Contents

using ESS with emacs org mode

intro to org mode

basics: document structuring, tables, math/LaTeX, exporting

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.

a .org file is structured like an outline, with outline sections and levels indicated by lines starting with one or more asterisks in the first column; these define "headlines" for the outline. between headlines one can include text, tables, bits of code, etc. bits of the .org file can be displayed and/or hidden by positioning the cursor on a headline and (possibly repeatedly) hitting `tab`.

asterisk-defined headlines can include "tags" (`:essintro:`; i am ignorant of their use), and can optionally be (immediately) followed by a special "drawer" [2] named `:PROPERTIES:` holding, well, properties, i.e, (key, value) tuples.

Org mode text markup allows the use of **asterisks**, `equal signs`, `tildes`, *[forward] slashes*, underscores and (though, apparently, deprecated) ~~plus signs~~, to achieve different font characteristics. (until writing this, i have always used equal signs for short bits of code, but now i will try to remember to try out tildes.) in-line math formatting `\(x = cos(y)\)` is supported;

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

[2]a structure starting with an more-or-less arbitrary name, surrouned by colons, and ending with `:END:`

Org mode also allows super- (10^9) and subscripts (I_e) (though these can be disabled, if desired).

various sorts of "blocks" are supported in org mode files. these typically start with `#begin_...`, and end with a corresponding `#end_...`, where the `...` are something like `src`, `example`, `quote`. a skeleton for one 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

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

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

```
a, c, C, E, h, l, v
```

seeing what you get, looking in the manual. (of course, one can always search the manual for `#[+]begin_`.)

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.)

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

#+begin_src R :var foo=ex :exports both
```

```
foo
#+end_src

#+RESULTS:
| this is a block          |
| that                     |
| holds more verbatim text |

#+begin_src R :var foo=ex :exports code
foo
#+end_src

#+RESULTS:
| this is a block          |
| that                     |
| holds more verbatim text |
```

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 variable) to another block in the file.[3]

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"). a good source for further information, in addition to the main org mode web

[3]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.

page, is the org mode worg site. also, once installed on your system, the org mode info pages are available in emacs (or, using the info).

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 (in emacs, use `M-x package-list-packages`, and regexp-search for /$^{..org}$ /).

babel – programming language support in org files

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, execute code blocks, pass the results of the execution of one code block to another code block, and include code blocks and/or the results of their execution in the document produced by exporting the org buffer.

literate programming, if you want

source blocks

Org mode source blocks look like this

```
#+begin_src R
"hello, world!"
#+end_src

#+RESULTS:
: hello, world!
```

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

Org mode source blocks can be edited, either inline in the org mode (".org") document buffer, or in a separate buffer (i call these "Org Src…" buffers, but they might also be known as "sub-edit buffers" or "source edit buffers) that you can "pop up" from the .org buffer ("pop out of the .org buffer"?), normally with `C-c '`. in the latter, there is better support for emacs so-called "font locks", which do program language-specific source code highlighting, etc.

once written, a source block can be executed, returning results, either the output of the code, or some terminating "value" of the code, as an element of the .org file. and, the blocks can be executed manually, or – and optionally, block by block – while exporting a .org file to a different format.

a source block can be executed in one of two sorts of contexts, within a "session", or outside of a "session". a "session" here means some process that retains state between executions of (possibly different) source blocks. on the other hand, a "non-session" starts up with no internal [4] state from prior runs.

source blocks can also be expressed using a `src_LANG` construct; a `#+name:` line can be used to name the `src_LANG` block.

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

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

#+RESULTS:
: 42
```

finally, and of particular interest in these tutorials, is the fact that org mode uses ESS to provide R language support.

worg page on R and org-mode.

naming blocks

there are two ways of naming blocks. first, an individual block itself can be preceded by a `#+name:` line

```
#+name: somecode
#+begin_src R :results output
cat("this is *some* code!\n")
#+end_src

#+RESULTS: somecode
: this is *some* code!
```

[4] obviously, prior runs may have, e.g., changed the state of the file system on which all these executions are running; that is, the **external** state may by influenced by what has previously been executed.

```
#+name: someothercode
#+begin_src R :results value
whynot <- "this is some *other* code!"
#+end_src

#+RESULTS: someothercode
: this is some *other* code!
```

second, a PROPERIES drawer, using the `header-args` attribute `noweb-ref` to name the (otherwise unnamed, i believe) code blocks in that branch.

```
#+property: header-args :noweb yes

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

now, code blocks will carry that name
#+begin_src R
x <- "we want some code, and we want it now!"
#+end_src

#+RESULTS:
: we want some code, and we want it now!

** somewhere else
#+begin_src R :results output

cat(x, "\n")
#+end_src

#+RESULTS:
: we want some code, and we want it now!
```

as you can see, the properties drawer can carry many of properties, including the name of a destination file for tangling. the very odd `«nowcode»` is our next topic: noweb.

noweb

noweb is a literate programming syntax to allow referencing blocks of code within some larger (.org file, say) context. it allows a programmer to re-use bits of code (sort of `#include` like).

noweb syntax is **disabled** by default. to enable noweb syntax, one can either enable it on the header line of each source code block, enable it in a properties drawer, or, as in the following, enable it once at the beginning of an org file:

```
#+property: header-args  :noweb yes
```

once enabled, references of the kind `«NAME»` will incorporate the contents of a previous source code block.

```
#+property: header-args :noweb yes

#+name: fubar
#+begin_src R
"this is an example"
#+end_src

#+RESULTS: fubar
: this is an example

#+begin_src R

#+end_src

#+RESULTS:
: this is an example
```

tangling

while often we are content to execute code blocks inside the .org file, equally often we might want to export some or all of the code blocks for execution (or inspection) outside of the .org file. for example, we might want to use some of the code in the .org file to create an R package.

in the world of literate programming, *tangling* is the process of extracting source code from a (theoretically primarily text) document. in org mode, one uses `org-babel-tangle` (normally bound to `C-c C-v t`) to tangle the source code blocks in a file. the file to which a source block will be tangled is specified in the `:tangle` attribute, placed on the `#+src_block` line or in a `:header-args` line (in a properties drawer, to apply to a subtree of the .org file, or stand-alone as above to apply to the entire .org file).

to actually *tangle* a file, use `M-x org-babel-tangle`, often bound to `C-c C-v t`.

executing

`C-c C-c` on a source block

`C-c C-c` on an inline source block

`C-c C-c` on a #+call

`C-c C-c` on an inline-call

```
#+name: find-orgs
#+begin_src R
1
#+end_src

#+RESULTS: find-orgs
: 1

call_find-orgs() {{{results(=1=)}}}

#+call: find-orgs()

#+RESULTS:
: 1

call_find-orgs() {{{results(=1=)}}}
```

:colnames and :rownames

```
#+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 | ca
|---------------+------+-----+------+-----+------+-------+-------+----+----+------+----
| Mazda RX4     |   21 |   6 |  160 | 110 |  3.9 |  2.62 | 16.46 |  0 |  1 |    4 |
| Mazda RX4 Wag |   21 |   6 |  160 | 110 |  3.9 | 2.875 | 17.02 |  0 |  1 |    4 |
| Datsun 710    | 22.8 |   4 |  108 |  93 | 3.85 |  2.32 | 18.61 |  1 |  1 |    4 |
```

results

```
#+name: somecode
#+begin_src R :results output
cat("this is *some* code!\n")
#+end_src

#+RESULTS: somecode
: this is *some* code!

#+name: someothercode
#+begin_src R :results value
whynot <- "this is some *other* code!"
#+end_src

#+RESULTS: someothercode
: this is some *other* code!
```

variables

– in and out

exporting

entire document or a subtree of the document

life in Org Src buffers

org-mode community

mailing lists

other tutorials, etc.

Rainer's screencasts about Org mode (now a course on Udemy)
tutorial on R and org-mode

```
From: Erik Iverson <erikriverson@gmail.com>
Date: Tue, 23 Feb 2021 12:30:03 -0800
Subject: Re: org-in-org
To: Greg Minshall <minshall@umich.edu>
Cc: emacs-orgmode <emacs-orgmode@gnu.org>
```

`https://raw.githubusercontent.com/vikasrawal/orgpaper/master/orgpapers.org` or `https://github.com/vikasrawal/orgpaper/blob/master/orgpapers.org` or, more recently:

```
From: Jeremie Juste <jeremiejuste@gmail.com>
To: Greg Minshall <minshall@umich.edu>
Subject: Re: org-in-org
Date: Tue, 23 Feb 2021 22:38:06 +0100
Cc: emacs-orgmode@gnu.org
```

code blocks

there are a few pieces of information org-mode needs to define a block of code

- (optionally) a `name` for the block (to use to include the block's code in another block with noweb, or to use the block's results as input to another block with :var.
- the language of the code in the block
- various arguments (`header arguments` and `switches`) that define how the code interacts with its environment

- the source code itself

there are at least two ways of encoding the needed information:

a source block

the most "normal" way of defining source code is with a *source block*.

```
#+begin_src R
  "hello, world"
#+end_src

#+RESULTS:
: hello, world
```

we use this form below in discussing the structure of a code block.

an *inline* code block

in a second form, known as an *inline code block*, the entire block can fit on one line (though multiple lines are possible).

```
src_R{"hello, world"} {{{results(=hello\, world=)}}}
```

i don't use this form. for this tutorial i looked at it briefly. it appears its semantics are different from that of source blocks. other than that, i will not discuss it further.

the anatomy of a source block

```
#+name: refid
#+begin_example
this is a test
#+end_example

#+name: anatomy
#+begin_src R :var varname=refid :results output
cat(varname)
#+end_src

#+RESULTS: anatomy
: this is a test
```

```
#+header: :var anothername=anatomy
#+header: :exports results
#+header: :results value
#+name: second
#+begin_src R
anothername
#+end_src

#+RESULTS: second
: this is a test
```

the block named *refid* is not a code block, but shows how the contents of another block (verbatim, in this case) can be used as input to a code block.

we set *anatomy* as the name of the first code block, using the `#+name` line.

then, we declare *anatomy's* source block with the `#+begin_src` line, which has

- the language (`R`, of course)
- a header argument specifying an input variable named *varname*, using the value of *refid* block
- another header argument specifying that the result of this block will be taken from whatever it prints on stdout

the code for *anatomy* just consists of printing the (input) variable *varname* to stdout.

we continue by defining yet another source block, which we name *second*. it shows a different way of specifying the header arguments, via `#+header` lines, each of which can define one or more header arguments. one can put some header arguments on the `#+begin_src` line, some on one or or more `#+header` lines.

- :exports code
- :results table
- :colnames yes
- :exports none

- :tangle no
- :results none
- :var csvsedtable=csvsedtable
- :results output verbatim
- :cache