)?	`(!snug) * (65 + 10 * emph)`
`numLines`	How many lines above and below do you want?	`1 + emph`
`bandChar`	What character do you want to use for the bands?	`"#"`
`center`	Foreign, alternative spelling of '`centre`'	`centre`

Examples

As well as `section` there are three other front-end functions that simply act as a call to `banner` with different default values for some of the arguments. Rather than describe them in detail it suffices simply to provide a few example, beginning with the primary function itself.

```
txt <- "This is the text of a comment"
banner(txt)  ## default heavy style
```

```
######################################################################
##                This is the text of a comment                     ##
######################################################################
```

```
banner(txt, centre = TRUE, bandChar = "-")
```

```
##--------------------------------------------------------------------
##                This is the text of a comment                     --
##--------------------------------------------------------------------
```

```
boxup(txt, snug = TRUE, bandChar = "=")
```

```
##==================================
##  This is the text of a comment   =
```

```
##==================================
```

```
open_box(txt, bandChar = ":")
```

```
##::::::::::::::::::::::::::::::::
##  This is the text of a comment
##::::::::::::::::::::::::::::::::
```

```
block("This is a chatty comment.  Entering it this way just",
      "saves a tiny bit of typing but it can be useful if",
      "you need multiple initial hash marks, as you may when",
      "using Emacs/ESS, for example.",
      "Or if you want the lines centred for some odd reason.",
      center = TRUE)
```

```
###  This is a chatty comment.  Entering it this way just
###  saves a tiny bit of typing but it can be useful if
###  you need multiple initial hash marks, as you may when
###  using Emacs/ESS, for example.
###  Or if you want the lines centred for some odd reason.
```

```
boxup("")  ## short lines of uniform length, for use as a separator
```

```
##------------------------------------------------------------------
```

```
section("")  ## heavier, longer double lines to separate bigger things
```

```
################################################################################
################################################################################
```

Notes

The number of leading hash characters

In some editing systems, where a comment has only white space before it on a single line, the number of leading comment characters is significant. It affects how the line is changed under automatic reformatting. For example, `Emacs/ESS` adopts this convention by default:

- A line with a single leading comment character, only, is aligned so that it begins near the middle of the line. (I have no explanation as to why!)
- A line with two leading comment characters, only, is aligned as if it were an active code line. This is often useful.
- A line with three or more leading comment characters is aligned so that it begins in the first column and so occupies the whole line.

Some commentators recommend using a single leading comment character for all comments, and `RStudio`, for example, facilitates this choice. However if the same code is handled by `Emacs/Ess` the comments are liable to be right shifted to start in the middle of the line (unless the default is changed, of course, which is not initially very clear).

This may be useful to keep in mind when two or more systems may be used to maintain the same `R` scripts.

Sending text strings to the clipboard

A helper function used in this package may be useful in its own right. The function `copy_to_clipboard` allows text strings to be copied to a clipboard file (or pipe) in a reasonably cross platform way, at least for `Linux`, `Windows` and `Mac OS`.

A call such as

```
copy_to_clipboard(txt)
```

will return `txt` invisibly, but will have the side effect of transferring any strings in the `txt` object to a clipboard device. In effect it behaves like a `print` method, but with the "printing" going on to a clipboard device rather than on to `stdout`.

To work on `Linux` the system command `xclip` has to be installed and visible on the `PATH` and on `Mac OS` the system command `pbcopy` has similarly to be installed and visible. On `Windows` it should work universally.

Note that this is *not* a file or pipe connection in itself, but a *function* which transfers strings to an appropriate clipboard device. Thus, for example, to write a short data frame onto a clipboard device in a way that works across the three platforms, you may need to do something like the following:

```
library(dplyr)
mtcarsText <- datasets::mtcars   %>%
  capture.output(write.table(.)) %>%
  copy_to_clipboard(sep = "\n")
```

At this point `mrcarsText` is a character string vector with the lines of `datasets::mtcars` as its elements, and the information would be available on the clipboard for a paste operation.

Gratuitous advice

Use sparingly and judiciously. Most comments will simply be done by typing the `#` character and proceeding. They will usually *not* require fancy banners. A potential danger of providing this simple facility is that some authors may be tempted to overdo their script decoration. Comments are important, but excessive decoration can detract from the aesthetics rather than enhance, and even with this package making them can waste a lot of time.

People who find the package useful but would like to suggest other tweaks or front-ends are welcome to contact the author at the email address given in the package itself.