using ESS with Emacs Org Mode

part of the ESS intro series

Greg Minshall

April 2, 2021

Contents

A file showing off some Org Mode and ESS features

Source blocks

Source blocks are lines of text, surrounded by a `#+begin_src` line before the beginning and a `#+end_src` line after the end of the code.

Source blocks can be named, and can have various *header arguments*. The name is on a `#+name` line. The header arguments can be on the

- on a `#+property:  header-args` line, to apply as defaults for all source blocks in a file

- inside a `/:PROPERTIES:/` drawer of a headline (node in the tree defined by the .org file), to apply as defaults for all source blocks in that subtree

- one the `#+begin_src` line that precedes the source block

- or, on separate `#+header` line, immediately preceding the `#+begin_src` line.

Here is an example .org file that shows those ways of defining various header arguments. The `#+header-args+` construct adds these arguments to the list of header args.

```
#+property: header-args :tangle ./nosuchdirectory/bigfile
```

```
** this is somewhere in this file
   :PROPERTIES:
   :header-args+: :tangle very/important/code.R
   :END:
```

```
now, code blocks will carry that name
```

```
#+header: :exports results
#+begin_src R :results value
   x <- "we want some code, and we want it now!"
#+end_src
```

Inline source blocks

```
There are also "inline" source blocks: src_R{"hello, world"} {{{results(=hello\, world=)}}}.
```

```
They have their own syntax for header arguments: src_R[:results
output]{cat("hello, again")} {{{results(=hello\, again=)}}}.
```

Evaluating source blocks

In order to enable evaluation of source blocks of R code, you will need to evaluate the following source block. (Emacs lisp is the only language whose evaluation is enabled by default.) To do this, position your cursor inside the source block (or on the `#+begin_src` or `#+end_src` line), and type [C-c C-c] (not the square brackets; just what is inside). Evaluating code in a file can be a security risk, so you will be prompted to make sure you want to execute the code.

```
#+name: set-allowed-languages
#+begin_src elisp :results none
  (org-babel-do-load-languages
   'org-babel-load-languages
   '((emacs-lisp . t) (R . t) (python . t)))
#+end_src
```

(Here, I named the source block mainly so that the confirmation message can provide you the name of the block for which it is asking permission to evaluate.)

If you customize the variable `org-babel-load-languages`, then those languages will be set to evaluate source blocks when you start Emacs. However, if you wish to add a language during execution, you will again need to call (`org-babel--do-load-languages`). Note that doing so will **not disable** any previously loaded language in this Emacs process.

An R source block, for example, can be evaluated using the same key strokes, [C-c C-c]:

```
#+name: first-r-block
#+begin_src R :results output
  cat("this block has been evaluated\n")
#+end_src

#+RESULTS: first-r-block
: this block has been evaluated
```

Evaluation via a `call` statement

Source blocks can also be evaluated by `call` statements in the .org file. Call statements have two forms, somewhat similar to the two forms of source blocks.

First, a `#+call` line can trigger evaluation of a source block. To trigger the triggering, you will need to type [C-c C-c] with point positioned on the call line.

```
#+call: first-r-block()

#+RESULTS:
: this block has been evaluated

Similarly, there is an inline call statement form which can trigger
evaluation of a source block.  here it is, inline!
call_first-r-block() {{{results(=this block has been evaluated=)}}}.
```

inline call statements have a syntax for block-specific header arguments similar to that of inline source blocks.

Exporting source blocks

Source blocks are, optionally, exported – to a .html or .pdf file, say – along with some of the rest of a .org file.

To start the export process, one types executes the command `M-x org-export-dispatch` (`[C-c C-e]`). Org mode then pops up a window that allows you to customize your export. At the top of this window are mostly options for defining the scope of the export. For example, if you then type `[C-s]`, only the subtree in which the export is initialized will be exported.

Then, one can choose the type of export: Org, Html, Latex (including PDF), Markdown, or Plain Text.

(The top and the lower parts talk about *publishing*, which accomplishes exporting in a slightly heavier way, requiring the parameters to be pre-set in the `org-publish-project-alist` variable. We won't discuss publishing here – you can read about it in the Org Mode manual – but it is, in fact, the way the .html and .pdf files associated with this tutorial have been produced.)

This code is based on a question to the R-help e-mail list. The goal was a simple way to find the peaks and valleys (highest and lowest values) in a vector of numbers.

```
#+name: peaks-and-valleys
#+begin_src R :exports both
  x <- c(1,0,0,0,2,2,3,4,0,1,1,0,5,5,5,0,1)
  (cumsum(rle(x)$lengths)-(rle(x)$length-1))[which(diff(diff(rle(x)$values)>=0)<0)+1]
  cumsum(rle(x)$lengths)[which(diff(diff(rle(x)$value)>0)>0)+1]
#+end_src
```

```
#+RESULTS: peaks-and-valleys
|  4 |
|  9 |
| 12 |
| 16 |
```

If you type [C-c C-e], then [C-s] (for "subtree"), then [h] (for "html"), and [o] (for "open"), then, assuming your Emacs is configured to know how to open a .html file in a browser (90% likely, I'd guess), you'll see the text in this subtree, the above code, and the indices of the valleys (the last result).

The `:exports both` is a header argument that tells Org Mode to export both the source code itself, as well as the results of its evaluation (you will be prompted to authorize the evaluation). One could also say one of: `:exports none`, `:exports code`, or `:exports results`.

Tangling code

Tangling is a term the literate programming world uses to mean extracting the source code from a "literate" document in a way that the code can then be compiled and/or executed. In Org Mode files, the `:tangle` header argument defines the filename to which a source block should be written. Like all header arguments, the `:tangle` header argument can be specified at the file, subtree, or individual source block level. All source blocks that have (or inherit) the same `:tangle` header argument will be tangled to the same file, in the order in which they appear in the .org file.

Repeating a source block had above,

```
#+name: peaks-and-valleys-tangling
#+begin_src R :exports both :tangle peaks-and-valleys.el
  x <- c(1,0,0,0,2,2,3,4,0,1,1,0,5,5,5,0,1)
  (cumsum(rle(x)$lengths)-(rle(x)$length-1))[which(diff(diff(rle(x)$values)>=0)<0)+1]
  cumsum(rle(x)$lengths)[which(diff(diff(rle(x)$value)>0)>0)+1]
#+end_src
```

```
#+RESULTS: peaks-and-valleys-tangling
| 4 |
| 9 |
```

```
| 12 |
| 16 |
```

Now, you could, if you chose, enter [`M-x org-babel-tangle`] ([`C-c C-v t`]). But, be warned! This will tangle all the source blocks in the current (this) .org file that have a `:tangle` header argument (specified or inherited). One can restrict the tangle operation to the source block at point (see [`M-x describe-function`] for `org-babel-tangle` for details), but even so, I would suggest being careful.

Editing a source block

The source block is just a number of text lines in text file. So, it is very normal to do minor edits in line, in the .org file. A certain amount of *code highlighting* (*fontification*, I guess, in Emacs parlance) is provided when editing source blocks inline.

On the other hand, by positioning the cursor on the source block and executing [`M-x org-edit-src-code`] ([`C-c '`]), one can "pop out" the source block and edit it in a buffer with the major mode set appropriately for the language of the source block. In the case of an R source block, the major mode will (the R-flavor of) ESS.

Feel free to experiment with the following code. On the R-help list, a new value for the vector x was proposed.

```
x <- c(1,1,1,2,2,3,4,4,4,5,6,6,6)
```

If you like, edit the source, change x, and evaluate it. What is its result? Is there a peak? A valley?

```
#+begin_src R
  x <- c(1,0,0,0,2,2,3,4,0,1,1,0,5,5,5,0,1)
  (cumsum(rle(x)$lengths)-(rle(x)$length-1))[which(diff(diff(rle(x)$values)>=0)<0)+1]
  cumsum(rle(x)$lengths)[which(diff(diff(rle(x)$value)>0)>0)+1]
#+end_src

#+RESULTS:
|  4 |
|  9 |
| 12 |
| 16 |
```

When editing, the "OrgSrc" minor mode is enabled. This mode provides binding for

[`C-x C-s`] to save the source edit back to the main .org buffer

[`C-c C'k`] to abort the current editing, discarding any unsaved changes

[`C-c '`] to save any unsaved changes back to the .org buffer and close the source edit buffer

NB: If you are editing a block that has `:noweb` references, the references will **not** be expanded when passing the block (in part or in whole) to a running iESS instance (via, e.g., [`C-c C-c`] or [`C-c C-b`]). And, in fact, the R interpreter will likely flag a syntax error should you try to do so.

(Some) source block header arguments

:noweb

Many (most?) programming languages have a way of "including" the "textual" contents of one source file in the compilation or execution phase of another source file. I think of `#include <stdio.h`, for example, in `C`. Literate programming defines a way of doing this known as "noweb".

In Org Mode, a <u>reference</u> to a previous block of code named `NAME` is denoted by `«NAME»`. The double angle brackets signal that this is a noweb reference. However, by default, noweb processing is **disabled** in an Org Mode buffer. The `:noweb` header argument defaults to `no`, but can be set to `yes` to enable noweb processing. Again, as with all header arguments, this can be done on a per-file, per-subtree, or per-(referring-)source block basis.

The noweb reference `«NAME»` can refer to either the name of another source block in the .org file:

```
#+name: a-noweb-name
#+begin_src R :noweb no
  cm <- "copy me!"
#+end_src

#+RESULTS: a-noweb-name
: copy me!
```

or with the `:noweb-ref` header argument:

```
#+begin_src R :noweb-ref another-noweb-name :noweb no
  cmt <- "copy me, too!"
#+end_src

#+RESULTS:
: copy me, too!

#+begin_src R :noweb yes :results output
  <<a-noweb-name>>
  <<another-noweb-name>>
  cat(cm, "\n", cmt, "\n", sep="")
#+end_src

#+RESULTS:
: copy me!
: copy me, too!
```

(If you edit the header of the previous source block by changing "yes" to "no" and then evaluate it, you will get an error.)

Note that, like the `:noweb` header argument, the `:noweb-ref` header argument can be set on a subtree basis. (I dread to think of what would happen were it to be set on a file basis. Okay, I have to try. Be right back... Not much happened. I suppose this is special-cased.)

:var

In Org Mode, the evaluation of source blocks can include initializing variables, using the `:var` header argument.

```
#+name: gives-pi
#+begin_src R :var pi=3.14
  pi
#+end_src

#+RESULTS: gives-pi
: 3.14
```

The left hand side of the argument to `:var` is the name of the variable as seen by the code inside the source block. The right hand side can be a constant (as above), or can designate the output of another source block in the .org file. In that case, the right hand side is the name (`#+name:`) of the source block providing the desired result.

If you evaluate the following source block, you will be asked to allow evaluation **twice**: once to produce the result from the **above** source block, and a second time to produce the result from the source block you are evaluating.

```
#+begin_src R :var pitoo=gives-pi :session R :results output
  cat("you were", pitoo/pi, "close!\n")
#+end_src
```

```
#+RESULTS:
: you were 0.999493 close!
```

The language of the source block providing the value of the variable does same as the language of the source block receiving the value. In general, though, there may be some adaptation required to mould the shape of the input value to that needed by the source code.

```
#+name: from-python
#+begin_src python :results value
  return 21
#+end_src
```

```
#+RESULTS: from-python
: 21
```

```
#+name: to-r
#+begin_src R :var howmany=from-python
  2*howmany
#+end_src
```

```
#+RESULTS: to-r
: 42
```

The name on the right hand side, instead of naming another source block in the file, can also name, e.g., an Org Mode table.

student	first exam	second exam	final
Greg	1	3	2
George	2	2	2
Linda	3	1	2
Georgia	4	4	4

In the following code block, the `:colnames` header argument is set to `yes` so that the first row is considered a row of column names.

```
#+begin_src R :var tbl=a-table :colnames yes :session R
  summary(tbl)
#+end_src
```

```
#+RESULTS:
| student          | first.exam   | second.exam  | final       |
|------------------+--------------+--------------+-------------|
```

```
| Length:4         | Min.    :1.00 | Min.    :1.00 | Min.    :2.0 |
| Class :character | 1st Qu.:1.75 | 1st Qu.:1.75 | 1st Qu.:2.0 |
| Mode  :character | Median :2.50 | Median :2.50 | Median :2.0 |
| nil              | Mean    :2.50 | Mean    :2.50 | Mean    :2.5 |
| nil              | 3rd Qu.:3.25 | 3rd Qu.:3.25 | 3rd Qu.:2.5 |
| nil              | Max.    :4.00 | Max.    :4.00 | Max.    :4.0 |
```

For R code, a second effect of setting `:colnames` to `yes` is that if the **result** is an R data frame, its column names (`colnames()`) will be preserved in the resulting Org Mode table

```
#+begin_src R :colnames yes
  mtcars[1:4,]
#+end_src
```

```
#+RESULTS:
| mpg  | cyl | disp | hp  | drat |   wt  | qsec  | vs | am | gear | carb |
|------+-----+------+-----+------+-------+-------+----+----+------+------|
|   21 |   6 |  160 | 110 |  3.9 |  2.62 | 16.46 |  0 |  1 |    4 |    4 |
|   21 |   6 |  160 | 110 |  3.9 | 2.875 | 17.02 |  0 |  1 |    4 |    4 |
| 22.8 |   4 |  108 |  93 | 3.85 |  2.32 | 18.61 |  1 |  1 |    4 |    1 |
| 21.4 |   6 |  258 | 110 | 3.08 | 3.215 | 19.44 |  1 |  0 |    3 |    1 |
```

```
#+begin_src R :colnames no
  mtcars[1:4,]
#+end_src
```

```
#+RESULTS:
|   21 | 6 | 160 | 110 |  3.9 |  2.62 | 16.46 | 0 | 1 | 4 | 4 |
|   21 | 6 | 160 | 110 |  3.9 | 2.875 | 17.02 | 0 | 1 | 4 | 4 |
| 22.8 | 4 | 108 |  93 | 3.85 |  2.32 | 18.61 | 1 | 1 | 4 | 1 |
| 21.4 | 6 | 258 | 110 | 3.08 | 3.215 | 19.44 | 1 | 0 | 3 | 1 |
```

:results

The `:results` header argument specifies many things. First, it specifies whether the results of an evaluation consist in the value "returned" by the evaluated source block, or by the output (to standard output) produced by the source block. The "value returned" means, in many programming languages (include R), the value of the last statement executed in the source block. You may have noticed examples of this use of `:results` in some of the previous source blocks. The values here are `value` and `output`.

```
#+name: usenowebhere
#+begin_src R :results none
  truepi <- pi
  cat(truepi, "\n", sep="")
  invisible(truepi*2)
#+end_src
```

```
#+begin_src R :results output
  <<usenowebhere>>
#+end_src
```

```
#+RESULTS:
: 3.141593
```

```
#+begin_src R :results value
  <<usenowebhere>>
#+end_src

#+RESULTS:
: 6.28318530717959
```

- Type of result

  Here, the question is what sort of Org structure should the returned value be considered to consist. The results can be interpreted to be an Org Mode table, list, or verbatim text. Or, the results can be output to a file.

  ```
  #+name: results-noweb
  #+begin_src R
    mtcars[c(1,12,13, 19),]
  #+end_src

  #+RESULTS: results-noweb
  |   21 | 6 |   160 | 110 |  3.9 |  2.62 | 16.46 | 0 | 1 | 4 | 4 |
  | 16.4 | 8 | 275.8 | 180 | 3.07 |  4.07 |  17.4 | 0 | 0 | 3 | 3 |
  | 17.3 | 8 | 275.8 | 180 | 3.07 |  3.73 |  17.6 | 0 | 0 | 3 | 3 |
  | 30.4 | 4 |  75.7 |  52 | 4.93 | 1.615 | 18.52 | 1 | 1 | 4 | 2 |
  ```

  If no result type is listed, Org Mode makes its pretty-good guess.

  ```
  #+begin_src R :results value
    <<results-noweb>>
  #+end_src

  #+RESULTS:
  |   21 | 6 |   160 | 110 |  3.9 |  2.62 | 16.46 | 0 | 1 | 4 | 4 |
  | 16.4 | 8 | 275.8 | 180 | 3.07 |  4.07 |  17.4 | 0 | 0 | 3 | 3 |
  | 17.3 | 8 | 275.8 | 180 | 3.07 |  3.73 |  17.6 | 0 | 0 | 3 | 3 |
  | 30.4 | 4 |  75.7 |  52 | 4.93 | 1.615 | 18.52 | 1 | 1 | 4 | 2 |
  ```

  But, to be sure, you can specify the type you would like. (Note the reapearance of `:colnames`.)

  ```
  #+begin_src R :results value table :colnames yes
    <<results-noweb>>
  #+end_src

  #+RESULTS:
  |  mpg | cyl |  disp |  hp | drat |    wt |  qsec | vs | am | gear | carb |
  |------+-----+-------+-----+------+-------+-------+----+----+------+------|
  |   21 |   6 |   160 | 110 |  3.9 |  2.62 | 16.46 |  0 |  1 |    4 |    4 |
  | 16.4 |   8 | 275.8 | 180 | 3.07 |  4.07 |  17.4 |  0 |  0 |    3 |    3 |
  | 17.3 |   8 | 275.8 | 180 | 3.07 |  3.73 |  17.6 |  0 |  0 |    3 |    3 |
  | 30.4 |   4 |  75.7 |  52 | 4.93 | 1.615 | 18.52 |  1 |  1 |    4 |    2 |
  ```

  `scalar` and `verbatim` are synonyms. They cause the value to be taken, well, at face value:

  ```
  #+begin_src R :results scalar
  ```

```
  <<results-noweb>>
#+end_src

#+RESULTS:
: 21 6 160 110 3.9 2.62 16.46 0 1 4 4
: 16.4 8 275.8 180 3.07 4.07 17.4 0 0 3 3
: 17.3 8 275.8 180 3.07 3.73 17.6 0 0 3 3
: 30.4 4 75.7 52 4.93 1.615 18.52 1 1 4 2
```

And, `list` converts the source block result into an Org Mode list.

```
#+begin_src R :results list
  <<results-noweb>>
#+end_src

#+RESULTS:
- (21 6 160 110 3.9 2.62 16.46 0 1 4 4)
- (16.4 8 275.8 180 3.07 4.07 17.4 0 0 3 3)
- (17.3 8 275.8 180 3.07 3.73 17.6 0 0 3 3)
- (30.4 4 75.7 52 4.93 1.615 18.52 1 1 4 2)
```

- How the results are formatted

  There are various ways Org Mode might decide to format the results, of, more or less, whatever type, before embedding them in the .org buffer.

  For example, to want the results to be wrapped in a source block, one can specify `code`:

```
#+begin_src R :results value code
  "<<results-noweb>>"
#+end_src

#+RESULTS:
#+begin_src R
mtcars[c(1,12,13, 19),]
#+end_src

#+RESULTS:
|   21 | 6 |   160 | 110 |  3.9 |  2.62 | 16.46 | 0 | 1 | 4 | 4 |
| 16.4 | 8 | 275.8 | 180 | 3.07 |  4.07 |  17.4 | 0 | 0 | 3 | 3 |
| 17.3 | 8 | 275.8 | 180 | 3.07 |  3.73 |  17.6 | 0 | 0 | 3 | 3 |
| 30.4 | 4 |  75.7 |  52 | 4.93 | 1.615 | 18.52 | 1 | 1 | 4 | 2 |
```

  This also shows that `noweb` syntax isn't defeated by, e.g., being embedded inside quotation marks. One can also specify that the results be embedded in an Org Mode "drawer" (which can be "closed", so the results don't show), or embedded in a source block of type "org", in blocks whose content will only be used when exporting to HTML, or to LaTeX (including PDF), etc.

  There is (apparently – writing this tutorial is a learning experience!) also a `:wrap` header argument, separate from the `:results` header argument, which specifies a value X, and the results are summarily wrapped in a `#+begin_X`...`#+end_X` block.

```
#+begin_src R :wrap foo
```

```
  3
#+end_src

#+RESULTS:
#+begin_foo
3
#+end_foo
```

- Handling of results

  Typically, the results of evaluating a source block are placed in the buffer following the source
  block, preceded by a line

  ```
  #+RESULTS: <NAME>
  ```

  where `<NAME>` is the name of the source block (if any).

  ```
  #+begin_src R
    Sys.time()
  #+end_src

  #+RESULTS:
  : 2021-04-02 09:56:08

  #+name: time
  #+begin_src R
    Sys.time()
  #+end_src

  #+RESULTS: time
  : 2021-04-02 09:56:08
  ```

  However, this behavior can be modified with the `:results` header argument. The relevant
  options are

  silent Don't change the buffer. The results are echoed in the minibuffer.

  replace This is the normal behavior; the `#+RESULTS:` block is replaced.

  append Each evaluation generates a separate `#+RESULTS:` block, which is placed after the
  previous result blocks.

  prepend Each evaluation generates a separate results block, which is placed before all pre-
  vious result blocks.

  Well, really, that is all pretty much just copied from the manual. Which, see.

:session

Normally, when Org Mode evaluates a source block, in instantiates a new **process** of the appropriate
type (the `R` command, for example), provides that process with the source from the source block
(expanding `:noweb` references), and lets it run.

   This ensures that each run starts from a "clean state". On the other hand, it means that
artefacts left behind by previous runs are not available to a future run. And, it can complicate
debugging.

   Thus, Org Mode provdies a `:session` header argument which names a buffer in which evalua-
tions of the current source block (or, all source blocks in the file, or all in a subtree) will occur.

During a given emacs instance, the first time you type [C-c C-c] in a source block with a given :session name, Emacs will prompt you for the name of the starting directory (with a default being the directory of the current file). It will then start up such a process, attach it to a buffer (i'm not totally sure of the Emacs terminology here) with the name provided as the argument to the :session header argument (or, use a language-specific default), and evaluate your code. This works for interpreted languages but not, to my knowledge, for compiled languages.

After evaluations, you can use [C-x b] to switch to that buffer and inspect the state, maybe set debug breakpoints, or debug after error or debug statement (browser(), say) in your code.

For R, the session buffer uses ESS[R] mode.

As I said above, the fact that future evaluations of the given source block, or of any other source block with the same :session argument, will run in the same R process is both a feature and a bug. On the bug side, you may end up developing code that only works with the accumulated state in the current buffer.

For example, say you are developing an R package and are testing it with some code that imports (require()) that package. And, say your test code points out an error in your package code, and that you fix your package code and re-build and re-install your package. Now, if you once again evaluate your test code, it will execute the require() statement but will do nothing as, as far as R is concerned, your package is **already** loaded. So, you will need to detach and unload your package which, if my memory serves me right

```
detach(package:PACKAGENAME, unload=TRUE)
```

or some such. **Then**, R will load the new version of your package.

Still, I find :session immensely valuable. If you find yourself wanting to use a debugger to debug your code, or to incrementally build up state as your are developing code, I recommend (carefully) using it.

:cache

```
#+begin_src R
1
#+end_src
```