Note Taking Manual

Ryan G

February 11, 2020

Contents

Summary

Basically take all your notes in Markdown with `vim`, bigger documents use `org-mode`:

- Find Notes
  - Searching
    - using `ripgrep` and =fzf
  - Notebooks
    - use `YAML` headers and the `tags` property to denote notebooks just like Notable does
      - Manipulate these `YAML` headers with ***R***, `ripgrep` and `fzf`
  - Tags
    - use `#tags` throughout the document to tag things
      - Retrieve and search these tags using `ripgrep` and `fzf`
- Browse Notes
  - Use `fzf`, `ripgrep` and the vim `FZF` plugin.
  - Use **Notable**
  - Use VSCode with the VSNotes and Nested Tags plugins
  - If you use MarkText, Zettlr or Typora in conjunction with tmsu and ranger (with the glow preview) you can really start to feel like this is a full-featured solution)
- Share notes
  - This ones easy, do it from vim:

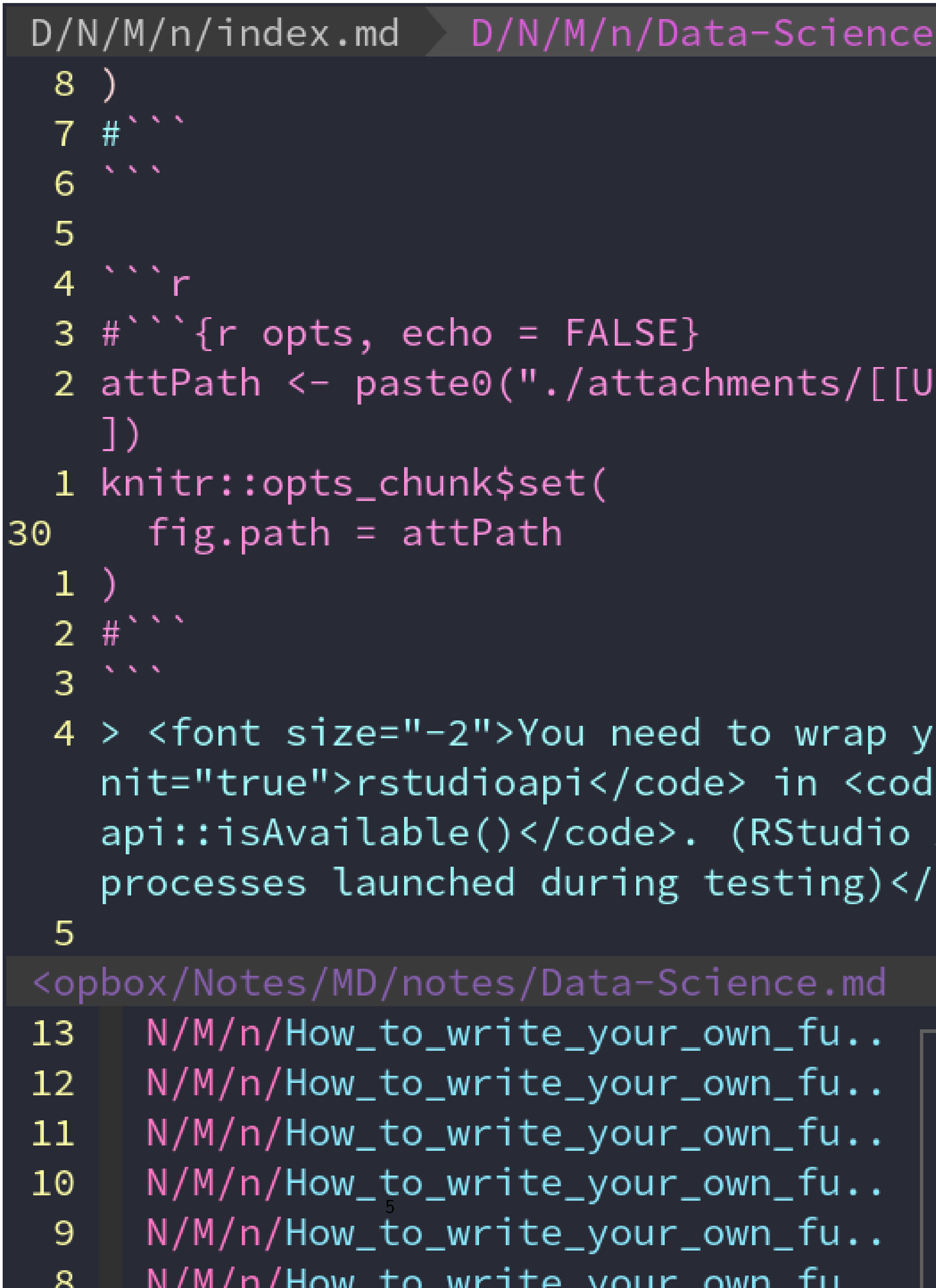
```
alias xclip='xclip -selection clipboard'
xclip -o | pandoc -f md --mathjax -s -t html -c
 ↪  ~/Templates/CSS_/github-pandoc.css | xclip
echo "<stle
```

  - PDF's are a little stickier because you're gonna wanna use the listings package (or rather the minted package:

```
alias xclip='xclip -selection clipboard'
echo "<style>" > /tmp/mystyle.css
cat ~/Templates/CSS_/github-pandoc.css >> /tmp/mystyle.css
echo "</style>" >> /tmp/mystyle.css
input=$(xclip -o)
pandoc -s --self-contained $input --listings --toc -H
 ↪  /tmp/mystyle.css --pdf-engine-opt=-shell-escape -o
 ↪  /tmp/output.pdf
echo /tmp/output.pdf | xclip
echo "Path is in clipboard"
```

– I can't figure out (an easy way) to get it in the minted package TBH, I recommend going from MD to Org to LaTeX or MD to LaTeX and manually.

When you've got this all set up you should have something that looks like this:



~~~
D/N/M/n/index.md  D/N/M/n/Data-Science
)
#```
```

```r
#```{r opts, echo = FALSE}
attPath <- paste0("./attachments/[[U
])
knitr::opts_chunk$set(
    fig.path = attPath
)
#```
```
> <font size="-2">You need to wrap y
nit="true">rstudioapi</code> in <cod
api::isAvailable()</code>. (RStudio
processes launched during testing)</

<opbox/Notes/MD/notes/Data-Science.md
  N/M/n/How_to_write_your_own_fu..
  N/M/n/How_to_write_your_own_fu..
  N/M/n/How_to_write_your_own_fu..
  N/M/n/How_to_write_your_own_fu..
  N/M/n/How_to_write_your_own_fu..
  N/M/n/How_to_write_your_own_fu
~~~

Why?

`OneNote` and `Evernote` aren't on linux (I did try NixNote but it failed to sync far to often, I may revisit it in time though because sharing HTML through evernote was a lot easier than deploying `gh-pages`.

Why not solely use org-mode

Because `emacs` is slow, ***R***-Markdown is really easy and iOS MD apps are nice, it doesn't really matter though because it's easy to convert between `org-mode` and *Markdown*:
I really like this simple solution but `org-mode` has it's place and it's already all implemented, I find myself using it more and more (this manual is in `org-mode`

```
  alias xclip='xclip -selection clipboard'
# To Org
  xclip -o |  pandoc -s -f markdown -t org | xclip -selelection
   ↪  clipboard
# Back to Markdown
  xclip -o |  pandoc -s -t markdown -f org | xclip -selection clipboard
```

Actually with pipes you can do all sort of cool things, like this is an example of how to pull a website into markdown:

```
curl https://orgmode.org/manual/Markdown-Export.html#Markdown-Export |
 ↪  pandoc -f HTML -t gfm | xclip -selection clipboard
```

and reference it using `beautifulsoup`:

```
    arglink=$(xclip -o -selection clipboard)
  title=$(wget -qO- $arglink |
  perl -l -0777 -ne 'print $1 if /<title.*?>\s*(.*?)\s*<\/title/si' |
  recode html..)
  outputlink="[$title]($arglink)"
echo $outputlink | xclip -selection clipboard
```

1. Problems with links Links may become a pain in the ass to maintain, so what i do is copy the file name to the clipboard and use fzf with this bash script to make a relative link to a file:

```bash
#!/bin/bash
# Don't forget to adjust the permissions with:
#chmod +x ~/somecrazyfolder/script1

## Program


### Description
# This will use fzf to find filenames that might correspond to a path from a broken link in the clipboard.

# so let's say that ~[](~/broken/path/to/rsico.png)~ is a broken link, I can use ~yi(~ to copy that to the clipboard in vim  and run the following bash scipt to return the correct link:

# Requires:
  # * gnu coreutils (specifically realpath)
  # * fzf
  # * xclip

### Code
brokenPath=$(xclip -o -selection clipboard)
#find ~/Dropbox/ -name $(echo $(basename $brokenPath)) | fzf | xclip -selection clipboard
NewFile=$(find ~/Dropbox/ -name $(echo $(basename $brokenPath)) | fzf)
echo $NewFile | xclip -selection clipboard

echo "
Put the path of the source file in the clipboard and Press any Key to Continue
"

# this will just continue after a key stroke
read -d'' -s -n1

echo "
Using:

"

sourceFile=$(xclip -o -selection clipboard)

echo "
SOURCE_FILE.......$sourceFile
NEW_ATTACHMENT....$NewFile
"


sourcePath=$(dirname $sourceFile)

relativePath=$(realpath --relative-to=$sourcePath $NewFile)
```

Sharing Material

Markdown

- HackMD is an option
- *MkDocs* and GitHub

MkDocs

when using MkDocs, make a seperate `docs` or `PUBLIC` directory like this:

```
.
 Notes
     MD
         attachments
         notes
         PUBLIC
```

when you want to syncronse the two directories before building or serving, do something like this:

```
for i in $(ls PUBLIC/*; do
    cp notes/$i PUBLIC/
done
```

Be aware though, if it's on GitHub it's not really private, instead you should use `org-mode` and `PGP` if you want to put personal journal/note-to-self stuff in the same folder.

HTML and iOS

- Shortcuts
  - iCloud is quicker
  - Dropbox allows mathjax
  - WorkingCopy (best)
    * phenomenal, uses mathml for math in `MD`
    * uses MathJax to view Math in HTML

Pandoc PDF and HTML

PDF

In order to Export to PDF put something like this in your `.vimrc`:

```
nmap <Leader>lo :!pandoc -s --self-contained "%" --listings --toc -H /home/ryan/Dropbox/profiles/Templates/LaTeX/ScreenStyle.sty --pdf-engine-opt=-shell-escape -o /tmp/note.pdf ; xdg-open /tmp/note.pdf
```

If you don't have code in the document I recommend using a `.sty` with `\twocolumn` because then you can read things on your phone.

HTML

HTML is really much the same, just remember to use `--mathjax` (or `mathml`) and to use the `-H` option for the style sheet so it's still self-contained.
If you have pictures this won't work so instead use one of these two solutions:

```
" Why doesn't this work?
"nmap <Leader>meeho :!pandoc -s --self-contained "%"  --toc -H Dropbox/profiles/Templates/mathjax ~/Templates/CSS/gitOrgWrapped.css  -o /tmp/note.html ;  xdg-open /tmp/note.html
" These do work thoughVV
nmap <Leader>meeho :!pandoc -s --self-contained "%"  --toc -H ~/Templates/CSS/gitOrgWrapped.css  -o /tmp/note.html ; cat ~/Templates/mathjax >> /tmp/note.html; xdg-open /tmp/note.html

nmap <Leader>meeho :!pandoc -s --self-contained "%" --mathml --toc -H ~/Templates/CSS/gitOrgWrapped.css  -o /tmp/note.html ;  xdg-open /tmp/note.html
```

I like `mathml` in this use case because it's robust, it is ugly though so if you want to use `mathjax` just make sure you make a `self-contained` document first and then tack the mathjax on later.

LaTeX and PDF Export   LaTeX

It is extremely important not to have filenames with whitespace characters, this WILL break LaTeX exports when using the minted package owing (I believe :shrug:) to the `-shell-escape` flag.
Heading dept is another common issue, anything equal to or deeper than `**** Heading 4` will be a list item, LaTeX list items are limited to a depth of I think 4, unless you use the `\usepackage{enumitem}` package, so if you make deep headlines followed by deep lists be aware that it will turn out odd in LaTeX and potentially fail.
If you have added the `enumitem` to your LaTeX style, and `org-mode` still won't compile the LaTeX, just open it in vim and use VimTeX to compile it, I've had success with the latter when the prior fails and I haven't cared to investigate why.
ďűčľţţ ś

Tips Tricks Misc

Using QR Codes

Best way to share a link to a mobile, by far, is using qrcodes, without a doubt, you can generate these on mobile using a Pythonista script. On *nix you can use `qrencode`: [1]

```
qrencode -t UTF8 $(xclip -o -selection clipboard)
```

Writing Material

TODO Diagrams

Apple Pencil

TiKz

Tikz is awesome but this is actually a bit of work to implement:
See Previous work on Including Tikz Plots

1. write in LaTeX
2. export using a luascript
3. grab the svg
   (a) be aware that getting an svg in LaTeX is a real pain, so you can go from TeX to HTML with the lua script but not really back, its madness.
   (b) Or you could go from `svg` to `png` but then you lose scaling so the whole thing sucks
   (c) You could also use LaTeX and just export to HTML with Mathjax This is probably the way to go tbh.
      i. You can open the HTML from notable :shrug:

InkScape

Working With *R*

Basically you want to be working entirely with NVim-R, all from within vim you can work with ***R*** and ***R***-*Markdown* files, it's just phenomenal, of course to get a full featured experience out of it you're going to need the following vim plugins:

[1] qrencode(1) - Linux man page

- CSV Viewer
- Copy/Paste Images
- NCM-R (works with Deoplete)
- ale (for Linting)

AutoCompletion

AutoCompletion Stuff, you can use Deoplete OR NCM2, NCM2 is better because it supports %>% piping, but, you will have to manually decide what completion you want, you can refer to my list in the appendix if you want work.

- NCM-R and NCM2 (AutoCompletion for Nvim-R)
  - Nvim-R comes with omni-completion in the box, if you use NCM2 as an AutoCompletion engine, NCM-R acts as a more complete source that works with %>% as well. (this is incompatible with *Deoplete*.
    * If you do switch to `ncm2`, for python, make sure you have jedi and neovim support, so do something like this:

      ```
      pip install --user neovim
      pip install --user jedi
      pip3 install --user neovim
      pip3 install --user jedi
      ```

      Although `ncm2` with `jedi` uses `py3` just do both to be sure.
  - NCM2-UltiSnips Snippet Integration
    * You might want to change the defaults if you don't like using `<C-n><C-p>`, I'm used to it, though `TAB` isn't uncommon:

    ```
    " First use tab and shift tab to browse the popup menu and
      " use enter to expand:
    inoremap ncm2_ultisnips#expand_or("<CR>, 'n')
    inoremap pumvisible() ? "<C-n>" : "<Tab>"
    inoremap pumvisible() ? "<C-p>" : "<S-Tab>
    "" However the previous lines alone wont work, we must
      " disable the UltiSnips Expand Trigger, I set it to
     ctrl-0let g:UltiSnipsExpandTrigger="<c-0>"
    ```

    * Use the Default Snippets here
- Deoplete (Completion Engine)
  - If you have the patience you could also try YCM but I found it a PITA to keep maintained across systems.

  - This comes with no support for ***R***, instead you have to allow it to grab stuff from the `omni-completion`, this isn't quite ideal and `NCM2` comes with more features, but you could:

```
"""""" Deoplete
" This might Conflict with NCM2 so dont use it for R?
if has('nvim')
  Plug 'Shougo/deoplete.nvim', { 'do': ':UpdateRemotePlugins' }
else
  Plug 'Shougo/deoplete.nvim'
  Plug 'roxma/nvim-yarp'
  Plug 'roxma/vim-hug-neovim-rpc'
endif
let g:deoplete#enable_at_startup = 1
call deoplete#custom#option('omni_patterns', {
    \ 'r': '[^. *\t]\.\w*',
    \})
```

Snippets

- UltiSnips (Main Tool)

Citations

Citation Stuff, basically the way these work is by using Unite to open a buffer of bibtex keys fed to it by `citation.vim` and then ZotCite will implement a pandoc filter to get it looking right, if you can get it to work (at the very least getting the keys in there is half the battle anyway).

- Zotero
  - Do not use the snap, due to sandboxing it will not integrate with bibtex and so is game-breaking.
- ZotCite OmniComplete AutoCompletion
  - make sure to `pip3 install pybtex`
- Unite (Slow FZF)
  - this works for anything, but, in particular it works for citations.
  - Citation Source for Unite that imports your `references.bib`
  - Citation.vim feeds it BibTeX

There is a good writeup on FreeCodeCamp and a follow up on Medium that will walk you through setting it up, but basically using this will allow you, all inside vim, to:

- Live preview markdown using iamcco's Live-Preview plugin
- Enter math using UltiSnips and *Gilles Castel's* Snippets by using `nnoremap <C-x><C-t><C-t> :setlocal filetype=tex<CR>`

  - switching back to ***R*** mode is trivial with `:  w | e!  <cr>`
  - Math will be previewed live

- Embedding Results as Comments
- Copy Paste from Help without Touching the mouse
- Get OmniCompletion with =<C-x><C-o>
- context sensitive knitting `SPC k R`
  - for knitting Script
  - for knitting rMarkdown
- Using Sympy or WolframScript within the document to solve LaTeX
- You can use `SPC k n` to turn the RMD into an MD, a potential workflow could be:
  - Knit to MD with `SPC k n`
  - open the file and copy it to the clipboard
  - go to the index with `SPC w w` and make a link to a new note
  - use `C-c C-y` to embed YAML Tags as detailed Below at 1.

  w= to make a new file in your index

Other Options

Things that I haven't looked into but have GitHub Stars are:

- GitHub - metakirby5/codi.vim: The interactive scratchpad for hackers.
  - I think this is like a live REPL, it looks awesome
  - I have since tried this, it's awesome, just make sure you `:Commands` and then `AutoSaveToggle`
  - Super handy if you do `:setfiletype=py` or `.R` or whatever and then `:Codi`, it's like having a live REPL to copy out of in an `md` file, brilliant.
    * Just open an r file, start `NVim-R` `SPC r f` to help with omni-Completion and then open Code with `:Codi`, awesome!.
- GitHub - szymonmaszke/vimpyter: Edit your Jupyter notebooks in Vim/Neovim
  - I could never get into Jupyter over using `org-mode=/=RMD` or `Mathematica`

Document Size

So the question is a big document or a small Document, the advantage to a big document is that it groups everything together and if it's well structured it's always easy to parse it back out, but a large document will lag really badly, so it might be better to use an index with listed files as links. Ultimately this will depend on your format because in `org-mode` you can switch to an indirect buffer by moving up to the desired heading with `C-c C-p=/C-n` and then capturing everything below that node with `C-c C-x b` and HTML and aTeX export will both work and only capture only the indirect buffer.

`org-mode`

In `org-mode` my opinion is to go for really big documents, it will run rally slow but you can move everything below the current headline to an indirect buffer with `C-c C-x b` and it speeds it right up.

Suggested workflow:

1. Use `C-c C-x b` to test export formats quickly
2. Quick Partial Export
   - Search for a note using helm-org-rifle-org-directory
     - This can all be set by using something like:

```
(setq user-full-name "Ryan G"
      user-mail-address "exogenesis@protonmail.com")
(setq org-directory "~/Notes/Org/")
```

   - Use `C-c C-x b` to execute org-tree-to-indirect-buffer
   - Use `C-c C-e l o` to export that buffer to a PDF (or HTML)
   - Open that PDF with Dropbox from the top of the recents list.
     - Say for example you had notes on dealing with tables and you wanted some notes by your side on the iPad, this would be perfect.

Export will work as you would expect this way as well so you can still get things out and you can debug a broken export.
This in my opinion is the whole advantage to `org-mode`, being able to wright high quality long complex material with embedded code and inline references `«target»` / `[[source]]` without having to put write it in it in LaTeX and suffer the high verbosity, slow render times and lack of clarity and cross platform support (plus breaking 90% of the features on the way out to HTML.
Even the HTML export works well with large documents with the JS folding, inline references and correct citation thanks to `CiteProc-org`.
Another cause for large documents in `org-mode` is the power of `helm-rifle`, I cannot understand why people aren't outside praising this, it's just amazing! It's just phenomenal. Using helm-org-rifle-current-buffer with `Tab` to preview

Markdown Documents

With *Markdown* Documents I think smaller/moderate files are better because:

- It makes it easier to search for things by leveraging both FileName and File Contents
  - This is different in `org-mode` because all searching with `helm-org-rifle` and the agenda search `C-c a s` is compartmentalised by headlines.
- fighting with `pandoc` won't be fun
- `FZF` and `NV` complement moderately sized files.

Conclusion

The appropriate structure for documents and notes is sort of related to there purpose:

- `Org` Documents should contain each weeks material for a whole class or even up to a semester
- *Markdown* files should contain from a small snippet of code up to one weeks worth of material.

Attachments

Attachments should be dealt with like file links, decide on a folder `~/Notes/MD/attachments` and then just put everything in there.
Images would be a pain in the ass but fourtunately Zettlr has a really high quality image-clipboard insert tool[2]

Finding Material

Tags

Tags revolve around:

- Entering the same tag
- Browsing through tags

Yaml Tags

These are basically Notebooks, they make more sense than folders because you can change them using `vim` and `sed` which scales better over many files and is easier to maintain than manually maintaining `symlinks` across directories.
These are supported by:

- `Notable`
- `VsCode`
  - VSNotes
  - Nested Tags

And they work really well for there rigid strucutre

[2]Marktext doesn't, also another issue with zettlr is that it will only rnder KaTeX of the form `$$\n MATH \n$$` in stark contrast to everything else :(

1. Inserting `YAML` Tags ATTACH In order to insert a `YAML` tag into a note with `vim` you can use `FZF` to read a text file and make suggestions from a text file with those entries:

```
imap <expr> <C-c><C-y> fzf#vim#complete('cat
↪  ~/Notes/MD/notes/00tags.csv')
```

In order to get that text file you can use an ***R*** that leverages `RMarkdown` to pull the `yaml` out:

```r
noteFiles <- c(dir(pattern="*.Rmd"), dir(pattern="*.md"),
↪  dir(pattern="*.txt"), dir(pattern="*.markdown"))
tagVector <- c()

# Run the following code over the entire folder
for (i in noteFiles){
  yamlExtract <- yaml_front_matter(input = i )
  MDTags      <- (yamlExtract$tags)
  tagVector <- c(MDTags, tagVector)


# Generate Symlinks
  for (tagDirPath in MDTags) {
        actDirPath  <- paste0("./aaaamytest/", tagDirPath)
        dir.create(path = actDirPath, recursive = TRUE)
        linkPath=paste0( actDirPath, "/", i)
        print(i)
        print(linkPath)
        createLink(link = linkPath, target = i)
  }
}
```

Then you can just regenerate the tags as needed with `$ Rscript makeTags.R` and insert them with `C-c C-y`.

This can be seen in this gif:

`./images/YAMLInsertion.gif`

2. Browsing Yaml Tags In order to browse `YAML` tags just search for the verbatim structure with `ripgrep`, in this context `YAML` tags are only used to set up notebook directories, so they will always be nested with / characters, hence the number of false positives will be small enough to justify this simplicity:

```
  cat 00tags.csv  | rg '[a-zA-Z0-9]+/[a-zA-Z0-9/]+'  | fzf | xargs
    ↪ rg -l > /tmp/kdkdjaksd;
  cat /tmp/kdkdjaksd
# I couldn't get a pipe to work so I had to save to /tmp
# I've set this all up in a script called TagFilter which I'll
  ↪ append in the appendinx
```

(a) Folder Structure You can Also browse through the YAML tags like a folder structure, the above /**R**/Script at 1 uses a nested `for` loop to create a directory structure with symlinks for the corresponding notes.
so the best way to browse through all the folders is:

   i. just use `FZF` from the terminal:

```
tagFilter.sh -y g
# Press M-c mapped to FZF to chdir to Notebooks
# Ret
# Press M-c to filter directories again
# C-t to filter by file name (or =fif= / =rg=)
```

      A. rememer that one file can be in two directories, don't forget there are

also `#tags`.

   i. Use `ranger`, open it jump to the location with a register map, I use `' n`,

then use `C-f` which is mapped to to `FZF` to filter the directories down.

(b) VisualStudio If you want to use VsCode the following add ons support this out of the box in a practical fashion:

- VSNotes
-

Nested Tags

(c) Notable Notable supports the perfectly, just be careful, sometimes it will add a 'deleted: true' parameter to the `YAML` which is really annoying because it becomes hidden in Notable.

#Tags

So the advantage to inline tags is that they are:

1. Really simple to implement and use
   (a) for example using `YAML` basically requires a parser because:
      i. if you're working with something rigid like `YAML` you need to support it properly or the system will fall apart, but `YAML` can use:
         A. python-style lists
         B. mappings

C. New line sequences

ii. Even if YAML used just one syntax, it would not be easy to implement in regex because look around features don't like to work with wild cards, moreover doing something like `\-\-\-\n[\w\W]*tags:\s` cannot be efficient.

2. Can appear anywhere in your document

   (a) If you use *iamcco's* preview, the scroll can be locked to once you jump to the tag it will be in the preview as well.

3. Because they are easy to implement they can be concurrently filtered

I've elected to use `#tags` rather than `@`[3] becuase it's more common and is implemented by `iaWriter`, meaning I can use smart folders on the `iPad`, `:tag:` was another option I rejected [4].
In order to reduce false positives it's important to have a really clear definition of a tag, I've chosen to do `\s#TAGNAME\s`, this should work well, I just need to be mindful to include spaces around the tags.

1. Listing already Created tags This is where `#tags` really shine, because they are easy to parse and because `ripgrep` and `fzf` are both lightning fast all the notes can be searched and all the tags listed on the fly like so:

```
imap <expr> <C-c><C-t> fzf#vim#complete('rg --pcre2
↪  "\s#[a-zA-Z-@]+\s" -o --no-filename $HOME/Notes/MD/notes -t md
↪  \| sort -u')
```

   Then simply presing `C-c C-t` in `vim` will allow you to enter the matching tag.

2. Filtering by the desired tag This is where `#tags` are amazing, because they are so simple, they can be recursively filtered for, first create a temporary file to store any notes that have tags in them[5], then use `ripgrep` with `look-around` to extract the tag name:

   First find all the tags and offer the user

[3]This is used by `Notes.vim`

[4]This is used by `VimWiki` and `org-mode`, I don't like them because `pymarkdown` already uses `:` to delimit emojis. Also `org-mode` only uses them in Headlines, according to the manual.

[5]A variable would be quicker than a file because it's stored in memory, I did a file when I was playing in the terminal using `for` loops rather than `pipes` and it just made it's way into the script I'm using, there is a TODO next to it but the whole script needs to be rewritten anyway.

```bash
    # Make a temp file to store results if necessary
  if test -f 00TagMatchList; then
          echo "subsequent search";
  else
           # List in order of modification Date, top newest
          ls -t *.md > 00TagMatchList;
  fi

tagval=$(cat 00TagMatchList | xargs -d '\n' rg --pcre2
 ↪  --no-pcre2-unicode --no-filename
 ↪  '(?<=[\n\s]#)[a-zA-z]+(?=[\s\n$])' -o | sort -u | fzf)
```

After the tags have been identified and passed to the user, save the `tagval` as a temp file and search through all the notes listed in the file for any matches:

```bash
if [[ ! -z $tagval ]]; then
    echo '
      wait for fzf to finish otherwise ripgrep has
nothing with which to filter  '
    exit 1
else
  cat 00TagMatchList | xargs rg ":$tagval:|\s#$tagval\s" -l >
   ↪  00TagMatchList;
fi
   bat 00TagMatchList
```

Now you can re run those last two commands (which I've written into a `bash` script as `tagFilter.sh` here, over and over again until the number of listed tags is down to a reasonable number[6] .

Once you're satisfied that the results are sufficiently filtered you can dump them as symlinks into a direcory like so:

```bash
mkdir 00TagMatch
rm 00TagMatch/*
ln -s $(realpath $(cat 00TagMatchList)) ./00TagMatch; rm
 ↪  00TagMatchList
```

Then you can go through that directory using `fzf` and `--preview` with [7]:

[6]This is why using descriptive file names is important, with all notes in one directory it will be necessary to have unique file names, but using a sample of 1000 words and never repeating a word and not using the same 3 words in two files there will be $\binom{1000}{3} > 10^6$ different combinations, so there is no need to use a UUID.

[7]I got this idea from the Github Wiki

```
rg --files-with-matches --no-messages "$1" | fzf --preview
↪  "highlight -O ansi -l {} 2> /dev/null | rg --colors
↪  'match:bg:yellow' --ignore-case --pretty --context 10 '$1' ||
↪  rg --ignore-case --pretty --context 10 '$1' {}"
```

Or you you could preview the files in ranger using `glow` by appending the following to your scope.sh:[8]

```
...
  md|markdown)
    glow -s dark "${FILE_PATH}" && { dump | trim; exit 5; }
    ;;
  ...
```

(a) Example This can be seen in the below gif:
`./images/hashtagfilter.gif`

3. Integrating with TMSU Alternatively you can leverage TMSU to make all the symlinks, both is probably the way to go because that way you're not tied to TMSU (plus the mounting is a little buggy) but you can still leverage it:

```
# Piping is still better than a loop because ripgrep acts on
↪  everything
    # Unfoutunately you cannot pipe directley into tmsu
  cd ~/Notes/MD/notes
  rg --pcre2 '(?<=\s#)[a-zA-Z]+(?=\s)' *.md -o \
      | sed s+:+\ + | sed s/^/tmsu\ tag\ / | bash
```

How this works:

(a) change into the notes directory
(b) use ripgrep to extract anything with a space followed by a # followed by atleast one or more letters and then another space, ripgrep is told to return the mathching output rather than the files, this output has the filename and the result like `notes.md:tagname`.
(c) sed is used to convert that into arguments for tmsu
(d) sed is used to prefix `tmsu` so this is a literal `tmsu` operation
(e) This string is piped into `bash` (you could have given it to `zsh=/=fish` whatever)

[8]`ranger` crashes all the time though, there's some bug with the way the directories are mounted and vim, for example vim can only change into that
directory after being loaded, so `vim -c 'cd 00tmsutags'` is fine but `vim 00tmsutags` is not fine.

TMSU wouldn't work so well with nested `yaml` TAGS because TMSU doesn't have any notion of nested tags, instead it would be easier to create symlinks directly from ***R*** using `R.utils::createLink()`

4. Browsing through the Symlinks

   Then you can mount the TMSU Virtual File System wherever you like and there is an easy to browse through tags.

   What works really well though is to open nvim in the directory with:

   ```
   # any other way causes a crash
   # also confusingly =|= is to vim as =;= is to bash
   nvim -c 'set noautochdir | cd 00tmsutags/ | pwd'#+end_src
   ```

   and then search through the files using `:Files` from FZF (mapped to `C-p` / `SPC f f`) and/or *NerdTree*, and use *iamcco's* preview in *Firefox* with Tree Style Tab to stay organised. If you'd rather work from `bash` you can use `fzf` with the `--preview` option from before.

   Or you could use vim with and a preview app like *iamcco's*.

   Or you could preview all the markdown in a rendered state by using *MarkText* / or *Typora*, or *Zettlr*, *Abricotine*, *Typora*, *MarkdownViewer* Chrome Plugin[9] , *MkDocs* or *Docsify*, are other options that work in some way.

   Another good option for navigating the '*tags are now folders*' structure is to use *VSCode* and/or *atom*, both are well suited to navigating through a dense structure.

   Dillinger is also really cool, but, I don't know what I'd use it for, could I get it on the ipad by hosting my own server maybe?

5. Renaming Tags Just use a careful application of `sed` if this is necessary:

   ```
   sed -i s+#OldTag+#NewTag+g
   ```

6. Renaming Files Somewhat integral to the Tag Filtering operation above is reasonable file names, if it's necessary to fix the file name just use `chdir "%p"; ! mv "%" newname.md`.

   This will break links, but if a note is already linked from elsewhere you'd make a new note and/or delete the old one.

7. Searching Tag Under Cursor In order to search for the word under the cursor you could just `:Rg` by `FZF` , `:NV` by `notational-fzf-vim` offers a seamless preview as well so I'll just use that.

   Add the following to `.vimrc` and you'll be able to search for a tag under the cursor with `SPC f g` and if you forget keyboard shortcuts `SPC Tab` will list them.

[9]You might want to start a python server for this with `python3 -m https.server 8392`

```
"let mapleader="\<Space>"
map <Space> <Leader>

:set iskeyword+=#
nmap <Space>fg :NV <C-r><C-w> <CR>
nmap <leader><tab> <plug>(fzf-maps-n)
```

PDF Files

The reality of life is such that you will have to deal with PDF Files. I haven't thought much about attaching them really because if you link to the PDF file does it matter much? Isn't an attachment a note linking to a PDF file right?

Searching PDF Files

1. Bash ATTACH In bash you can basically acheive this by performing:

```
rga '.*' *.pdf | fzf | cut -d ':' -f 1
```

   of course the question then becomes script or alias, I always choose script because I can only get NVM to work in bash but I prefer fish and when I absolutely need bash compatability I use zsh so I don't want to maintain 3 `rc` files and two `.profiles`, a script is bearable, here is a sample script.

2. Ranger Searching PDF Files is obviously a big deal, though, I've basically acheived this by using ripgrep-all in ranger by adding the following to my `~/.config/ranger/commands.py` :

```python
# RipGrep with FZF
class fzf_rga_documents_search(Command):
    """
    :fzf_rga_search_documents
    Search in PDFs, E-Books and Office documents in current
    ↪ directory.
    Allowed extensions: .epub, .odt, .docx, .fb2, .ipynb, .pdf.

    Usage: fzf_rga_search_documents <search string>
    """
    def execute(self):
        if self.arg(1):
            search_string = self.rest(1)
        else:
            self.fm.notify("Usage: fzf_rga_search_documents <search
            ↪ string>", bad=True)
            return

        import subprocess
        import os.path
        from ranger.container.file import File
        command="rga '%s' . --rga-adapters=pandoc,poppler | fzf +m
        ↪ | awk -F':' '{print $1}'" % search_string
        fzf = self.fm.execute_command(command,
        ↪ universal_newlines=True, stdout=subprocess.PIPE)
        stdout, stderr = fzf.communicate()
        if fzf.returncode == 0:
            fzf_file = os.path.abspath(stdout.rstrip('\n'))
            self.fm.execute_file(File(fzf_file))
```

Then you can just use `:fzf_rga_documents_search term` in order to search for the material, it's pretty helpful to change to a narrow directory first though, PDF's can be previewed in ranger by:

(a) Using a terminal like kitty.

(b) Uncommenting these lines from the `~/.config/ranger/scope.sh`:

```
# PDF
application/pdf)
    pdftoppm -f 1 -l 1 \
             -scale-to-x 1920 \
             -scale-to-y -1 \
             -singlefile \
             -jpeg -tiffcompression jpeg \
             -- "${FILE_PATH}" "${IMAGE_CACHE_PATH%.*}" \
```

In order to preview

This can also

Viewing Material

Vim-Plugin

Firefox

while Previewing MD in the browser, in order to keep your sanity, you're going to want to use this Tree add on to keep yourself organised.

Chrome Browser Extension

Firefox will not allow local MD files to be rendered with an add on as a security policy, instead you can Firefox doesn't allow markdown files to be rendered inside the browser if they are local (nor does it make it easy if you do it with a simple `python3 -m http.server 8089 --bind 192.168.0.137`) you'd be better off just doing it in chrome with This extension

WYSIWYG Editor

Notable

Org-Mode

The more I use `org-mode` the more I see how useful it is, with `org-wiki` and `org-brain` using it as a central dispatch with vim as a text editor there isn't much that I can't do to be honest.

Speeding Emacs up

One of my biggest gripes with emacs is speeding it up, fortunately that's not to hard to fix by:

1. Running it as a daemon
2. Loading it into memory
3. Switching to PosFrame
   (a) use `company-posframe-mode` it's faster
4. Doom Emacs
   (a) this is faster but I couldn't get , try as I might, doom emacs to have `vim` source code blocks to come out as `vimrc`, this is unique to *Doom* it doesn't happen in spacemacs so I had to leave it.
      i. basically the issue is that you need to list source blocks as

`vimrc` so that they correspond to `vimrc-mode` in emacs in order to get syntax highlighting in emacs and in HTML export, unfourtunately in LaTeX you need to use `minted` package which wants them listed as `vim` so you have to change the =org=variable, *Doom* gives an errror every time that happens though.

Running emacs as a daemon

This works really well, follow the instructions on the Emacs Wiki [10] and then create a bash script called something like `ema` and tie it to a keyboard shurtcut like =s-e=that calls:

```
if [ $1="" ]; then
    emacsclient --create-frame ~/Notes/index.org
else
    emacsclient --create-frame $1
fi
```

Throw it in RAM

OK so I'm not sure if it's my imagination or not, but throwing emacs in ram seems to make it feel smoother, regardless, emacs and all my org notes are 2.5 % of my total system memory and this is like my main tool so I don't see why I wouldn't use memory I paid so much for.
I got this hint from this blog and the idea is to use vmtouch to load everything into memory, so at startup, have the following run:

[10] `https://www.emacswiki.org/emacs/EmacsAsDaemon`

```
# -v is verbose
# -t is *T*ouch into memory
# How much is currently in memory?
vmtouch ~/.emacs.d/
# Put it in memory
vmtouch -vt ~/.emacs.d/
vmtouch -vt ~/Notes/Org

# Now how much is in memory??
vmtouch ~/.emacs.d
```

1. Load All Agenda Files With everything in memory you may as well load all your agenda files as buffers, they're already in memory and this means you won't have to waste time looking for material, in the end you'd be hard pressed to eat up too much memory and if it ever got that bad you'd just have to trim your agenda down.

   Put this in your `~/.emacs.d/init.el` and you'll be able to use `M-x open-all-org-agenda-files` to get all buffers ready to go.

   ```
   ;;;;; Open all Agenda Files
   (defun
   open-all-org-agenda-files () (interactive) (let ((files
   ↪  (org-agenda-files))) (mapcar (lambda (x) (find-file
   x)) files)))
   ```

Helpful Packages

There's a few packages, just refer to my DotFiles.

Org-Brain

This is on MELPA so it's pretty easy to use, but the key bindings are a nightmare, once you get used to them though they're alright, just remember you need to use `C-x o` to get out of the window. Basically `Org-Brain` recreates an index for you as a temporary buffer by using extensive tags in the documents property draw.
It's pretty complicated and the index is generated on the fly, I wouldn't recommend it over just maintaining your own index properly.

1. Important Requirements

   (a) All relevant Headlines MUST have an `org-id` [11]

[11] GitHub - Kungsgeten/org-brain: Org-mode wiki + concept-mapping

i. This can be done from `org-mode` with `M-x org-id-get-create` or by using `M-x org-brain-refile` to do it automatically.

2. KeyBindings For reference sake the keybindings are:

Key	Command	Description
m	`org-brain-visualize-mind-map`	Toggle between normal and mind-map visualization
j or TAB	`forward-button`	Goto next link
k or S-TAB	`backward-button`	Goto previous link
b	`org-brain-visualize-back`	Like the back button in a web browser.
h or *	`org-brain-add-child-headline`	Add a new child *headline* to entry
c	`org-brain-add-child`	Add an existing entry, or a new *file*, as a child
C	`org-brain-remove-child`	Remove one the entry's child relations
e	`org-brain-annotate-edge`	Annotate the connection between the visualized ent
p	`org-brain-add-parent`	Add an existing entry, or a new *file*, as a parent
P	`org-brain-remove-parent`	Remove one of the entry's parent relations
f	`org-brain-add-friendship`	Add an existing entry, or a new *file*, as a friend
F	`org-brain-remove-friendship`	Remove one of the entry's friend relations
n	`org-brain-pin`	Toggle if the entry is pinned or not
s	`org-brain-select-dwim`	Select an entry for batch processing.
S	`org-brain-select-map`	Prefix key to do batch processing with selected entr
t	`org-brain-set-title`	Change the title of the entry.
T	`org-brain-set-tags`	Change the tags of the entry.
d	`org-brain-delete-entry`	Choose an entry to delete.
l	`org-brain-visualize-add-resource`	Add a new resource link in entry
r	`org-brain-open-resource`	Choose and open a resource from the entry.
C-y	`org-brain-visualize-paste-resource`	Add a new resource link from clipboard
a	`org-brain-visualize-attach`	Run `org-attach` on entry (headline entries only)
A	`org-brain-archive`	Archive the entry (headline entries only)
o	`org-brain-goto-current`	Open current entry for editing
O	`org-brain-goto`	Choose and edit one of your `org-brain` entries
v	`org-brain-visualize`	Choose and visualize a different entry
V	`org-brain-visualize-follow`	Similar to `org-agenda-follow-mode`; view visuali
w	`org-brain-visualize-random`	Visualize one of your entries at random.
W	`org-brain-visualize-wander`	Visualize at random, in a set interval. `W` again to ca
C-c C-x C-v	`org-toggle-inline-images`	Display org-mode images in the entry text.
M	Move prefix	Move (refile) the current entry.
M r	`org-brain-refile`	Move current entry to another entry (change local
M p	`org-brain-change-local-parent`	Choose among the entry's parents and make anoth

Org-Wiki

This has Some niceties to build a wiki index because `org-mode`, although has `helm-rifle` and `agenda-tag` search, it does not have notebooks or anyway to list all agenda files that I'm aware of, you've really gotta make it on your own.

Magit

This is phenomenal, you can commit changes to git all from emacs, it's so easy to stage and commit changes! then when you're finally done you can Push, so nice to commit straight from the document you just edited and only that document as well so the comments will be far more targeted!

Helm

`helm-rifle` is amazing and I much prefer `Helm-M-x` over the `ivy` equivalent (if and only if you use the popup window, otherwise it's too slow), it simply returns more results more accurately. `Helm-find-files` is also simpler to use than the ivy equivalent.

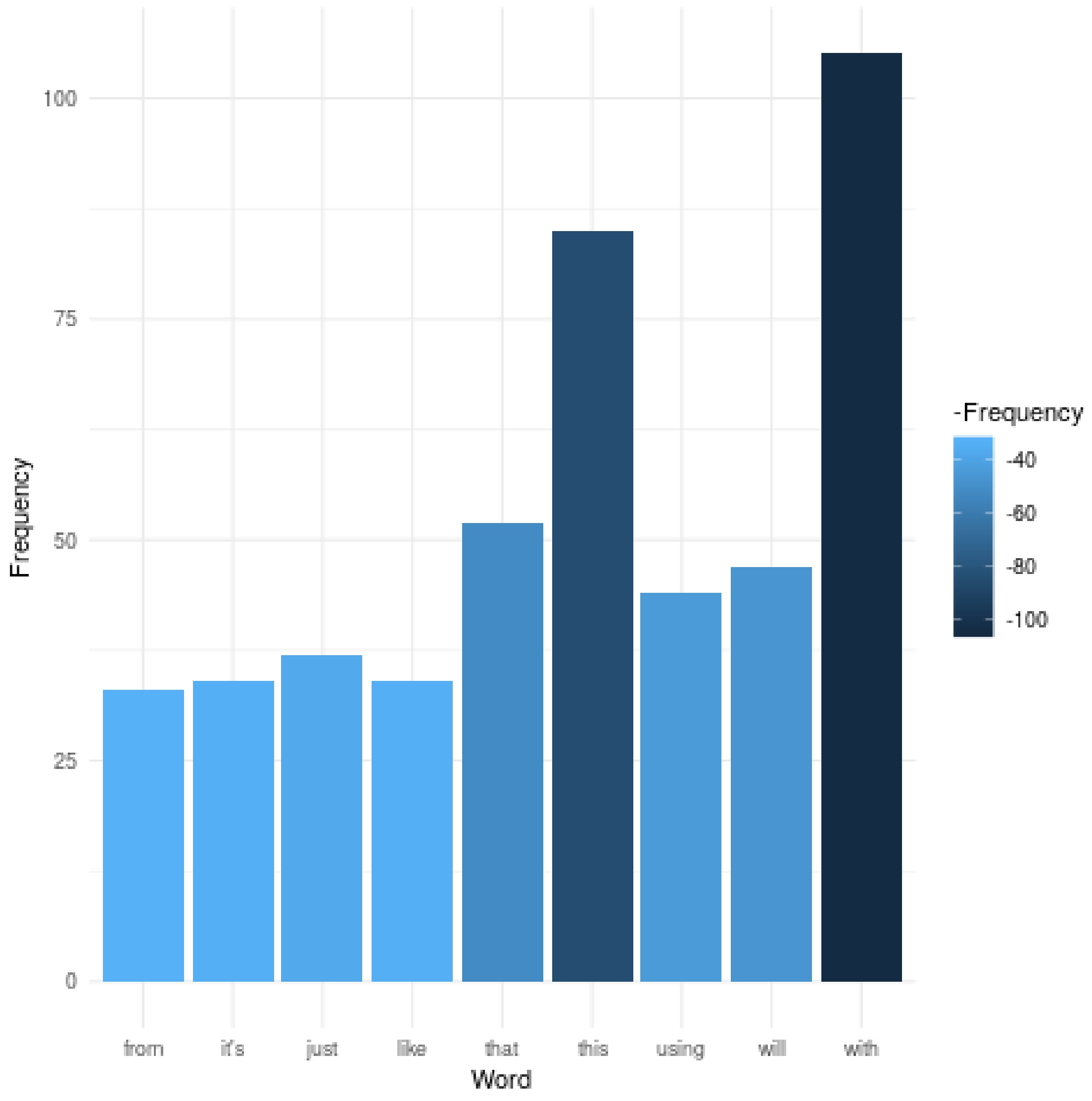
Emacs Speaks Statistics

Emacs is amazing for literate programming, but it's a nightmare to work with, it has it's place but `Nvim-R` is definitely best in class for literate programming using ***R***-*Markdown*, but there are cases where using ***R*** in `org-mode` is really handy, for example, to make a bar chart of all the words in this manual:

```r
## Load Tidy Verse
library(tidyverse)

## Scan Through the Manual
  words <- tolower(scan("./manual.org",
                        what = "", na.strings = c("|",":"))
                       )

## Use Grep to pull out org #+ Blocks
  words <- words[-grep(pattern = "#.*", x = words)]

## Sort the words by frequency and cut the results
  topwords <- sort(table(words[nchar(words) > 3]), decreasing =
   ↪  TRUE)[1:9]
  topwords <- as_tibble(topwords, .name_repair = "universal")
  names(topwords) <- c("Word", "Frequency")
  print(topwords)
```

Observe that setting `:session name` allowed `org-babel` to connect the two code chunks into a single session so that the output of the prior was accessible to the second chunk.
Now Generating the plot:

```r
     ## Make Plot
library(tidyverse)
ggplot(topwords, aes(y = Frequency, x = Word, fill = -Frequency))  +
    geom_col() +
    theme_minimal()
```



1. Using ESS with `org-babel`

   (a) Output types

      i. Results Type Notice that this block when executed with `C-c` will give the STDOUT

      ```
      rnorm(9)
      ```

      Where as this block will give the expected results:

      ```
      rnorm(9)
      ```

      ii. Format Type
      You can also change the output to better integrate with org by using `:format raw` or `:format org`:

```r
rnorm(9)
```

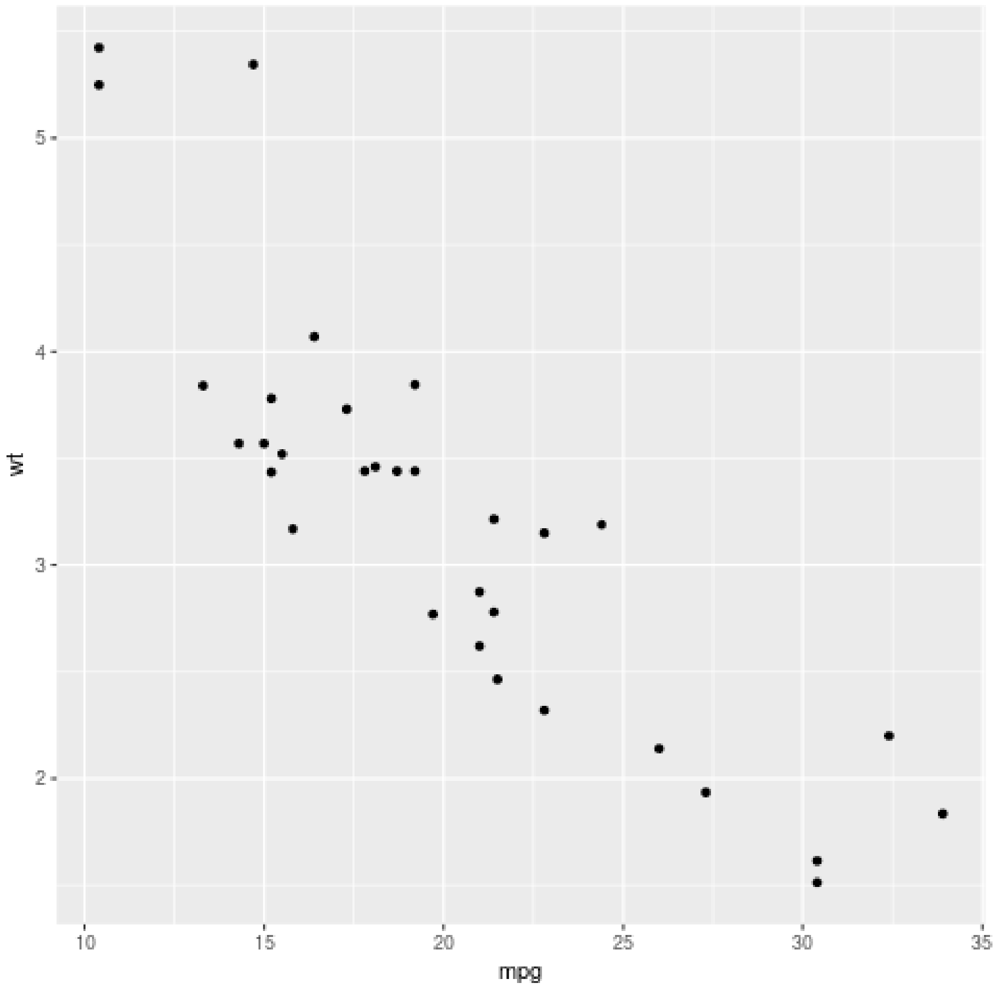
(b) Working with Graphics

Switching back to `:results value` so we don't get crap from the graphic export:

```r
library(tidyverse)
rnorm(9) %>% hist()
```

We can see that this gives us a popup, which we don't really want, instead, let's deal with an inline image, this can be acheived by using `:results output graphics file`:

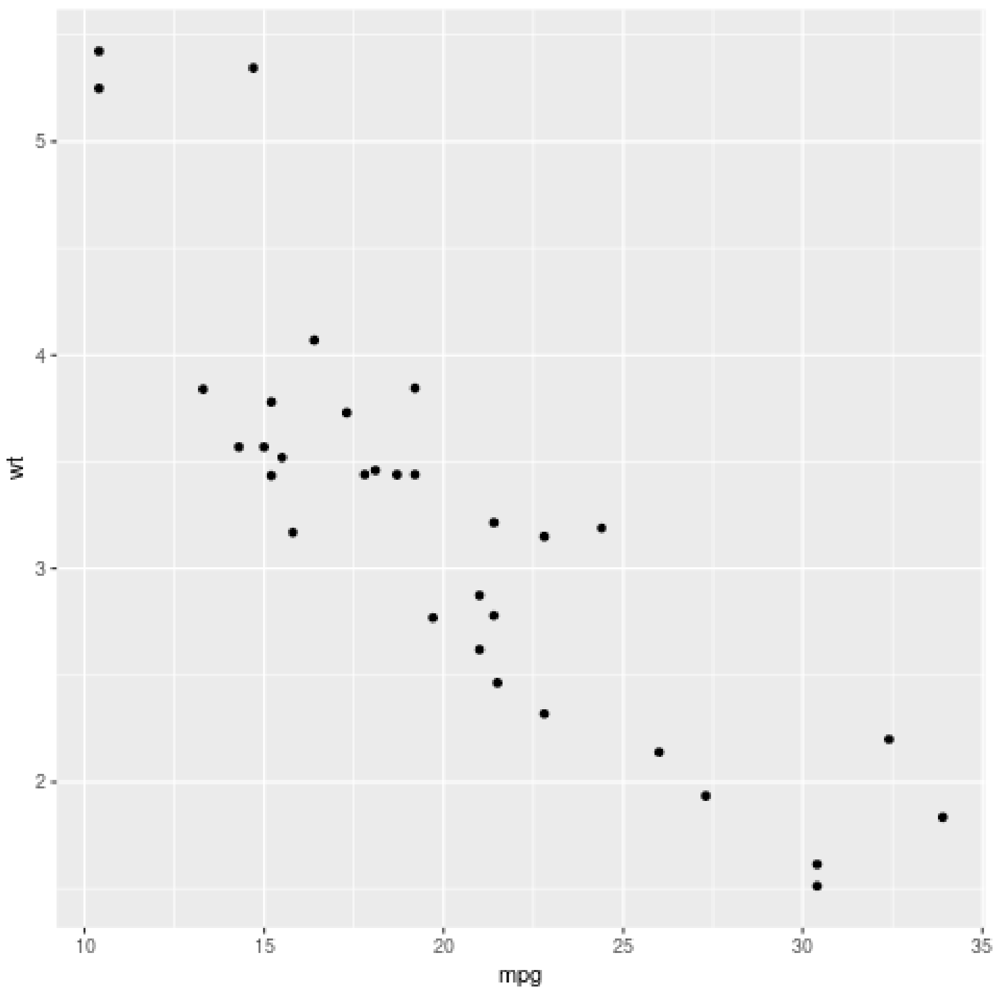
```r
library("ggplot2")
qplot(mpg, wt, data = mtcars)
```



In markdown just passing to a `REPL` it would be necessary to do something really ugly like this:

```r
library("ggplot2")
# Using SVG is also an option, but LaTeX sucks.

# Give the filename of the plot and a Description
plotname="chocolatemouselamp"
description= "Just a test to get =org-babel= plots"

# Export the Plot into the working directory
png(filename=paste0(plotname, ".png"))
print(qplot(mpg, wt, data = mtcars))
dev.off()

# Print the =org-mode= link syntax
print(paste0(

   "\n",
    "Output:", "\n",
     "[[./", plotname, ".png", "]]"))
```

Which gives the outpu:



Attachments

These are super handy because generally I don't actually want to edit from emacs, i'd rather edit in vim and leverage all the poewr of `org-mode` and emacs and jump to my text editor as shown below at , basically this makes `org-mode` my operating system, and vim my text editor.
Open the attach dispatcher with `org-attach` using `C-c C-a` (or `, A` on Spacemacs) and then use `s` to change the directory to something reasonable so you can find them later with `fzf`.
open the dispatcher again with `C-c C-a` and press

Important

It is imperative that all attachment directories are given ***Relatively*** not absolutely because otherwise upon export the links will break.
also if you copy a link to an attachment into another org file that doesn't share that same attachment directory, then, obviously, that link will break, so attachment directories are only a little more robust than an ordinary directory really.

Sharing

- put the HTML on Github
    - use the JavaScript for HTML with `#+INFOJS_OPT`
    - In order to get custom CSS use the following code:

```elisp
;;;;;;  Add CSS to HTML
;; Add CSS (Be mindful that you may want to implement this in a more
↪  sensible way, similar to how beorg does it
;; Put your css files there
(defvar org-theme-css-dir "~/.emacs.d/org-css/")

(defun toggle-org-custom-inline-style ()
 (interactive)
 (let ((hook 'org-export-before-parsing-hook)
       (fun 'set-org-html-style))
   (if (memq fun (eval hook))
       (progn
         (remove-hook hook fun 'buffer-local)
         (message "Removed %s from %s" (symbol-name fun) (symbol-name
         ↪  hook)))
     (add-hook hook fun nil 'buffer-local)
     (message "Added %s to %s" (symbol-name fun) (symbol-name hook)))))

; Enable Css hook by default
(add-hook 'org-mode-hook 'toggle-org-custom-inline-style)

(defun org-theme ()
  (interactive)
  (let* ((cssdir org-theme-css-dir)
         (css-choices (directory-files cssdir nil ".css$"))
         (css (completing-read "theme: " css-choices nil t)))
    (concat cssdir css)))
(defun set-org-html-style (&optional backend)
 (interactive)
 (when (or (null backend) (eq backend 'html))
(let ((f (or (and (boundp 'org-theme-css) org-theme-css) (org-theme))))
     (if (file-exists-p f)
         (progn
           (set (make-local-variable 'org-theme-css) f)
           (set (make-local-variable 'org-html-head)
                (with-temp-buffer
                  (insert "<style type=\"text/css\">\n<!-- [CDATA]
                  ↪  -->\n")
                  (insert-file-contents f)
                  (goto-char (point-max))
                  (insert "\n/;]]>;/-->\n</style>\n")
                  (buffer-string)))
           (set (make-local-variable
           ↪  'org-html-head-include-default-style)
                nil)
           (message "Set custom style from %s" f))
       (message "Custom header file %s doesnt exist")))))
```

- In order to bulk export HTML use the following:
  - You will be propted to enter themes, e.g. ‘mystyle.css‘ and ‘org-babel‘ working directories, e.g. ‘./‘

```
emacs --batch -l ~/.emacs.d/init.el -f org-wiki-export-html-sync --kill
```

Mobile

Basically `beorg` is pretty good `WorkingCopy` makes you feel like a first class citizen when looking through the HTMLs.

Browsing / Editing

The bestway to browse `Org` files is to open the repo in working copy, travel to `index.html` and then hit the `preview` option in order to establish a local server, then you can browse through everything on the ipad without having to host a webpage.
If people are on the same network you can even share the link with them! its awsome

ToDos

Use beorg when you need to set to dos, and either file sync working copy into icloud (this way beorg can access via git) or just use dropbox (the git can stay on dropbox and it 90% works).

Finding Notes

Tags

use `org-tags-view` (mapped to `C-c a m` / `SPC a o m`) to search for a tag, and once your in the results view use \ to further filter by tags. using `C-SPC` will mark a file to be opened and `TAB` will preview the result.
You can also make a tags filter sparse-tree by using =C-c / m = inside a buffer, that's super handy for making sure your tags are well behaved.
Even though tags only work for agenda files, it's not really possible to list all agenda files out of the box because the agenda only shows files with a date or a =TODO =status. For this reason, I don't see any reason not to add all notes into the agenda and then use the TODO/date features of the agenda to ignore none-task-based notes, this can be acheived by doing:

```
;; This is Not Recursive
       ;; (but helm and org-wiki are so bear that in mind)
(setq org-agenda-files '("~/Notes/Org"))
```

1. Open all Agenda Files It is possible however to open all agenda files with some `elisp`:

```
(defun
open-all-org-agenda-files () (interactive) (let ((files
↪  (org-agenda-files))) (mapcar (lambda (x) (find-file
x)) files)))
```

Then you can rifle through all open files with `helm-org-rifle`, which will start way quicker now.

Searching

1. Generally All of these methods allow a preview with TAB, but, `Rifle` has an in place preview

Method	**Fuzzy**	**Search Domain**
`Helm-Org-Rifle`	YES	HEADLINE NODE
`Helm-Swoop`	YES	LINE
`Helm-rg`	NO	LINE
`Helm-org -agenda-files-headings`	YES	HEADLINE

`helm-rg` does however support `regex`

So generally, first mark where you are `mh` before running `helm-org-rifle`, then use use `C-c C-f` to automatically preview or use `C-j` to preview, once you're happy with what you see (or the popups in the way), use `Ret` to get a look at it, jump back to the search with:

After following the result with `Ret` you can resume the last search with:

- `helm-resume` (`SPC r l`)
- `Spacemacs/resume-last-search-buffer` (`SPC s l`)

Also worth noting is `helm-show-kill-ring` (`SPC r y` / `M-m r y`), say you need to fix a link, open a buffer to the target and copy the path with (`SPC f y`), when you go back to the link to fix it with C-c C-l and remove the old link with `C-a C-k` that will compromise your kill ring, this is where you hit `M-m r y` in order to go through your kill ring to retrieve it.

2. Helm-Org-Rifle `helm-org-rifle` is brilliant, it searches like *Google* off to the side, matches a search relative to fuzzy material through beneath the headline () (as opposed to only in the line like `ripgrep`, I'm also not sure if that includes all children as well), results can be previewed with `C-j`, and automatic preview (`follow-mode`) can be toggled with `C-c C-f`)

   There's more information , and also Tab can be remapped to preview with:

```
(define-key helm-map (kbd "<tab>") 'helm-execute-persistent-action)
↪  ; rebind tab to do persistent action
(define-key helm-map (kbd "C-i") 'helm-execute-persistent-action) ;
↪  make TAB works in terminal
(define-key helm-map (kbd "C-z")  'helm-select-action) ; list
↪  actions using C-z
```

- `helm-org-rifle` which is mapped to `M-m a o r` which is like a live fuzzy search with preview. You can use `C-n` / `C-p` to cycle through the buffers and `C-j` to preview, if the math was left generated as images it will seen that way in the preview as well.
  - Basically headlines act like files so the matches will be within a headline from what I can tell.

3. `RipGrep` Another option is `helm-rg` (`SPC /`) but it's not fuzzy and it's line by line

4. Swop

5. Helm

   (a) Helm-Rifle Basically just use `helm-rifle`, it will search all the contents below a headline and return you to a *node* that matches a fuzzy search.

   This is superior to say `grep` which only search that line, which is pretty useless unless you're really organised.

Command	**Description**
`helm-org-rifle-directory`	Choose a directory to search over org files in
`helm-org-rifle-org-directory`	Search over org files in org directory [12]
`helm-org-rifle`	Search through open Buffers
`helm-org-rifle-agenda-files`	Search through agenda files
`helm-org-in-buffer-headings`	Search through headings of buffer
`helm-rg`	Use `rg` over a directory (so only line by line!)
`helm-ag`	Use `ag` over a directory (so only line by line!)
`helm-swoop` (`SPC s s`)	Use `ag` over a directory (so only line by line!)

   After a search is dispatched:

   - `TAB` will preview the result, Enter will open it
   - `C-Space` will mark a file to be opened
   - hitting `C-c C-f` will enter follow mode making the preview automatic.
     - F2 will take the match into an indirect buffer.
     - Enter will Open the match directly

   After the search is complete if you want to go back to the searches you need to use `helm-resume`, the problem with this however is that on many setups `M-x` is already bound to `helm-M-x` meaning resume will just open that, instead it is imperative that you use a keybinding for it, `C-x c b` or lazy resume with `Spc r l`.

   I have tested this, when it fails often it's because it's generating LaTeX Previews.

   i. Display window Left Obviously the popup window at the bottom is pretty stupid on a 16:9 monitor, ideally it would be on the left, this can be acheived, (at least I got it working in with no other splits) with `C-t` from the helm pane).

   So if you have no other splits just use `M-x helm-org-rifle-org-directory Ret` then press `C-t C-t`.

   To make this the default use:

[12]

```
(setq helm-split-window-default-side 'left)
(helm-autoresize-mode 1)
```

but this only works for `swoop` and `rg` NOT for `helm-org-rifle`

ii. Popup Window Another option is to have a popup frame [13], this performs way faster and I've found that using `menu-set-font` to choose well behaving font can be a good idea [14]

```
(setq helm-display-function
 ↪  'helm-display-buffer-in-own-frame
        helm-display-buffer-reuse-frame t
        helm-use-undecorated-frame-option t)
```

Be mindful that you can `Alt-Tab` between this popup it's a first class `X11` frame [15]

`t`

(b) Other neat Helm Features

While we're talking about helm, you should definitely try:

- `helm-regexp`.
- `helm-google-suggest` so type `C-x c C-c g` then type a search term and then press tab to preview.
- `helm-color` for working with HTML and CSS is nice.
- `helm-calcul-expressoin` for quick math, try `C-x c C-,` and then type something like `sin(9)` (degrees).
- `Helm-show-kill-ring` `C-x c M-y` / `Spc r y`
- this is far more effective if you use descriptive headlines
- All the searches in org mode encapsulate everything beneath the first
- there's also:
  - `org-wiki-nav` which is an alias to helm-org-in-buffer-headings

  this is probably the preferred method to browse because it's a sanity check for your index. It will fuzzy search accross index headlines.
  - `org-wiki-server-toggle` will start a python3 server (think python3 -m https 8000) for you to browse through your notes with.
  - with `org-agenda-search` (mapped to = Spc a o s=) headline, so basically treat the first headline like you would a single file/filename.

(c) Potential Workflow

- Open the index with `Spc w w`
- `Helm-multi-swoop-all` to jump where you need to be

[13] Now you can use Helm with frames instead of windows : emacs

[14] Also if you're using Spacemacs definitely use the `base` NOT the full package, bear in mind this breaks using `Ret` to follow links, you instead need to use `SPC m l`

[15] I haven't tested on Wayland because *Nvidia*

    - change your mind and jump back to the index with `org-wiki-nav`
    - Alternatively use `helm-bookmarks` (`SPC f b` in Spacemacs)
  - Search for related material with `helm-org-rifle-org-directory`

6. Within the Agenda To work with the agenda specifically `Helm-Rifle` and `Helm-org` have corresponding commands e.g:

   - `helm-org-rifle` agenda files will search through only the agenda files.
     - It is also possible to use `org-search-view` but it's not live.
       * the search also must use + to denote boolean and.
   - `C-c a /` Will also search for occurences, of words without a live

   preview though .

Finding Material Old

Basically you just want to use a compination of searching content and browsing tags.

1. Browsing There's no real concept of notebooks, from what I can understand, a headline is sort of like a Section and maybe an org file is like a Notebook bookshelf? regardless it's fairly trivial to just use links with `org-wiki`, if not arguable preferable because atleast it's all self contained, I suppose you cald also use `treemacs`)and the emacs ranger) to use a sort-of folder structure if you're brave enough to fight bad links. along use the search option from inside the HTML with the java script

Other Cool Things

Helm REPL's

So I thought Codi was a nead vim add-on, but, in emacs you can do `SPC a '` and open any REPL you want.

Appendix

NCM2 Most Completion Frameworks

Most of the completion framework you're going to want is provided by the following minimal `~/.vimrc` that Assumes you're using JuneGunns Plugin Manager.

```vim
call plug#begin('~/.vim/plugged')
"""""" NCM2
Plug 'ncm2/ncm2'
if !has('nvim')
    Plug 'roxma/vim-hug-neovim-rpc'
endif
Plug 'Shougo/neco-vim'
Plug 'roxma/nvim-yarp'
autocmd BufEnter * call ncm2#enable_for_buffer()
" IMPORTANT: :help Ncm2PopupOpen for more information
set completeopt=noinsert,menuone,noselect
" NOTE: you need to install completion sources to get completions. Check
" our wiki page for a list of sources: https://github.com/ncm2/ncm2/wiki
""""""" Sources
" R
Plug 'gaalcaras/ncm-R'
" Go
Plug 'ncm2/ncm2-go'
" Python
Plug 'ncm2/ncm2-jedi'
" Java
Plug 'ObserverOfTime/ncm2-jc2'
" CSS
Plug 'ncm2/ncm2-cssomni'
" VimScript
Plug 'ncm2/ncm2-vim'
" UltiSnips Completion Source
Plug 'ncm2/ncm2-ultisnips'

"""""""" General
" Words in Buffer
Plug 'ncm2/ncm2-bufword'
" Path
Plug 'ncm2/ncm2-path'

"""""""" Potentially Annoying
" Single Line Clipboard Looks in clipboard history
"Plug 'svermeulen/ncm2-yoink' " This caused an error
" Word Completion Looks up possible words
Plug 'filipekiss/ncm2-look.vim'
" Highlights what caused the match
"Plug 'ncm2/ncm2-match-highlight'
Plug 'fgrsnau/ncm2-otherbuf'
" Initialize plugin system
call plug#end()
```

Open Vim From Emacs

Actually Open Vim

Use the following to open vim from emacs, in this case I'm using `kitty` because it's become one of my favourite terminal emulators:

```
(defun my-open-current-file-in-vim ()
 (interactive)
 (async-shell-command
;  (format "gvim +%d %s"
  (format "~/.local/kitty.app/bin/kitty -e nvim +%d %s"
      (+ (if (bolp) 1 0) (count-lines 1 (point)))
      (shell-quote-argument buffer-file-name))))
```

in order to get vim to play ball with an `org` file you'll need this plugin:

```
call plug#begin('~/.vim/plugged')

"Org Mode
Plug 'jceb/vim-orgmode'

call plug#end()
```

TagFilter Script ATTACH

The script for filtering `yaml` and `#tags` can be found here

References

TODO Clean up and Replace previous Method

+ Old Manual