Contents

A file showing off some Org Mode and ESS features

Introduction to Org Mode

> First, some notation :: A string inside square brackets, `[abecde]`, means you, the user, should type the string, leaving out the square brackets. Also, the Emacs convention for control and modifier (alt) keys is used. So, `[C-x b]` means type `control x`, then the letter `b`. And, `[M-x b]` would mean type `modifier x` (maybe better known as `alt x`), then the letter `b` (almost certainly an error!).

(To skip the introductory material and go straight to the demonstration work, click here.)

Emacs (wikipedia) Org Mode [1] sometimes exhorts one to do something like "Organize Your Life In Plain Text!"; the Org Mode manual, on the other hand, starts off by saying, "Org is an outliner". The philosophy of Org Mode (indeed, of Emacs, and maybe, to some extent, of any of the *nix operating systems) is that using non-proprietary file formats and software provides the most "liberating" and "horizon-free" way of taking advantage of modern information technology. And, to some extent, that a mostly-command line interface, rather than a graphical user interface, is also "the way to go".

I think Org Mode started off as a way of simplifying the creation of formatted documents, with tables, etc., for taking notes, creating agendas (items with date elements), and evolved into a much larger system of utilities for, for example, converting ("exporting", in org mode parlance) between the Org Mode syntax to .html, .pdf, etc., documents, with good support for doing mathematical (LaTeX) formatting.

Org Mode does a good job exporting to LaTeX, html. (Some people primarily use Org Mode as a more "user friendly" interface to LaTeX, though for serious documents, in the final analysis they probably spend a lot of time tweaking LaTeX, and org-to-LaTeX, configuration.)

Though I have used Org Mode for more than a decade, I know very little of most of its capabilities, as I mostly use it to centralize the source code within a project, as well as to produce the random document, especially documents with embedded code segments (known, in Org Mode, as "source blocks").

Org Mode is bundled into the main Emacs distribution, but a more-than-casual user might like to use the more-than-likely up-to-date package available via melpa (`[M-x package-list-packages]`, and regexp-search for /^..org /).

Babel is one name for talking about programming language support in Org Mode. You are able to embed source code inside of Org Mode buffers, edit these code blocks in a language-specific way, evaluate code blocks, pass the results of the evaulation of one code block to another code block, and include code blocks and/or the results of their evaulation in the document produced by exporting the org buffer. The rest of this document is dedicated to explaining these things.

Blocks of various sizes, shapes, colors

Various sorts of "blocks" are supported in Org Mode files. These start with `#begin_...`, and end with a corresponding `#end_...`, where the `...` are something like `src`, `example`, `quote`. A skeleton for some of these can conveniently be created by, in column 1 of an otherwise empty line, typing `[<X]` followed by `[tab]`, where `X` is one of

- `e` for an example block (line wrapping does not apply) (`#+begin_example`)
- `q` for a quote block (line wrapping applies) (`#+begin_quote`)
- `s` for a source block (`#+begin_src`)

[1] a.k.a., "org-mode", Org Mode, orgmode – the proliferation of notation makes googling somewhat of a challenge.

there are various other blocks; you can experiment by typing (in column one, of an otherwise empty line) `[<]` and then one of the following

- a
- c
- C
- E
- h
- l
- v

then hitting `[tab]`: and seeing what you get, looking in the manual for enlightenment. (Of course, one can always search the manual for `#[+]begin_`.)

In addition to the source blocks mentioned above (and elaborated on below), Org Mode allows short bits of "verbatim" code to be introduced by a colon as the first non-blank column of a line

```
: this is verbatim
```

One can have longer runs of verbatim ("example") data (which is not word wrapped, etc.:

```
#+name: ex
#+begin_example
this is a block
that
holds more verbatim text
#+end_example
```

As well as longer bits of text that **will** be wrapped ("quote"):

```
#+begin_quote
this is a bit of
text that should
show up, eventually, strung out in a smaller
number
of
lines when "filled"
(e.g., [M-x org-fill-paragraph] ([M-q]))
#+end_quote
```

Blocks can optionally have names. These names can be used as `:noweb` references, or to use the value of a block (or, if a source block, of the block's results) as input (a `:var`) to another block in the file.[2]

Drawers, Properties, Property Drawers

Org Mode drawers are a way of "closing away" information you normally won't want to see. These start with a line `:NAME:`, where `NAME` is a string, and end with a simple(r) `:end:`.

A property allows you to associate values with keys. Of particular interest to us here are property drawers, which combine the two concepts to provide a place to put parameters that customize the environment in which Org Mode processes your file (and, in particular, for our case, the context in which source code is evaluated).

On the other hand, if you are **not** looking at this on the web, and would like to see a webified version, with a bit more of an introduction to Org Mode, you can look here XXX

[2] I believe there are facilities for "naming" bits of information in other .org files, but I don't know the details and I don't know if it works with source code blocks.

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 by a `#+name` line which appears immediately before the `#+begin_src` line.

Source blocks can also have various *header arguments* that modify the context in which the source block executes.

(As mentioned above, a source block skeleton can be created by typing, in column 1, `<s`, and hitting `tab`.)

Inline source blocks

```
Source blocks can also be expressed using a =src_LANG= construct; a
=#+name:= line can be used to name the =src_LANG= block.  for example,
src_R{"hello, world"} {{{results(=hello\, world=)}}}.
```

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

And, the results of an inline source block can be used as input (via :var) to another source block.

```
#+name: whyo
src_R{"42"} {{{results(=42=)}}}
```

```
#+begin_src R :var x=whyo :exports results
x
#+end_src

#+RESULTS:
| 42 |
|    |
```

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 `[M-x org-babel-execute-src-block]` (`[C-c C-c]`) (not the square brackets; just what is inside). Additionally, 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 command, `[M-x org-babel-execute-src-block]` (`[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 `[M-x org-babel-execute-src-block]` (`[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.)

The following 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(1,0,0,0,2,2,3,4,0,1,1,1,0,5,5,5,0,1)
  (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 `M-x org-export-dispatch` (`[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 :noweb yes
  <<peaks-and-valleys>>
#+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]` (`[C-h f]`) 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? [3]

[3] One solution was proposed here.

```
#+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

`[M-x org-edit-src-save]` (`[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

`[M-x org-edit-src-exit]` (`[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., `[M-x org-babel-execute-src-block]` (`[C-c C-c]`) or `[M-x ess-eval-buffer]` (`[C-c C-b]`)). And, in fact, the R interpreter will likely flag a syntax error should you try to do so.

Header arguments

Header arguments modify the execution context in which Org Mode source blocks are evaluated and/or in which the results of those evaluations are processed.

Header arguments can be specified on, and apply to

- 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
- the `#+begin_src` line that precedes the source block
- separate `#+header` line(s), immediately preceding the `#+begin_src` line.

Here is an example from a .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
```

The following are some of the header arguments I most often use.

`:colnames and :rownames`

When processing a table result of a previous source block to be used as an input variable to another code block, the `:colnames` header argument instructs Org Mode how to interpret the first row: should it be assumed to be a row of column names, a row of data, or should Org Mode use a heuristic to make that determination.

Similarly, Org Mode uses `:rownames` to determine whether the first column of each row should be considered as a name for that row, or simply as the data of the first column of that row.

Also, when processing the **results** of a code block, these header arguments tell some Org Mode language support routines how to deal with column and row names of the returned result. If left off, the Org Mode result of evaluation will drop any language-specific column and row names. However, if these header arguments are set to `yes`, language-specific column and row names will be included in the results placed in the Org Mode buffer.

```
#+BEGIN_SRC R
mtcars[1:3,]
#+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 |

#+begin_src R :colnames yes :rownames yes
mtcars[1:3,]
#+end_src

#+RESULTS:
|               |  mpg | cyl | disp |  hp | drat |    wt |  qsec | vs | am | gear | carb |
|---------------+------+-----+------+-----+------+-------+-------+----+----+------+------|
| Mazda RX4     |   21 |   6 |  160 | 110 |  3.9 |  2.62 | 16.46 |  0 |  1 |    4 |    4 |
| Mazda RX4 Wag |   21 |   6 |  160 | 110 |  3.9 | 2.875 | 17.02 |  0 |  1 |    4 |    4 |
| Datsun 710    | 22.8 |   4 |  108 |  93 | 3.85 |  2.32 | 18.61 |  1 |  1 |    4 |    1 |
```

`: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's implementation, inside a source block, a reference to a previous block of code named `NAME` is denoted by `«NAME»`. The double angle brackets signal that this is a noweb reference.[4] 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
```

[4] Outside of a source block, `«NAME»` is used to tag an internal link in the .org file; the reader may have noticed instances of this in this document.

```
#+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 as R tries to parse a line that begins with «.)

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, or blank.

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

#+RESULTS:
: 2021-04-03 19:05:52

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

#+RESULTS: time
: 2021-04-03 19:05:52
```

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 previous 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 in cases where that might be desirable. And, it can complicate debugging.

Thus, Org Mode provdies a `:session` header argument which names a buffer (that Org Mode will create, if it doesn't already exist) in which evaluations 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 `[M-x org-babel-execute-src-block]` (`[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.

```
#+begin_src R :session R-session-name :exports code :eval never-export :results none
x <- 1
#+end_src

#+begin_src R :session R-session-name :eval never-export :exports code
x
#+end_src
```

If you evaluate the above two blocks in order, the second will return as a result the value of `x` set by the first.

After evaluations, you can use `[M-x switch-to-buffer]` (`[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 well as in the OrgSrc minor 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 the next time you or your code `require's` it.

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

`:cache`

Sometimes you will have a long-running computation, the results of which are used as input to other computations, or for export. In this case, you may not want to have to re-run the computation if its results won't change. To help with this situation, Org Mode has the `:cache` header argument.

If a source block is marked with `:cache yes` is scheduled to be evaluated, Org Mode checks the following:

- the contents of the source block itself
- the header arguments to the source block
- the values of the sources of all the `:var` header arguments

If none of these have changed (as testified by a cryptographic hash over these) since the last evaluation of the source block, then the value of the previous evaluation is used as the value of the current evaluation.

Of course, if a source code block takes input from some external source, such as a file, that may have changed since the previous evaluation, Org Mode won't notice, and you may get incorrect results. Also, a source code block that makes use of the results of a random number generator will not be re-evaluated. It is the responsibility of the programmer to **not** mark source blocks of these types with `:cache yes`.

Here is an example. When it is actually evaluated, this source code block will block for two seconds. If you evaluate it a **second** time, the results will appear immediately.

```
#+name: long-computation
#+begin_src R :cache yes
  Sys.sleep(2)
  42
#+end_src
```

```
#+RESULTS[1850a21ec1cef57306973138b0c17881d00f9820]: long-computation
: 42
```

Now, we look at how `:cache` deals with one of the inputs changing. First, here is a source which produces the same value each time it is run:

```
#+name: stable
#+begin_src R
  23
#+end_src

#+RESULTS: stable
: 23
```

Then, a source that produces a different value each second:

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

#+RESULTS: unstable
: 2021-04-03 19:05:55
```

Then, two functions. The first uses results of the unvarying source code block as input. The first time you evaluate it with `[M-x org-babel-execute-src-block]` (`[C-c C-c]`), it will sleep the two seconds. However, after that, evaluating it will return results immediately.

```
#+begin_src R :var input=stable :cache yes
  Sys.sleep(2)
  input
#+end_src

#+RESULTS[1de9d77db24319fea2fea80b19d13eb552cda0f3]:
: 23
```

This function, on the other hand, takes as input results from the varying function. Each time (well, separated by at least one second) you evaluate it, it will take the full two seconds.

```
#+begin_src R :var input=unstable :cache yes
  Sys.sleep(2)
  input
#+end_src

#+RESULTS[8ab38ec3113c12ec5617d73decc1a6eecbbfc9d9]:
: 2021-04-03 19:05:57
```

Other sources of information

The official Quick Start guide provides a very good introduction to Org Mode. The main Org Mode web page is another good source for further information, as is the Org Mode worg site. (You may find it useful to take a linear stroll through the worg site map.) Also, once installed on your system, the Org Mode info pages are available in Emacs (or, using `info(1)`).