A Life Configuring Emacs

Musa Al-hassy

2018-07-25

Contents

Abstract

Herein I document the configurations I utilise with Emacs.

As a literate program file with Org-mode, I am ensured optimal navigation through my ever growing configuration files, ease of usability and reference for peers, and, most importantly, better maintainability for myself!

Dear reader, when encountering a foregin command `X` I encourage you to execute (`describe-symbol 'X`), or press `C-h o` with the cursor on `X`. An elementary Elisp Cheat Sheet can be found here.

1 Why Emacs?

Emacs is a flexible platform for developing end-user applications –unfortunately it is generally perceived as merely a text editor. Some people use it specifically for one or two applications.

For example, writers use it as an interface for Org-mode and others use it as an interface for version control with Magit. Org is an organisation tool that can be used for typesetting which subsumes LaTeX, generating many different formats –html, latex, pdf, etc– from a single source, keeping track of schedules & task management, blogging, habit tracking, personal information management tool, and much more. Moreover, its syntax is so natural that most people use it without even knowing! For me, Org allows me to do literate programming: I can program and document at the same time, with no need to seperate the two tasks and with the ability to generate multiple formats and files from a single file.

> If you are a professional writer. . . Emacs outshines all other editing software in approximately the same way that the noonday sun does the stars. It is not just bigger and brighter; it simply makes everything else vanish. —Neal Stephenson, *In the beginning was the command line*

Of course Emacs comes with the basic features of a text editor, but it is much more; for example, it comes with a powerful notion of 'undo': Basic text editors have a single stream of undo, yet in Emacs, we have a tree –when we undo and make new edits, we branch off in our editing stream as if our text was being version controlled as we type! –We can even switch between such branches!

```
;; Allow tree-semantics for undo operations.
(package-install 'undo-tree)
(global-undo-tree-mode)
(diminish 'undo-tree-mode)

;; Execute (undo-tree-visualize) then navigate along the tree to witness
;; changes being made to your file live!

;; Each node in the undo tree should have a timestamp.
(setq undo-tree-visualizer-timestamps t)

;; Show a diff window displaying changes between undo nodes.
(setq undo-tree-visualizer-diff t)
```

Emacs is an extensible editor: You can make it into the editor of your dreams! You can make it suited to your personal needs. If there's a feature you would like, a behaviour your desire, you can simply code that into Emacs with a bit of Lisp. As a programming language enthusiast, for me Emacs is my default Lisp interpreter and a customisable IDE that I use for other programming languages –such as C, Haskell, Agda, Racket, and Prolog. Moreover, being a Lisp interpreter, we can alter the look and feel of Emacs live, without having to restart it –e.g., press `C-x C-e` after the final parenthesis of (`scroll-bar-mode 0`) to run the code that removes the scroll-bar.

> *I use Emacs every day. I rarely notice it. But when I do, it usually brings me joy.* Norman Walsh

I have used Emacs as an interface for developing cheat sheets, for making my blog, and as an application for 'interactively learning C'. If anything Emacs is more like an OS than just a text editor –"living within Emacs"

3

provides an abstraction over whatever operating system my machine has: It's so easy to take everything with me. Moreover, the desire to mould Emacs to my needs has made me a better programmer: I am now a more literate programmer and, due to Elisp's documentation-oriented nature, I actually take the time and effort to make meaningful documentation –even when the project is private and will likely only be seen by me.

> *Seeing Emacs as an editor is like seeing a car as a seating-accommodation.* – Karl Voit

Some possibly interesting reads:

- How to Learn Emacs: A Hand-drawn One-pager for Beginners / A visual tutorial

- Emacs org-mode examples and cookbook

- An Opinionated Emacs guide for newbies and beyond

- Emacs Mini-Manual, Part I of III

- Org and R Programming —a tutorial on literate programming, e.g., evaluating code within `src` bloc.

- Reference cards for GNU Emacs, Org-mode, and Elisp.

- "When did you start using Emacs" discussion on Reddit

- "How to Learn Emacs"

- Video Series on Why Emacs Rocks

- The Org-mode Reference Manual or Worg: Community-Written Docs which includes a meta-tutorial.

- Awesome Emacs: A community driven list of useful Emacs packages, libraries and others.

- A list of people's nice emacs config files

Remember: Emacs is a flexible platform for developing end-user applications; e.g., this configuration file is at its core an Emacs Lisp program that yields the editor of my dreams –it encourages me to grow and to be creative, and I hope the same for all who use it; moreover, it reflects my personality such as what I value and what I neglect in my workflow.

Moreover, as will be shown below, you can litterally use Emacs anywhere for textually input in your operating system –no copy-paste required.

Finally, here's some fun commands to try out:

- `M-x doctor` —generalising the idea of rubber ducks

- `M-x tetris` or `M-x gomoku` —a break with a classic

- `M-x butterfly` —in reference to "real programmers"

2 Booting Up

Let's always load local variables that we've marked as safe. (I tend to use loads of such locals!)

```
(setq enable-local-variables :safe)
```

2.1 ~/.emacs vs. init.org

Why not keep Emac's configurations in the `~/.emacs` file? This is because the Emacs system may explicitly add, or alter, code in it.

For example, execute the following

1. `M-x customize-variable RET line-number-mode RET`

2. Then press: `toggle`, `state`, then `1`.

3. Now take a look: `(find-file "~/.emacs")`

Notice how additions to the file have been created by 'custom'.

As such, I've chosen to write my Emacs' initialisation configurations in a file named `~/.emacs.d/init.org`: I have a literate configuration which is then loaded using org-mode's tangling feature. Read more about Emacs' initialisation configurations here.

Off topic, I love tiling window managers and had been using xmonad until recently when I obtained a mac machine and now use Amethyst – "Tiling window manager for macOS along the lines of xmonad."

Let the Emacs' gui insert default configurations and customisation into its own file, not touching or altering my initialisation file. For example, I tend to have local variables to produce `README.md`'s and other matters, so Emacs' Custom utility will remember to not prompt me each time for the safety of such local variables.

```
;; (eshell-command "touch ~/.emacs.d/custom.el")

(setq custom-file "~/.emacs.d/custom.el")
(load custom-file)
```

Rather than manually extracting the Lisp code from this literate document each time we alter it, let's instead add a 'hook' a method that is invoked on a particular event, in this case when we save the file.

```
(defun my/make-init-el-and-README ()
  (interactive)
  ;; Make init.el
  (org-babel-tangle)
  (byte-compile-file "~/.emacs.d/init.el")
  (load-file "~/.emacs.d/init.el")

  ;; Make README.md
  (save-excursion
    (org-babel-goto-named-src-block "make-readme")
    (org-babel-execute-src-block))

  (message "Tangled, compiled, and loaded init.el; and made README.md")
)

(add-hook 'after-save-hook 'my/make-init-el-and-README nil 'local-to-this-file-please)
```

Where the following block has `#+NAME: make-readme` before it. This source block generates the `README` for the associated github repository.

```
(with-temp-buffer
    (insert
    "#+EXPORT_FILE_NAME: README.md
     #+HTML: <h1> A Life Configuring Emacs </h1>

     <p align=\"center\"><img src=\"emacs-logo.png\" width=150 height=150/></p>
```

```
    <p align=\"center\">
       <a href=\"https://www.gnu.org/software/emacs/\">
            <img src=\"https://img.shields.io/badge/GNU%20Emacs-26.1-b48ead.svg?style=plastic\"/></a>
       <a href=\"https://orgmode.org/\"><img src=\"https://img.shields.io/badge/org--mode-9.2.3-489a9:
    </p>

    #+HTML: <h3> My Literate Setup </h3>
    #+OPTIONS: toc:nil d:nil
    # Toc is displayed below at a strategic position.

    I enjoy reading others' /literate/ configuration files and incorporating what I learn
    into my own. The result is a sufficiently well-documented and accessible read that yields
    a stylish and functional system (••)

    This ~README.md~ has been automatically generated from my configuration
    and its contents below could also be read in blog format, with /colour/, or as colourful PDF,
    [[https://alhassy.github.io/init/][here]]. Enjoy :smile:

     #+TOC: headlines 2
     #+INCLUDE: init.org
    ")
    (org-mode)
    (org-md-export-to-markdown)
    ;; Coloured html does not work in Github, afaik.
    ;; (org-html-export-to-html)
    ;; (shell-command "mv README.html README.md")
)
```

2.2 use-package –The start of `init.el`

There are a few ways to install packages –run `C-h C-e` for a short overview. The easiest, for a beginner, is to use the command `package-list-packages` then find the desired package, press `i` to mark it for installation, then install all marked packages by pressing `x`.

Alternatively, one uses the declarative configuration tool use-package –a meta-package that manages other packages and the way they interact.

Background: Recently I switched to mac –first time trying the OS. I had to do a few `package-install`'s and it was annoying. I'm looking for the best way to package my Emacs installation –inlcuding my installed pacakages and configuration– so that I can quickly install it anywhere, say if I go to another machine. It seems `use-package` allows me to configure and auto install packages. On a new machine, when I clone my `.emacs.d` and start emacs, on the first start it should automatically install and compile all of my packages through `use-package` when it detects they're missing.

First we need the basic `package` module which not only allows us to obtain `use-package` but acts as its kernel.

```
;; Make all commands of the "package" module present.
(require 'package)

;; Speef up start up by not loading any packages at startup.
;; (setq package-enable-at-startup nil)
;; Look at the *Messages* buffer before setting this to nil, then after.

;; Internet repositories for new packages.
(setq package-archives '(("org"        . "https://orgmode.org/elpa/")
                         ("gnu"        . "https://elpa.gnu.org/packages/")
```

```
                        ("melpa"       . "https://melpa.org/packages/")
                        ("melpa-stable" . "https://stable.melpa.org/packages/")
                        ;; Maintainer is AWOL.
                        ;; ("marmalade" . "https://marmalade-repo.org/packages/")
                        ))
```

```
;; Actually get "package" to work.
(package-initialize)
```

We can now:

- `M-x list-packages` to see all melpa packages that can install

    – Not in alphabetical order, so maybe search with `C-s`.

- For example to download the haskell mode: `M-x package-install RET haskell-mode RET`.

    – Or maybe to install `unicode-fonts` ;-)

- Read more at `http://ergoemacs.org/emacs/emacs_package_system.html` or at `https://github.com/milkypostma` `melpa`

We now bootstrap `use-package`,

```
;; Unless it's already installed, update the packages archives,
;; then install the most recent version of "use-package".
(unless (package-installed-p 'use-package)
  (package-refresh-contents)
  (package-install 'use-package))
```

```
(require 'use-package)
```

We can now invoke (`use-package XYZ :ensure t`) which should check for the `XYZ` package and make sure it is accessible. If not, the `:ensure t` part tells `use-package` to download it –using `package.el`– and place it somewhere accessible, in `~/.emacs.d/elpa/` by default.

By default we would like to download packages, since I do not plan on installing them manually by download `.el` files and placing them in the correct places on my system.

```
(setq use-package-always-ensure t)
```

Here's an example use of `use-package`. Below I have my "show recent files pop-up" command set to `C-x C-r`; but what if I forget? This mode shows me all key completions when I type `C-x`, for example. Moreover, I will be shown other commands I did not know about! Neato :-)

```
;; Making it easier to discover Emacs key presses.
(use-package which-key
 :diminish which-key-mode
 :init (which-key-mode)
 :config (which-key-setup-side-window-bottom)
         (setq which-key-idle-delay 0.05)
)
```

The `:diminish` keyword indicates that we do not want the mode's name to be shown to us in the modeline –the area near the bottom of Emacs. It does so by using the `diminish` package, so let's install that.

```
(use-package diminish)
```

```
;; Let's hide some markers.
(diminish 'eldoc-mode)
(diminish 'org-indent-mode)
(diminish 'subword-mode)
```

Here are other packages that I want to be installed onto my machine.

```
;; Efficient version control.
(use-package magit
  :config (global-set-key (kbd "C-x g") 'magit-status)
)

(use-package htmlize)
;; Main use: Org produced htmls are coloured.
;; Can be used to export a file into a coloured html.

(use-package biblio)      ;; Quick BibTeX references, sometimes.

;; Get org-headers to look pretty! E.g., * → , ** ◯, ***
;; https://github.com/emacsorphanage/org-bullets
(use-package org-bullets)
(add-hook 'org-mode-hook 'org-bullets-mode)

(use-package haskell-mode)

(use-package dash)    ;; "A modern list library for Emacs"
(use-package s  )     ;; "The long lost Emacs string manipulation library".
```

Note:

- dash: "A modern list library for Emacs"

  - E.g., `(--filter (> it 10) (list 8 9 10 11 12))`

- s: "The long lost Emacs string manipulation library".

  - E.g., `s-trim, s-replace, s-join`.

Finally, since I've symlinked my `.emacs`:

```
;; Don't ask for confirmation when opening symlinked files.
(setq vc-follow-symlinks t)
```

2.3   `magit` –Emacs' porcelain interface to git

Why use `magit` as the interface to the git version control system? In a magit buffer nearly everything can be acted upon: Press `return,` or `space`, to see details and `tab` to see children items, usually.

Below is my personal quick guide to working with magit. A quick magit tutorial can be found on jr0cket's blog

`magit-init` Put a project under version control. The mini-buffer will prompt you for the top level folder version. A `.git` folder will be created there.

`magit-status` , `C-x g` See status in another buffer. Press `?` to see options, including:

> `q` Quit magit, or go to previous magit screen.
>
> `s` Stage, i.e., add, a file to version control. Add all untracked files by selecting the *Untracked files* title.
>
> `k` Kill, i.e., delete a file locally.
>
> `K` This' (magit-file-untrack) which does `git rm --cached`.
>
> `i` Add a file to the project `.gitignore` file. Nice stuff =)
>
> `u` Unstage a specfif staged change highlighed by cursor. `C-u s` stages everything –tracked or not.

c Commit a change.

- A new buffer for the commit message appears, you write it then commit with `C-c C-c` or otherwise cancel with `C-c C-k`. These commands are mentioned to you in the minibuffer when you go to commit.
- You can provide a commit to *each* altered chunk of text! This is super neat, you make a series of local such commits rather than one nebulous global commit for the file. The `magit` interface makes this far more accessible than a standard terminal approach!
- You can look at the unstaged changes, select a *region*, using `C-SPC` as usual, and commit only that if you want!
- When looking over a commit, `M-p/n` to efficiently go to previous or next altered sections.
- Amend a commit by pressing `a` on `HEAD`.

d Show differences, another `d` or another option.

- This is magit! Each hunk can be acted upon; e.g., `s` or `c` or `k` ;-)
- The staging area is akin to a pet store; commiting is taking the pet home.

v Revert a commit.

x Undo last commit. Tantamount to `git reset HEAD~` when cursor is on most recent commit; otherwise resets to whatever commit is under the cursor.

l Show the log, another `l` for current branch; other options will be displayed.

- Here `space` shows details in another buffer while cursour remains in current buffer and, moreover, continuing to press `space` scrolls through the other buffer! Neato.

P Push.

F Pull.

: Execute a raw git command; e.g., enter `whatchanged`.

The status buffer may be refereshed using `g`, and all magit buffer by `G`.

Press `tab` to see collapsed items, such as what text has been changed.

Notice that every time you press one of these commands, a 'pop-up' of realted git options appears! Thus not only is there no need to memorize many of them, but this approach makes discovering other commands easier.

Use `M-x (magit-list-repositories) RET` to list local repositories:

Below are the git repos I'd like to clone.

```
(use-package magit)

;; Do not ask about this variable when cloning.
(setq magit-clone-set-remote.pushDefault t)

(cl-defun maybe-clone (remote &optional (local (concat "~/" (file-name-base remote))))
  "Clone a 'remote' repository if the 'local' directory does not exist.
    Yields 'nil' when no cloning transpires, otherwise yields ''cloned-repo''.

    'local' is optional and defaults to the base name; e.g.,
    if 'remote'is 'https://github.com/X/Y' then 'local' becomes '~/Y'.
  "
  (if (file-directory-p local)

    'repo-already-exists

    (async-shell-command (concat "git clone " remote " " local))
    (add-to-list 'magit-repository-directories `(,local   . 0))
```

```
      'cloned-repo)
)

;; Set variable without asking.
(setq magit-clone-set-remote.pushDefault 't)

;; Public repos
(maybe-clone "https://github.com/alhassy/emacs.d" "~/.emacs.d")
(maybe-clone "https://github.com/alhassy/alhassy.github.io")
(maybe-clone "https://github.com/alhassy/ElispCheatSheet")
(maybe-clone "https://github.com/alhassy/CatsCheatSheet")
(maybe-clone "https://github.com/alhassy/org-agda-mode")
(maybe-clone "https://github.com/JacquesCarette/TheoriesAndDataStructures")
(maybe-clone "https://github.com/alhassy/islam")
```

Let's always notify ourselves of a file that has uncommited changes –we might have had to step away from the computer and forgotten to commit.

```
(require 'magit-git)

(defun my/magit-check-file-and-popup ()
  "If the file is version controlled with git
  and has uncommitted changes, open the magit status popup."
  (let ((file (buffer-file-name)))
    (when (and file (magit-anything-modified-p t file))
      (message "This file has uncommited changes!")
      (when nil ;; Became annyoying after some time.
      (split-window-below)
      (other-window 1)
      (magit-status)))))

;; I usually have local variables, so I want the message to show
;; after the locals have been loaded.
(add-hook 'find-file-hook
  '(lambda ()
      (add-hook 'hack-local-variables-hook 'my/magit-check-file-and-popup)
  ))
```

Let's try this out:

```
(progn (eshell-command "echo change-here >> ~/dotfiles/.emacs")
       (find-file "~/dotfiles/.emacs")
)
```

In doubt, execute `C-h e` to jump to the `*Messages*` buffer.

2.4   Fix spelling as you type –thesaurus & dictionary too!

I would like to check spelling by default.

`C-;` Cycle through corrections for word at point.

`M-$` Check and correct spelling of the word at point

`M-x ispell-change-dictionary RET TAB` To see what dictionaries are available.

```
(use-package flyspell
  :hook (
          (prog-mode . flyspell-prog-mode)
          (text-mode . flyspell-mode))
)
```

Enabling fly-spell for text-mode enables it for org and latex modes since they derive from text-mode.

Flyspell needs a spell checking tool, which is not included in Emacs. We install `aspell` spell checker using, say, homebrew via `brew install aspell`. Note that Emacs' `ispell` is the interface to such a command line spelling utility.

```
(setq ispell-program-name "/usr/local/bin/aspell")
(setq ispell-dictionary "en_GB") ;; set the default dictionary

(diminish 'flyspell-mode) ;; Don't show it in the modeline.
```

Let us select a correct spelling merely by clicking on a word.

```
(eval-after-load "flyspell"
  ' (progn
     (define-key flyspell-mouse-map [down-mouse-3] #'flyspell-correct-word)
     (define-key flyspell-mouse-map [mouse-3] #'undefined)))
```

Colour incorrect works; default is an underline.

```
(global-font-lock-mode t)
(custom-set-faces '(flyspell-incorrect ((t (:inverse-video t)))))
```

Finally, save to user dictionary without asking:

```
(setq ispell-silently-savep t)
```

Let's keep track of my personal word set by having it be in my version controlled .emacs directory. Note that the default location is `~/.[i|a]spell.DICT` for a specified dictionary `DICT`.

```
(setq ispell-personal-dictionary "~/.emacs.d/.aspell.en.pws")
```

Nowadays, I very rarely write non-literate programs, but if I do I'd like to check spelling only in comments/strings. E.g.,

```
(add-hook        'c-mode-hook 'flyspell-prog-mode)
(add-hook 'emacs-lisp-mode-hook 'flyspell-prog-mode)
```

Use the thesaurus Emacs frontend Synosaurus to avoid unwarranted repetition.

```
(use-package synosaurus
  :diminish synosaurus-mode
  :init    (synosaurus-mode)
  :config  (setq synosaurus-choose-method 'popup) ;; 'ido is default.
           (global-set-key (kbd "M-#") 'synosaurus-choose-and-replace)
)
```

The thesaurus is powered by the Wordnet `wn` tool, which can be invoked without an internet connection!

```
;; (shell-command "brew cask install xquartz &") ;; Dependency
;; (shell-command "brew install wordnet &")
```

Let's use Wordnet as a dictionary via the wordnut package.

```
(use-package wordnut
 :bind ("M-!" . wordnut-lookup-current-word))
```

```
;; Use M-& for async shell commands.
```

Use `M-↑,↓` to navigate dictionary results, and `wordnut-search` for a new search.

Use this game to help you learn to spell words that you're having trouble with; see `~/Dropbox/spelling.txt`.

```
(autoload 'typing-of-emacs "~/.emacs.d/typing.el" "The Typing Of Emacs, a game." t)
```

Practice touch typing using speed-type.

```
(use-package speed-type)
```

Running `M-x speed-type-region` on a region of text, or `M-x speed-type-buffer` on a whole buffer, or just `M-x speed-type-text` will produce the selected region, buffer, or random text for practice. The timer begins when the first key is pressed and stats are shown when the last letter is entered.

Other typing resources include:

- Typing of Emacs –an Emacs alternative to speed type, possibly more engaging.

- Klavaro –a GUI based yet language-independent typing tutor.

    - I'm enjoying this tool in getting started with Arabic typing.
    - The plan is to move to using the online Making Hijrah tutor which concludes the basic lesson plan with a few short narrations.

- Typing.io is a tutor for coders: Lessons are based on open source code, such some XMonad written in Haskell or Linux written in C.

- GNU Typist –which is interactive in the terminal, so not ideal in Emacs–,

To assist in language learning, it may be nice to have an Emacs interface to Google translate —e.g., invoke `google-translate-at-point`.

```
(use-package google-translate
 :config
   (global-set-key "\C-ct" 'google-translate-at-point)
)
```

Select the following then `C-c t`,

Hey buddy, what're you up to?

Then *detect language* then *Arabic* to obtain:

Neato

2.5   Using a Grammar & Style Checker

Let's install a grammar and style checker. We get the offline tool from the bottom of the LanguageTool website, then relocate it as follows.

```
(use-package langtool
 :config
  (setq langtool-language-tool-jar
     "~/Applications/LanguageTool-4.5/languagetool-commandline.jar")
)
```

Now we can run `langtool-check` on the subsequent grammatically incorrect text –which is from the LanguageTool website– which colours errors in red, when we click on them we get the reason why; then we may invoke `langtool-correct-buffer` to quickly use the suggestions to fix each correction, and finally invoke `langtool-check-done` to stop any remaining red colouring.

```
LanguageTool offers spell and grammar checking. Just paste your text here
and click the 'Check Text' button. Click the colored phrases for details
on potential errors. or use this text too see an few of of the problems
that LanguageTool can detecd. What do you thinks of grammar checkers?
Please not that they are not perfect. Style issues get a blue marker:
It's 5 P.M. in the afternoon. The weather was nice on Thursday, 27 June 2017
--uh oh, that's the wrong date ;-)
```

By looking around the source code, I can do all three stages smoothly (●●)

```
;; Quickly check, correct, then clean up /region/ with M-^

(add-hook 'langtool-error-exists-hook
  (lambda ()
    (langtool-correct-buffer)
    (langtool-check-done)
  ))


(global-set-key "\M-^" 'langtool-check)
```

2.6   Unicode Input via Agda Input

Agda is one of my favourite languages, it's like Haskell on steroids. Let's set it up.
Executing `agda-mode setup` appends the following text to the `.emacs` file. Let's put it here ourselves.

```
(load-file (let ((coding-system-for-read 'utf-8))
             (shell-command-to-string "/usr/local/bin/agda-mode locate")))
```

I almost always want the `agda-mode` input method.

```
(require 'agda-input)
(add-hook 'text-mode-hook (lambda () (set-input-method "Agda")))
(add-hook 'org-mode-hook (lambda () (set-input-method "Agda")))
```

Below are my personal Agda input symbol translations; e.g., \set → . Note that we could give a symbol new Agda TEX binding interactively: M-x `customize-variable agda-input-user-translations` then INS then for key sequence type `set` then INS and for string paste .

```
;; category theory
(add-to-list 'agda-input-user-translations '("set" ""))
(add-to-list 'agda-input-user-translations '("alg" ""))
(add-to-list 'agda-input-user-translations '("split" ""))
(add-to-list 'agda-input-user-translations '("join" ""))
(add-to-list 'agda-input-user-translations '("adj" ""))
(add-to-list 'agda-input-user-translations '(";;" ""))
(add-to-list 'agda-input-user-translations '(";;" ""))
(add-to-list 'agda-input-user-translations '(";;" ""))


;; lattices
(add-to-list 'agda-input-user-translations '("meet" ""))
```

```
(add-to-list 'agda-input-user-translations '("join" ""))

;; residuals
(add-to-list 'agda-input-user-translations '("syq"   ""))
(add-to-list 'agda-input-user-translations '("over" ""))
(add-to-list 'agda-input-user-translations '("under" ""))
        ;; Maybe "\\" shortcut?

;; Z-quantification range notation, e.g., " x  R • P"
(add-to-list 'agda-input-user-translations '("|" ""))
(add-to-list 'agda-input-user-translations '("with" ""))

;; adjunction isomorphism pair
(add-to-list 'agda-input-user-translations '("floor"  ""))
(add-to-list 'agda-input-user-translations '("lower"  ""))
(add-to-list 'agda-input-user-translations '("lad"    ""))
(add-to-list 'agda-input-user-translations '("ceil"   ""))
(add-to-list 'agda-input-user-translations '("raise"  ""))
(add-to-list 'agda-input-user-translations '("rad"    ""))

;; silly stuff
;;
;; angry, cry, why-you-no
(add-to-list 'agda-input-user-translations
    '("whyme" "()" "_" "()"))
;; confused, disapprove, dead, shrug
(add-to-list 'agda-input-user-translations
    '("what" "(°°)" "(_)" "()" "¯\\_()_/¯"))
;; dance, csi
(add-to-list 'agda-input-user-translations
    '("cool" "(-_-)(-_-)(-_- )" "●_●)
( ●_●)>-
(_)
"))
;; love, pleased, success, yesss
(add-to-list 'agda-input-user-translations
    '("smile" "" "()" "(●●)" "(_)"))
```

Finally let's effect such translations.

```
;; activate translations
(agda-input-setup)
```

Note that the effect of Emacs unicode input could be approximated using `abbrev-mode`.

2.7   Syncing to the System's `$PATH`

For one reason or another, on OS X it seems that an Emacs instance begun from the terminal may not inherit the terminal's environment variables, thus making it difficult to use utilities like `pdflatex` when Org-mode attempts to produce a PDF.

```
(use-package exec-path-from-shell
  :init
    (when (memq window-system '(mac ns x))
```

```
        (exec-path-from-shell-initialize))
)
```

See these docs for setting other environment variables.

2.8 Keeping My System Up to Date

```
(defun my/stay-up-to-date ()

  "Ensure that OS and Emacs pacakges are up to date.

   Takes ~5 secons when everything is up to date.
  "

  (async-shell-command "brew update && brew upgrade")
  (other-window 1)
  (rename-buffer "Keeping-system-up-to-date")

  (package-refresh-contents)
  (insert "Emacs packages have been updated.")

  (other-window 1)
)


(add-hook 'after-init-hook 'my/stay-up-to-date)

;; For now, doing this since I'm also calling my/stay-up-to-date with
;; after-init-hook which hides the startup message.
(add-hook 'after-init-hook 'display-startup-echo-area-message)
```

2.9 Who am I? Using Gnus for Gmail

Let's set the following personal Emacs-wide variables to be used in other locations besides email.

```
(setq user-full-name    "Musa Al-hassy"
      user-mail-address "alhassy@gmail.com")
```

By default, in Emacs, we may send mail: Write it in Emacs with `C-x m`, then press `C-c C-c` to have it sent via your OS's default mailing system –mine appears to be Gmail via the browser. Or cancel sending mail with `C-c C-k` –the same commands for capturing, discussed below (●●)

To send and read email in Emacs we use GNUS –which, like many GNU itself, it a recursive acronym: GNUS Network User Service.

1. Execute, rather place in your init:

   ```
   (setq message-send-mail-function 'smtpmail-send-it)
   ```

   Revert to the default OS mailing method by setting this variable to `mailclient-send-it`.

2. Follow only the quickstart here; namely, make a file named `~/.gnus` containing:

   ```
   ;; user-full-name and user-mail-address should be defined

   (setq gnus-select-method
         '(nnimap "gmail"
                  (nnimap-address "imap.gmail.com")
   ```

15

```
                    (nnimap-server-port "imaps")
                    (nnimap-stream ssl)))

  (setq smtpmail-smtp-server "smtp.gmail.com"
        smtpmail-smtp-service 587
        gnus-ignored-newsgroups "^to\\.\\|^[0-9. ]+\\( \\|$\\)\\|^[\"]\"[#'()]")
```

3. Enable "2 step authentication" for Gmail following these instructions.

4. You will then obtain a secret password, the `x` marks below, which you insert in a file named `~/.authinfo` as follows –using your email address.

```
    machine imap.gmail.com login alhassy@gmail.com password xxxxxxxxxxxxxxxx port imaps
    machine smtp.gmail.com login alhassy@gmail.com password xxxxxxxxxxxxxxxx port 587
```

5. In Emacs, `M-x gnus` to see what's there.

   Or compose mail with `C-x m` then send it with `C-c C-c`.

   - Press `C-h m` to learn more about message mode for mail composition; or read the Message Manual.

In gnus, by default items you've looked at disappear –i.e., are archived. They can still be viewed in, say, the online browser if you like. In the `Group` view, `R` resets gnus, possibly retriving mail or alterations from other mail clients. `q` exits gnus in `Group` mode, `q` exits the particular view to go back to summary mode. Only after pressing `q` from within a group do changes take effect on articles –such as moves, reads, deletes, etc.

RET Open an article.

B m Move an article, in its current state, to another group –i.e., 'label' using Gmail parlance.

Something to consider doing when finished with an article.

To delete an article, simply move it to 'trash' –of course this will delete it in other mail clients as well. There is no return from trash.

Emails can always be achieved –never delete, maybe?

! mark an article as read, but to be kept around –e.g., you have not replied to it, or it requires more reading at a later time.

R Reply to email with sender's content there in place.

   - `r` to reply to an email with sender's content in adjacent buffer.

d mark an article as done, i.e., read it and it can be archived.

Learn more by reading The Gnus Manual; also available within Emacs by `C-h i m gnus` (●●)

- Or look at the Gnus Reference Card.

- Or, less comprehensively, this outline.

2.10 Emacs keybindings for my brower

I've downloaded the Vimium extension for Google Chrome, and have copy-pasted these Emacs key bindings into it. Now `C-h` in my browser shows which Emacs-like bindings can be used to navigate my browser ˆ_ˆ

2.11 Using Emacs in any text area on my OS

Using the Emacs-Anywhere tool, I can press `Cmd Shift e` to have an Emacs frame appear, produce text with Emacs editing capabilities, then `C-x 5 0` to have the resulting text dumped into the text area I was working in.

This way I can use Emacs literally anywhere for textual input!

For my Mac OSX:

```
(shell-command "curl -fsSL https://raw.github.com/zachcurry/emacs-anywhere/master/install | bash")
```

```
(server-start)
```

The tools that use emacs-anywhere –such as my web browser– and emacs-anywhere itself need to be given sufficient OS permissions:

```
System Preferences → Security & Privacy → Accessibility
```

Then check the emacs-anywhere box from the following gui and provide a keyboard shortcut:

```
System Preferences → Keyboard → Shortcuts → Services
```

(●●)

I always want to be in Org-mode and input unicode:

```
(add-hook 'ea-popup-hook
  (lambda (app-name window-title x y w h)
    (org-mode)
    (set-input-method "Agda")
  )
)
```

3 Cosmetics

```
;; Make it very easy to see the line with the cursor.
(global-hl-line-mode t)


;; Clean up any accidental trailing whitespace and in other places,
;; upon save.
(add-hook 'before-save-hook 'whitespace-cleanup)


;; Keep self motivated!
(setq frame-title-format '("" "%b - Living The Dream (●●)"))
```

3.1 Themes

```
;; Treat all themes as safe; no query before use.
(setf custom-safe-themes t)


;; Nice looking themes ^_^
(use-package solarized-theme)
(use-package doom-themes)
;; (use-package spacemacs-theme)
;; this gives me an error for some reason


(defun my/disable-all-themes ()
  (dolist (th custom-enabled-themes)
        (disable-theme th))
```

```elisp
)

(defun my/load-dark-theme ()
  ;;   (load-theme 'spacemacs-dark)    ;; orginally
  (my/disable-all-themes)
  (load-theme 'doom-vibrant)
)

(defun my/load-light-theme ()
  ;;   (load-theme 'spacemacs-light)    ;; orginally
  ;; Recently I'm liking this ordered mixture.
  (load-theme 'solarized-light) (load-theme 'doom-solarized-light)
)

;; "C-x t" to toggle between light and dark themes.
(defun my/toggle-theme () "Toggle between dark and light themes."
  (interactive)
  ;; Load dark if light is top-most enabled theme, else load light.
  (if (equal (car custom-enabled-themes) 'doom-vibrant)
      (my/load-light-theme)
      (my/load-dark-theme)
  )

  ;; The dark theme's modeline separator is ugly.
  ;; Keep reading below regarding "powerline".
  ;; (setq powerline-default-separator 'arrow)
  ;; (spaceline-spacemacs-theme)
)

(global-set-key "\C-x\ t" 'my/toggle-theme)

;; Initially begin with the light theme.
(load-theme 'spacemacs-light t)
```

The Doom Themes also look rather appealing. A showcase of many themes can be found here.

3.2 Startup message: Emacs & Org versions

```elisp
;; Silence the usual message: Get more info using the about page via C-h C-a.
(setq inhibit-startup-message t)

(defun display-startup-echo-area-message ()
  "The message that is shown after 'user-init-file' is loaded."
  (message
      (concat "Welcome "       user-full-name
              "! Emacs "       emacs-version
              "; Org-mode "   org-version
              "; System "     (system-name)
                  (format "; Time %.3fs"
                      (float-time (time-subtract (current-time)
                                      before-init-time)))
      )
  )
)
```

Now my startup message is,

```
;; Welcome Musa Al-hassy! Emacs 26.1; Org-mode 9.2.3; System alhassy-air.local
```

For some fun, run this cute method.

```
(animate-birthday-present user-full-name)
```

Moreover, since I end up using org-mode most of the time, let's make that the default mode.

```
(setq initial-major-mode 'org-mode)
```

3.3   Persistent Scratch Buffer

The `*scratch*` buffer is a nice playground for temporary data.
I use Org-mode often, so that's how I want things to appear.

```
(setq initial-scratch-message (concat
  "#+Title: Persistent Scratch Buffer"
  "\n#\n # Welcome! This' a place for trying things out. \n"))
```

We might accidentally close this buffer, so we could utilise the following.

```
;; A very simple function to recreate the scratch buffer:
;; ( http://emacswiki.org/emacs/RecreateScratchBuffer )
(defun scratch ()
   "create a scratch buffer"
   (interactive)
   (switch-to-buffer-other-window (get-buffer-create "*scratch*"))
   (insert initial-scratch-message)
   (org-mode))


;; This doubles as a quick way to avoid the common formula: C-x b RET *scratch*
```

However, by default its contents are not saved –which may be an issue if we have not relocated our playthings
to their appropriate files. Whence let's save & restore the scratch buffer by default.

```
(use-package persistent-scratch
  :config
  (persistent-scratch-setup-default))
```

3.4   Spaceline: A sleek mode line

I may not use the spacemacs starter kit, since I do not like evil-mode and find spacemacs to "hide things" from me
–whereas Emacs ""encourages" me to learn more–, however it is a configuration and I enjoy reading Emacs configs
in order to improve my own setup. From Spacemacs I've adopted Helm for list completion, its sleek light & dark
themes, and its modified powerline setup.
    The 'modeline' is a part near the bottom of Emacs that gives information about the current mode, as well as
other matters –such as time & date, for example.

```
(use-package spaceline
  :config
  (require 'spaceline-config)
  (setq spaceline-buffer-encoding-abbrev-p nil)
  (setq spaceline-line-column-p nil)
  (setq spaceline-line-p nil)
  (setq powerline-default-separator 'arrow)
  :init
```

```
 (spaceline-helm-mode) ;; When using helm, mode line looks prettier.
 (spaceline-spacemacs-theme)
)
```

Other separators I've considered include `'brace` instead of an arrow, and `'contour`, `'chamfer`, `'wave`, `'zigzag` which look like browser tabs that are curved, boxed, wavy, or in the style of driftwood.

3.5 Flashing when something goes wrong no blinking

Make top and bottom of screen flash when something unexpected happens thereby observing a warning message in the minibuffer. E.g., C-g, or calling an unbound key sequence, or misspelling a word.

```
(setq visible-bell 1)
;; Enable flashing mode-line on errors
;; On MacOS, this shows a caution symbol ^_^

;; Blinking cursor rushes me to type; let's slow down.
(blink-cursor-mode -1)
```

3.6 My to-do list: The initial buffer when Emacs opens up

```
(find-file "~/Dropbox/todo.org")
;; (setq initial-buffer-choice "~/Dropbox/todo.org")

(split-window-right)                        ;; C-x 3
(other-window 1)                            ;; C-x 0
;; toggle enable-local-variables :all          ;; Load *all* locals.
    ;; toggle org-confirm-babel-evaluate nil    ;; Eval *all* blocks.
      (find-file "~/.emacs.d/init.org")
```

3.7 Showing date, time, and battery life

```
(setq display-time-day-and-date t)
(display-time)

;; (display-battery-mode 1)
;; Nope; let's use a fancy indicator ...

(use-package fancy-battery
  :diminish
  :config
    (setq fancy-battery-show-percentage t)
    (setq battery-update-interval 15)
    (fancy-battery-mode)
    (display-battery-mode)
)
```

This will show remaining battery life, coloured green if charging and coloured yellow otherwise. It is important to note that this package is no longer maintained. It works on my machine.

3.8 Hiding Scrollbar, tool bar, and menu

```
(tool-bar-mode -1)
(scroll-bar-mode -1)
(menu-bar-mode -1)
```

3.9 Increase/decrease text size

```
(global-set-key (kbd "C-+") 'text-scale-increase)
(global-set-key (kbd "C--") 'text-scale-decrease)
  ;; C-x C-0 restores the default font size


(add-hook 'text-mode-hook
          '(lambda ()
             (visual-line-mode 1)
                 (diminish 'visual-line-mode)
             ))
```

3.10 Delete Selection mode

Delete Selection mode lets you treat an Emacs region much like a typical text selection outside of Emacs: You can replace the active region. We can delete selected text just by hitting the backspace key.

```
  (delete-selection-mode 1)
```

3.11 Highlight & complete parenthesis pair when cursor is near ;-

```
;; Highlight expression within matching parens when near one of them.
(setq show-paren-delay 0)
(setq blink-matching-paren nil)
(setq show-paren-style 'expression)
(show-paren-mode)


;; Colour parens, and other delimiters, depending on their depth.
;; Very useful for parens heavy languages like Lisp.
(use-package rainbow-delimiters)


(add-hook 'org-mode-hook
  '(lambda () (rainbow-delimiters-mode 1)))
(add-hook 'prog-mode-hook
  '(lambda () (rainbow-delimiters-mode 1)))
```

For example,

```
(blue (purple (forest (green (yellow (blue))))))
```

There is a powerful package called 'smartparens' for working with pair-able characters, but I've found it to be too much for my uses. Instead I'll utilise the lightweight package `electric`, which provided by Emacs out of the box.

```
(electric-pair-mode 1)
```

It supports, by default, ACSI pairs `{}, [], ()` and Unicode ' ', " ", , .

3.12 Minibuffer should display line and column numbers

```
(global-display-line-numbers-mode t)
; (line-number-mode t)
(column-number-mode t)
```

3.13 Neotree: Directory Tree Listing

We open a nifty file manager upon startup.

```
;; neotree --sidebar for project file navigation
(use-package neotree
  :config (global-set-key "\C-x\ d" 'neotree-toggle))

;; Only do this once:
(when nil
  (use-package all-the-icons)
  (all-the-icons-install-fonts 'install-without-asking))

(setq neo-theme 'icons)
(neotree-refresh)


;; Open it up upon startup.
(neotree-toggle)
```

By default `C-x d` invokes `dired`, but I prefer `neotree` for file management.
  Useful navigational commands include

- `U` to go up a directory.

- `C-c C-c` to change directory focus; `C-C c` to type the directory out.

- `?` or `h` to get help and `q` to quit.

  As always, to go to the neotree pane when it's the only other window, execute `C-x o`.
  I *rarely* make use of this feature; company mode & Helm together quickly provide an automatic replacement for nearly all of my uses.

3.14 Tabs   DISABLED

I really like my Helm-supported `C-x b`, but the visial appeal of a tab bar for Emacs is interesting. Let's try it out and see how long this lasts —it may be like Neotree: Something cute to show to others, but not as fast as the keyboard.

```
; (async-shell-command
;   "git clone --depth=1 https://github.com/manateelazycat/awesome-tab.git  ~/.emacs.d/elpa/awesome-tab

(load-file "~/.emacs.d/elpa/awesome-tab/awesome-tab.el")

;; Show me /all/ the tabs at once, in one group.
(defun awesome-tab-buffer-groups ()
  (list (awesome-tab-get-group-name (current-buffer))))


(awesome-tab-mode t)
```

  It's been less than three days and I've found this utility to be unhelpful, to me anyhow.

3.15 Window resizing using the golden ratio   DISABLED

Let's load the following package, which automatically resizes windows so that the window containing the cursor is the largest, according to the golden ratio. Consequently, the window we're working with is nice and large yet the other windows are still readable.

```
(use-package golden-ratio
  :diminish golden-ratio-mode
  :init (golden-ratio-mode 1))
```

After some time this got a bit annoying and I'm no longer using this.

4 Life within Org-mode

Let's obtain Org-mode along with the extras that allow us to ignore heading names, but still utilise their contents –e.g., such as a heading named 'preamble' that contains org-mode setup for a file.

```
(use-package org
  :ensure org-plus-contrib
  :config
  (require 'ox-extra)
  (ox-extras-activate '(ignore-headlines)))
```

This lets us use the `:ignore:` tag on headlines you'd like to have ignored, while not ignoring their content –see here.

- Use the `:noexport:` tag to omit a headline *and* its contents.

Now, let's replace the content marker, "", with a nice unicode arrow.

```
(setq org-ellipsis " ")
```

Also:

```
;; Fold all source blocks on startup.
(setq org-hide-block-startup t)

;; Avoid accidentally editing folded regions, say by adding text after an Org "".
(setq org-catch-invisible-edits 'show)

;; I use indentation-sensitive programming languages.
;; Tangling should preserve my indentation.
(setq org-src-preserve-indentation t)

;; Tab should do indent in code blocks
(setq org-src-tab-acts-natively t)

;; Give quote and verse blocks a nice look.
(setq org-fontify-quote-and-verse-blocks t)

;; Pressing ENTER on a link should follow it.
(setq org-return-follows-link t)
```

I rarely use tables, but here is a useful Org-Mode Table Editing Cheatsheet.

4.1 High Speed Literate Programming

4.1.1 Manipulating Sections

Let's enable the Org Speed Keys so that when the cursor is at the beginning of a headline, we can perform fast manipulation & navigation using the standard Emacs movement controls, such as

- `#` toggle `COMMENT`-ing for an org-header.

- `s` toggles "narrowing" to a subtree; i.e., hide the rest of the document.

  If you narrow to a subtree then any export, `C-c C-e`, will only consider the narrowed detail.

- `I/O` clock In/Out to the task defined by the current heading.

  - Keep track of your work times!
  - `v` view agenda.

- `u` for jumping upwards to the parent heading.

- `c` for cycling structure below current heading, or `C` for cycling global structure.

- `i` insert a new same-level heading below current heading.

- `w` refile current heading; options list pops-up to select which heading to move it to. Neato!

- `t` cycle through the available TODO states.

- `^` sort children of current subtree; brings up a list of sorting options.

- `n/p` for next/previous *visible* heading.

- `f/b` for jumping forward/backward to the next/previous *same-level* heading.

- `D/U` move a heading down/up.

- `L/R` recursively promote (move leftwards) or demote (more rightwards) a heading.

- `1,2,3` to mark a heading with priority, highest to lowest.

We can add our own speed keys by altering the `org-speed-commands-user` variable. Moreover, `?` to see a complete list of keys available.

```
(setq org-use-speed-commands t)
```

4.1.2   Seamless Navigation Between Source Blocks

Finally, let's use the "super key" –aka the command or windows key– to jump to the previous, next, or toggle editing org-mode source blocks.

```
;; Overriding keys for printing buffer, duplicating gui frame, and isearch-yank-kill.
;;
(define-key org-mode-map (kbd "s-p") #'org-babel-previous-src-block)
(define-key org-mode-map (kbd "s-n") #'org-babel-next-src-block)
(define-key org-mode-map (kbd "s-e") #'org-edit-src-code)
(define-key org-src-mode-map (kbd "s-e") #'org-edit-src-exit)
```

Interestingly, `s-l` is "goto line".

4.1.3   Modifying <return>

- `C-RET, C-S-RET` make a new heading where the latter marks it as a `TODO`.

- By default `M-RET` makes it easy to work with existing list items, headings, tables, etc by creating a new item, heading, etc.

- Usually we want a newline then we indent, let's make that the default.

  ```
  (add-hook 'org-mode-hook '(lambda ()
    (local-set-key (kbd "<return>") 'org-return-indent))
    (local-set-key (kbd "C-M-<return>") 'electric-indent-just-newline))
  ```

Notice that I've also added another kind of return, for when I want to break-out of the indentation approach and start working at the beginning of the line.

In summary,

key	method	behaviour
`<return>`	`org-return-indent`	Newline with indentation
`M-<return>`	`org-meta-return`	Newline with new org item
`C-M-<return>`	`electric-indent-just-newline`	Newline, cursor at start
`C-<return>`	`org-insert-heading-respect-content`	New heading *after* current content
`C-S-<return>`	`org-insert-todo-heading-respect-content`	Ditto, but with a `TODO` marker

4.1.4 `C-a,e,k` and Yanking of sections

```
;; On an org-heading, C-a goes to after the star, heading markers.
;; To use speed keys, run C-a C-a to get to the star markers.
;;
;; C-e goes to the end of the heading, not including the tags.
;;
(setq org-special-ctrl-a/e t)


;; C-k no longer removes tags, if activated in the middle of a heading's name.
(setq org-special-ctrl-k t)


;; When you yank a subtree and paste it alongside a subtree of depth 'd',
;; then the yanked tree's depth is adjusted to become depth 'd' as well.
;; If you don't want this, then refile instead of copy pasting.
(setq org-yank-adjusted-subtrees t)
```

4.2 Using org-mode as a Day Planner

This section is based on a dated, yet delightful, tutorial of the same title by John Wiegley.
  We want a day-planner with the following use:

1. "Mindlessly" & rapidly create new tasks.

2. Schedule and archive tasks at the end, or start, of the work day.

3. Glance at a week's tasks, shuffle if need be.

4. Prioritise the day's tasks. Aim for 15 tasks.

5. Progress towards `A` tasks completion by documenting work completed.

6. Repeat! During the day, if anything comes up, capture it and intentionally forget about it.

Capture lets me quickly make notes & capture ideas, with associated reference material, without any interruption to the current work flow. Without losing focus on what you're doing, quickly jot down a note of something important that just came up.
  E.g., I have a task, or something I wish to note down, rather than opening some file, then making a heading, then writing it; instead, I press `C-c c t` and a pop-up appears, I make my note, and it disappears with my notes file(s) now being altered! Moreover, by default it provide a timestamp and a link to the file location where I made the note –helpful for tasks, tickets, to be tackled later on.

```
(setq org-default-notes-file "~/Dropbox/todo.org")
(define-key global-map "\C-cc" 'org-capture)
```

By default we only get a 'tasks' form of capture, let's add some more.

```
(cl-defun my/make/org-capture-template
    (shortcut heading &optional (no-todo nil) (description heading) (category heading))
  "Quickly produce an org-capture-template.

  After adding the result of this function to 'org-capture-templates',
  we will be able perform a capture with ''C-c c 'shortcut'''
  which will have description 'description'.
  It will be added to the tasks file under heading 'heading'
  and be marked with category  'category'.

  'no-todo' omits the 'TODO' tag from the resulting item; e.g.,
  when it's merely an interesting note that needn't be acted upon.
  Probably a bad idea

  Defaults for 'description' and 'category' are set to the same as
  the 'heading'. Default for 'no-todo' is 'nil'.
  "
  '(,shortcut ,description entry
      (file+headline org-default-notes-file
          ,(concat heading "\n#+CATEGORY: " category))
      , (concat "*" (unless no-todo " TODO") " %?\n:PROPERTIES:\n:CREATED: %U\n:END:\n\n")
      :empty-lines 1)
)


(setq org-capture-templates
  '(
      ,(my/make/org-capture-template "t" "Tasks, Getting Things Done")
      ,(my/make/org-capture-template "r" "Research")
      ,(my/make/org-capture-template "m" "Email")
      ,(my/make/org-capture-template "e" "Emacs (●●)")
      ,(my/make/org-capture-template "b" "Blog")
      ,(my/make/org-capture-template "a" "Arbitrary Reading and Learning")
      ,(my/make/org-capture-template "p" "Personal Matters")
))
```

For now I capture everything into a single file. One would ideally keep separate client, project, information in its own org file. The `#+CATEGORY` appears alongside each task in the agenda view –keep reading.

Where am I currently capturing?

- During meetings, when a nifty idea pops into my mind, I quickly capture it.

    – I've found taking my laptop to meetings makes me an active listener and I get much more out of my meetings since I'm taking notes.

- Through out the day, as I browse the web, read, and work; random ideas pop-up, and I capture them indiscriminately.

- I envision that for a phone call, I would open up a capture to make note of what the call entailed so I can review it later.

- Anywhere you simply want to make a note, for the current heading, just press `C-c C-z`. The notes are just your remarks along with a timestamp; they are collected at the top of the tree, under the heading.

    ```
    ;; Ensure notes are stored at the top of a tree.
    (setq org-reverse-note-order nil)
    ```

Anyhow. . .

Step 1: When new tasks come up Isn't it great that we can squirrel away info into some default location then immediately return to what we were doing before –with speed & minimal distraction!   Indeed, if our system for task management were slow then we may not produce tasks and so forget them altogether! ()

- Entering tasks is a desirably impulsive act; do not make any further scheduling considerations.

  The next step, the review stage occurring at the end or the start of the workday, is for processing.

  The reason for this is that entering new tasks should be impulsive, not reasoned./ Your reasoning skills are required for the task at hand, not every new tidbit./ You may even find that during the few hours that transpire between creating a task and categorizing it, you've either already done it or discovered it doesn't need to be done at all! – John Wiegley

When my computer isn't handy, make a note on my phone then transfer it later.

Step 2: Filing your tasks At a later time, a time of reflection, we go to our tasks list and actually schedule time to get them done by `C-c C-s` then pick a date by entering a number in the form `+n` to mean that task is due `n` days from now.

- Tasks with no due date are ones that "could happen anytime", most likely no time at all.

- At least schedule tasks reasonably far off in the future, then reassess when the time comes.

- An uncompleted task is by default rescheduled to the current day, each day, along with how overdue it is.

  - Aim to consciously reschedule such tasks!

With time, it will become clear what is an unreasonable day verses what is an achievable day.

Step 3: Quickly review the upcoming week The next day we begin our work, we press `C-c a a` to see the scheduled tasks for this week –`C-c C-s` to re-schedule the task under the cursor and `r` to refresh the agenda.

```
(define-key global-map "\C-ca" 'org-agenda)
```

Step 4: Getting ready for the day After having seen our tasks for the week, we press `d` to enter daily view for the current day. Now we decide whether the items for today are `A`: of high urgency & important; `B`: of moderate urgency & importance; or `C`: Pretty much optional, or very quick or fun to do.

- `A` tasks should be both important *and* urgently done on the day they were scheduled.

  - Such tasks should be relatively rare!
  - If you have too many, you're anxious about priorities and rendering priorities useless.

- `C` tasks can always be scheduled for another day without much worry.

  - Act! If the thought of rescheduling causes you to worry, upgrade it to a `B` or `A`.

- As such, most tasks will generally be priority `B`: Tasks that need to be done, but the exact day isn't as critical as with an `A` task. These are the "bread and butter" tasks that make up your day to day life.

On a task item, press `,` then one of `A, B, C` to set its priority. Then `r` to refresh.

Step 5: Doing the work Since `A` tasks are the important and urgent ones, if you do all of the `A` tasks and nothing else today, no one would suffer. It's a good day ().

There should be no scheduling nor prioritising at this stage. You should not be touching your tasks file until your next review session: Either at the end of the day or the start of the next.

- Leverage priorities! E.g., When a full day has several `C` tasks, reschedule them for later in the week without a second thought.

  - You've already provided consideration when assigning priorities.

Step 6: Moving a task toward completion My workflow states are described in the section 4.5 and contain states: `TODO, STARTED, WAITING, ON_HOLD, CANCELLED, DONE`.

- Tasks marked `WAITING` are ones for which we are awaiting some event, like someone to reply to our query. As such, these tasks can be rescheduled until I give up or the awaited event happens –in which case I go to `STARTED` and document the reply to my query.

- The task may be put off indefinitely with `ON_HOLD`, or I may choose never to do it with `CANCELLED`. Along with `DONE`, these three mark a task as completed and so it needn't appear in any agenda view.

I personally clock-in and clock-out of tasks –keep reading–, where upon clocking-out I'm prompted for a note about what I've accomplished so far. Entering a comment about what I've done, even if it's very little, feels like I'm getting something done. It's an explicit marker of progress.

In the past, I would make a "captain's log" at the end of the day, but that's like commenting code after it's written, I didn't always feel like doing it and it wasn't that important after the fact. The continuous approach of noting after every clock-out is much more practical, for me at least.

Step 7: Archiving Tasks During the review state, when a task is completed, 'archive' it with `C-c C-x C-s`: This marks it as done, adds a time stamp, and moves it to a local `*.org_archive` file. This was our 'to do' list becomes a 'ta da' list showcasing all we have done (●●)

Archiving keeps task lists clutter free, but unlike deletion it allows us, possibly rarely, to look up details of a task or what tasks were completed in a certain time frame –which may be a motivational act, to see that you have actually completed more than you thought, provided you make and archive tasks regularly. We can use `(org-search-view)` to search an org file *and* the archive file too, if we enable it so.

```
;; Include agenda archive files when searching for things
(setq org-agenda-text-search-extra-files (quote (agenda-archives)))

;; Invoing the agenda command shows the agenda and enables
;; the org-agenda variables.
(org-agenda "a" "a")
```

Let's install some helpful views for our agenda.

- `C-c a c`: See completed tasks at the end of the day and archive them.

  ```
  ;; Pressing 'c' in the org-agenda view shows all completed tasks,
  ;; which should be archived.
  (add-to-list 'org-agenda-custom-commands
    '("c" todo "DONE|ON_HOLD|CANCELLED" nil))
  ```

- `C-c a u`: See unscheduled, undeadlined, and undated tasks in my todo files. Which should then be scheduled or archived.

  ```
  (add-to-list 'org-agenda-custom-commands
    '("u" alltodo ""
       ((org-agenda-skip-function
          (lambda ()
               (org-agenda-skip-entry-if 'scheduled 'deadline 'regexp  "\n]+>")))
               (org-agenda-overriding-header "Unscheduled TODO entries: ")))))
  ```

4.3   Automating Pomodoro –Dealing with dreadful tasks

Effort estimates are for an entire task. Yet, sometimes it's hard to even get started on some tasks.

- The code below ensures a 25 minute timer is started whenever clocking in happens.

  - The timer is in the lower right of the modeline.

- When the timer runs out, we get a notification.

- We may have the momentum to continue on the dreadful task, or clock-out and take a break after documenting what was accomplished.

```
;; Tasks get a 25 minute count down timer
(setq org-timer-default-timer 25)

;; Use the timer we set when clocking in happens.
(add-hook 'org-clock-in-hook
  (lambda () (org-timer-set-timer '(16))))

;; unless we clocked-out with less than a minute left,
;; show disappointment message.
(add-hook 'org-clock-out-hook
  (lambda ()
  (unless (s-prefix? "0:00" (org-timer-value-string))
    (message-box "The basic 25 minutes on this dreadful task are not up; it's a shame to see you leave
    (org-timer-stop)
    ))
```

Note that this does not conflict with the total effort estimate for the task.

4.4   Journaling

Thus far I've made it easy to quickly capture ideas and tasks, not so much on the analysis phase:

- What was accomplished today?

- What are some notably bad habits? Good habits?

- What are some future steps?

Rather than overloading the capture mechanism for such thoughts, let's employ `org-journal` –journal entries are stored in files such as `journal/20190407`, where the file name is simply the date, or only one file per year as I've set it up below. Each entry is the week day, along with the date, then each child tree is an actual entry with a personal title preceded by the time the entry was made. Unlike capture and its agenda support, journal ensures entries are maintained in chronological order with calendar support.

Since org files are plain text files, an entry can be written anywhere and later ported to the journal.

The separation of concerns is to emphasise the capture stage as being quick and relatively mindless, whereas the Journaling stage as being mindful. Even though we may utilise capture to provide quick support for including journal entries, I have set my journal to be on a yearly basis –one file per year– since I want to be able to look at previous entries when making the current entry; after all, it's hard to compare and contrast easily unless there's multiple entries opened already.

As such, ideally at the end of the day, I can review what has happened, and what has not, and why this is the case, and what I intend to do about it, and what problems were encountered and how they were solved –in case the problem is encountered again in the future. **Consequently, if I encounter previously confronted situations, problems, all I have to do is reread my journal to get an idea of how to progress.** Read more about the importance of reviewing your day on a daily basis.

Moreover, by journaling with Org on a daily basis, it can be relatively easy to produce a report on what has been happening recently, at work for example. For now, there is no need to have multiple journals, for work and for personal life, as such I will utilise the tag `:work:` for non-personal matters.

Anyhow, the setup:

```
(use-package org-journal
  :bind (("C-c j" . org-journal-new-entry))
  :config
  (setq org-journal-dir "~/Dropbox/journal/")
  (setq org-journal-file-type 'yearly)
)
```

Bindings available in `org-journal-mode`, when journaling:

- `C-c C-j`: Insert a new entry into the current journal file.

    – Note keys for `org-journal-new-entry` overwrite those for `org-goto`.

- `C-c C-s`: Search the journal for a string.

    – Note keys for `org-journal-search` overwrite those for `org-schedule`.

All journal entries are registered in the Emacs Calendar. To see available journal entries do `M-x calendar`. Bindings available in the calendar-mode:

- `j`: View an entry in a new buffer.

- `i j`: add a new entry into the day's file

- `f w/m/y/f/F`: Search in all entries of the current week, month, year, all of time, of in all entries in the future.

4.5   Workflow States

Here are some of my common workflow states, –the '!' indicates a timestamp should be generated–

```
(setq org-todo-keywords
      (quote ((sequence "TODO(t)" "STARTED(s@/!)" "|" "DONE(d/!)")
              (sequence "WAITING(w@/!)" "ON_HOLD(h@/!)" "|" "CANCELLED(c@/!)")
             )
      )
)
```

The `@` brings up a pop-up to make a local note about why the state changed. **Super cool stuff!** In particular, we transition from `TODO` to `STARTED` once 15 minutes, or a reasonable amount, of work has transpired. Since all but one state are marked for logging, we could use the `lognotestate` logging facility of org-mode, which prompts for a note every time a task's state is changed.

Entering a comment about what I've done, even if it's very little, feels like I'm getting something done. It's an explicit marker of progress and motivates me to want to change my task's states more often until I see it marked `DONE`.

Here's how they are coloured,

```
(setq org-todo-keyword-faces
      (quote (("TODO" :foreground "red" :weight bold)
              ("STARTED" :foreground "blue" :weight bold)
              ("DONE" :foreground "forest green" :weight bold)
              ("WAITING" :foreground "orange" :weight bold)
              ("ON_HOLD" :foreground "magenta" :weight bold)
              ("CANCELLED" :foreground "forest green" :weight bold)))))
```

Now we press `C-c C-t` then the letter shortcut to actually make the state of an org heading.

```
(setq org-use-fast-todo-selection t)
```

We can also change through states using Shift- left, or right.

Let's draw a state diagram to show what such a workflow looks like.

PlantUML supports drawing diagrams in a tremendously simple format –it even supports Graphviz/DOT directly and many other formats. Super simple setup instructions can be found here; below are a bit more involved instructions. Read the manual here.
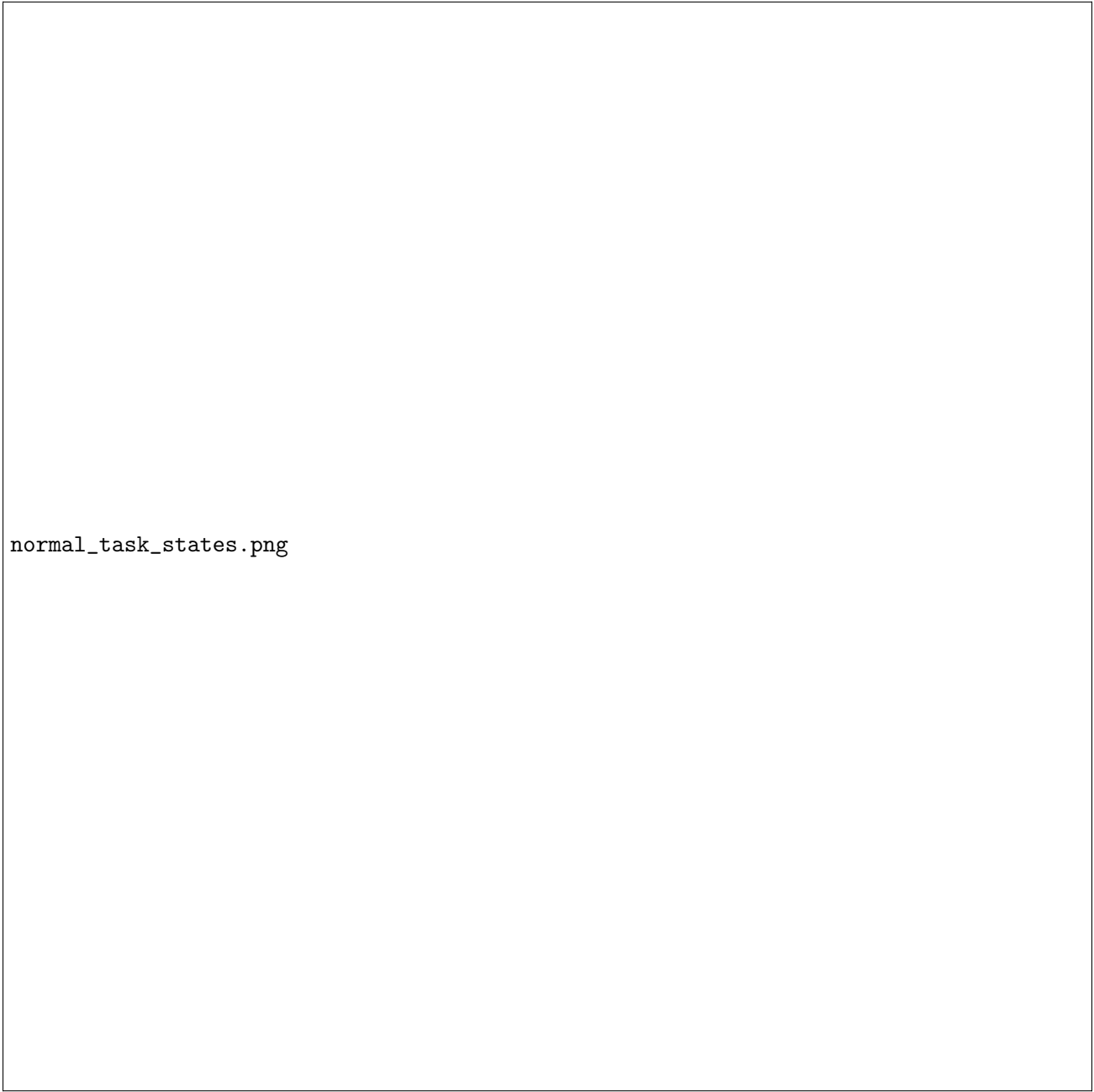
```
;; Install the tool
; (async-shell-command "brew cask install java") ;; Dependency
; (async-shell-command "brew install plantuml")

;; Tell emacs where it is.
;; E.g., (async-shell-command "find / -name plantuml.jar")
(setq org-plantuml-jar-path
      (expand-file-name "/usr/local/Cellar/plantuml/1.2019.5/libexec/plantuml.jar"))

;; Enable C-c C-c to generate diagrams from plantuml src blocks.
(add-to-list 'org-babel-load-languages '(plantuml . t) )
(require 'ob-plantuml)

; Use fundamental mode when editing plantuml blocks with C-c '
(add-to-list 'org-src-lang-modes (quote ("plantuml" . fundamental)))
```

Let's use this!

```
normal_task_states.png
```

Of note:

- Multiline comments are with `/' comment here '/`, single quote starts a one-line comment.

- Nodes don't need to be declared, and their names may contain spaces if they are enclosed in double-quotes.

- One forms an arrow between two nodes by writing a line with `x ->[label here] y` or `y <- x`; or using `-->` and `<--` for dashed lines. The label is optional.

  To enforce a particular layout, use `-X->` where `X  {up, down, right, left}`.

- To declare that a node `x` has fields `d, f` we make two new lines having `x :  f` and `x :  d`.

- One adds a note by a node `x` as follows: `note right of x:  words then newline\nthen more words`. Likewise for notes on the `left, top, bottom`.

  – Interesting sprites and many other things can be done with PlantUML. Read the docs.

This particular workflow is inspired by Bernt Hansen –while quickly searching through the PlantUML manual: The above is known as an "activity diagram" and it's covered in §4.

4.6 Show off-screen Heading at the top of the window

In case we forgot which heading we're under, let's keep the current heading stuck at the top of the window.

```
(use-package org-sticky-header
 :config
 (setq-default
  org-sticky-header-full-path 'full
  ;; Child and parent headings are seperated by a /.
  org-sticky-header-outline-path-separator " / ")
 :init (org-sticky-header-mode)
)
```

4.7 Clocking Work Time

Let's keep track of the time we spend working on tasks that we may have captured for ourselves the previous day. Such statistics provides a good idea of how long it actually takes me to accomplish a certain task in the future and it lets me know where my time has gone.

Clock in on a heading with `I`, or in the subtree with `C-c C-x C-i`.

Clock out of a heading with `O`, or in the subtree with `C-c C-x C-o`.

Clock report See clocked times with `C-c C-x C-r`.

After clocking out, the start and end times, as well as the elapsed time, are added to a drawer to the heading. We can punch in and out of tasks as many times as desired, say we took a break or switched to another task, and they will all be recorded into the drawer.

```
;; Record a note on what was acciomplished when clocking out of an item.
(setq org-log-note-clock-out t)
```

To get started, we could estimate how long a task will take and clock-in; then clock-out and see how long it actually took.

Moreover, we can overlay due dates and priorities to tasks in a non-intrusive way that is easy to edit by hand.

```
;; List of all the files where todo items can be found. Only one for now.
(setq org-agenda-files '("~/Dropbox/todo.org"))

;; How many days ahead the default agenda view should look
(setq org-agenda-ndays 7)

;; How many days early a deadline item will begin showing up in your agenda list.
(setq org-deadline-warning-days 14)

;; In the agenda view, days that have no associated tasks will still have a line showing the date.
(setq org-agenda-show-all-dates t)

(setq org-agenda-skip-deadline-if-done t)

;; Scheduled items marked as complete will not show up in your agenda view.
(setq org-agenda-skip-scheduled-if-done t)

;; The agenda view - even in the 7-days-at-a-time view - will always begin on the current day.
;; This is important, since while using org-mode as a day planner, you never want to think of
;; days gone past. That's something you do in other ways, such as when reviewing completed tasks.
(setq org-agenda-start-on-weekday nil)
```

Sometimes, at the beginning at least, I would accidentally invoke the transposed command `C-x C-c`, which saves all buffers and quits Emacs. So here's a helpful way to ensure I don't quit Emacs accidentally.

```
(setq confirm-kill-emacs 'yes-or-no-p)

;; Resume clocking task when emacs is restarted
(org-clock-persistence-insinuate)

;; Show lot of clocking history
(setq org-clock-history-length 23)

;; Resume clocking task on clock-in if the clock is open
(setq org-clock-in-resume t)

;; Sometimes I change tasks I'm clocking quickly - this removes clocked tasks with 0:00 duration
(setq org-clock-out-remove-zero-time-clocks t)

;; Clock out when moving task to a done state
(setq org-clock-out-when-done t)

;; Save the running clock and all clock history when exiting Emacs, load it on startup
(setq org-clock-persist t)

;; Do not prompt to resume an active clock
;; (setq org-clock-persist-query-resume nil)

;; Include current clocking task in clock reports
(setq org-clock-report-include-clocking-task t)
```

Finding tasks to clock in Use one of the following options, with the top-most being the first to be tried.

- From anywhere, `C-u C-c C-x C-i` yields a pop-up for recently clocked in tasks.

- Pick something off today's agenda scheduled items.

- Pick a `Started` task from the agenda view, work on this unfinished task.

- Pick something from the `TODO` tasks list in the agenda view.

- `C-c C-x C-d` also provides a quick summary of clocked time for the current org file.

Estimates versus actual time Before clocking into a task, add to the properties drawer `:Effort: 1:25` or `C-c C-x C-e`, for a task that you estimate will take an hour an twenty-five minutes, for example. Now the modeline will have will mention the time elapsed alongside the task name.

- This is also useful when you simply want to put a time limit on a task that wont be completed anytime soon, say writing a thesis or a long article, but you still want to work on it for an hour a day and be warned when you exceed such a time constraint.

  When you've gone above your estimate time, the modeline shows it to be red.

4.8   Reveal.JS – The HTML Presentation Framework

Org-mode documents can be transformed into beautiful slide decks with org-reveal with the following two simple lines.

```
(use-package ox-reveal
 :config (setq org-reveal-root "https://cdn.jsdelivr.net/reveal.js/3.0.0/"))
```

For example, execute –`C-c C-c`– the following block to see an example slide-deck ()

```
(shell-command "curl https://raw.githubusercontent.com/yjwen/org-reveal/master/Readme.org >> Trying_out
(with-temp-buffer (find-file "Trying_out_reveal.org")
 (org-reveal-export-to-html-and-browse)
)
```

Org-mode exporting –`C-c C-e`– now includes an option `R` for such reveal slide decks.

4.9 Coloured LaTeX using Minted

Execute the following for bib ref as well as minted Org-mode uses the Minted package for source code highlighting in PDF/LaTeX –which in turn requires the pygmentize system tool.

```
(setq org-latex-listings 'minted
      org-latex-packages-alist '(("" "minted"))
      org-latex-pdf-process
      '("pdflatex -shell-escape -interaction nonstopmode -output-directory %o %f"
        "biber %b"
        "pdflatex -shell-escape -interaction nonstopmode -output-directory %o %f"
        "pdflatex -shell-escape -interaction nonstopmode -output-directory %o %f")
)
```

For faster pdf generation, may consider invoking:

```
(setq org-latex-pdf-process
      '("pdflatex -interaction nonstopmode -output-directory %o %f"))
```

4.10 Executing code from `src` blocks

For example, to execute a shell command in emacs, write a `src` with a shell command, then `C-c c-c` to see the results. Emacs will generally query you to ensure you're sure about executing the (possibly dangerous) code block; let's stop that:

```
; Seamless use of babel: No confirmation upon execution.
;; Downside: Could accidentally evaluate harmful code.
(setq org-confirm-babel-evaluate nil)
```

A worked out example can be obtained as follows: `<g TAB` then `C-c C-C` to make a nice simple graph –the code for this is in the next section.

Some initial languages we want org-babel to support:

```
(org-babel-do-load-languages
  'org-babel-load-languages
  '(
    (emacs-lisp . t)
    ;; (shell        . t)
    (python . t)
    (haskell . t)
    (ruby        . t)
    (ocaml        . t)
    (C . t)   ;; Captial "C" gives access to C, C++, D
    (dot        . t)
    (latex        . t)
    (org        . t)
    (makefile        . t)
    ))
```

```elisp
;; Preserve my indentation for source code during export.
(setq org-src-preserve-indentation t)


;; The export process hangs Emacs, let's avoid this.
;; MA: For one reason or another, this crashes more than I'd like.
;; (setq org-export-in-background t)
```

More languages can be added using `add-to-list`.

4.11   Hiding Emphasise Markers & Inlining Images

```elisp
;; org-mode math is now highlighted ;-)
(setq org-highlight-latex-and-related '(latex))


;; Hide the *,=,/ markers
(setq org-hide-emphasis-markers t)


;; (setq org-pretty-entities t)
;; to have \alpha, \to and others display as utf8 http://orgmode.org/manual/Special-symbols.html
```

The following is now disabled –it makes my system slower than I'd like.

```elisp
;; Let's set inline images.
(setq org-display-inline-images t)
(setq org-redisplay-inline-images t)
(setq org-startup-with-inline-images "inlineimages")


;; Automatically convert LaTeX fragments to inline images.
(setq org-startup-with-latex-preview t)
```

4.12   Jumping without hassle

```elisp
(defun my/org-goto-line (line)
  "Go to the indicated line, unfolding the parent Org header.

  Implementation: Go to the line, then look at the 1st previous
  org header, now we can unfold it whence we do so, then we go
  back to the line we want to be at.
  "
  (interactive "nEnter line: ")
  (goto-line line)
  (org-previous-visible-heading 1)
  (org-cycle)
  (goto-line line)
)
```

4.13   Folding within a subtree

```elisp
(defun my/org-fold-current-subtree-anywhere-in-it ()
  "Hide the current heading, while being anywhere inside it."
  (interactive)
  (save-excursion
    (org-narrow-to-subtree)
    (org-shifttab)
```

```
      (widen))
  )

(add-hook 'org-mode-hook '(lambda ()
    (local-set-key (kbd "C-c C-h") 'my/org-fold-current-subtree-anywhere-in-it)))
```

4.14 Making then opening html's from org's

```
(cl-defun my/org-html-export-to-html (&optional (filename (buffer-name)))
  "Produce an HTML from the given 'filename', or otherwise current buffer,
   then open it in my default brower.
  "
 (interactive)
 (org-html-export-to-html)
 (let ((it (concat (file-name-sans-extension buffer-file-name) ".html")))
   (browse-url it)
   (message (concat it " has been opened in Chromium."))
   'success ;; otherwise we obtain a "compiler error".
 )
)
```

4.15 Making then opening pdf's from org's

```
(cl-defun my/org-latex-export-to-pdf (&optional (filename (buffer-name)))
  "Produce a PDF from the given 'filename', or otherwise current buffer,
   then open it in my default viewer.
  "
 (interactive)
 (org-latex-export-to-pdf)
 (let ((it (concat (file-name-sans-extension filename) ".pdf")))
   (eshell-command (concat "open " it  " & ")))
   (message (concat it " has been opened in your PDF viewer."))
   'success ;; otherwise we obtain a "compiler error".
)
```

4.16 Interpret the Haskell source blocks in a file

```
(defvar *current-module* "NoModuleNameSpecified"
  "The name of the module, file, that source blocks are
   currently being tangled to.

   This technique is insipired by ''Interactive Way to C'';
   see https://alhassy.github.io/InteractiveWayToC/.
  ")

(defun current-module ()
  "Returns the current module under focus."
  *current-module*)

(defun set-module (name)
   "Set the name of the module currently under focus.

    Usage: When a module is declared, i.e., a new file has begun,
    then that source blocks header should be '':tangle (set-module ''name-here'')''.
```

```
    succeeding source blocks now inherit this name and so are tangled
    to the same module file. How? By placing the following line at the top
    of your Org file: ‘‘‘#+PROPERTY: header-args :tangle (current-module))’.

    This technique structures ‘‘Interactive Way to C’’.
    "
  (setq *current-module* name)
)


(cl-defun my/org-run-haskell (&optional target (filename (buffer-name)))
  "Tangle Haskell source blocks of given ‘filename’, or otherwise current buffer,
   and load the resulting ‘target’ file into a ghci buffer.

   If no name is provided for the ‘target’ file that is generated from the
   tangeling process, it is assumed to be the buffer’s name with a ‘hs’ extension.

   Note that this only loads the blocks tangled to ‘target’.

   For example, file ‘X.org’ may have haskell blocks that tangle to files
   ‘X.hs’, ‘Y.hs’ and ‘Z.hs’. If no target name is supplied, we tangle all blocks
   but only load ‘X.hs’ into the ghci buffer. A helpful technique to load the
   last, bottom most, defined haskell module, is to have the module declaration’s
   source block be ‘:tangle (setq CODE ‘‘Y.hs’’)’, for example; then the following
   code blocks will inherit this location provided our Org file has at the top
   ‘#+PROPERTY: header-args :tangle (current-module))’.
   Finally, our ‘compile-command’ suffices to be ‘(my/org-run-haskell CODE)’.

   This technique structures ‘‘Interactive Way to C’’.
   "
  (let* ((it  (if target target (concat (file-name-sans-extension filename) ".hs")))
         (buf (concat "*GHCI* " it)))

    (-let [kill-buffer-query-functions nil] (ignore-errors (kill-buffer buf)))
    (org-babel-tangle it "haskell")
    (async-shell-command (concat "ghci " it) buf)
    (switch-to-buffer-other-window buf)
    (end-of-buffer)
  )
)


;; Set this as the ‘compile-command’ in ‘Local Variables’, for example.
```

5 TODO Expected IDE Support

```
;; Use 4 spaces in places of tabs when indenting.
(setq-default indent-tabs-mode nil)
(setq-default tab-width 4)
```

5.1 Backups

By default, Emacs saves backup files – those ending in ~ – in the current directory, thereby cluttering it up. Let's place them in ~/.emacs.d/backups, in case we need to look for a backup; moreover, let's keep old versions since there's disk space to go around –what am I going to do with 500gigs when nearly all my 'software' is textfiles

interpreted within Emacs

```
;; New location for backups.
(setq backup-directory-alist '(("." . "~/.emacs.d/backups")))

;; Never silently delete old backups.
(setq delete-old-versions -1)

;; Use version numbers for backup files.
(setq version-control t)

;; Even version controlled files get to be backed up.
(setq vc-make-backup-files t)
```

Why backups? Sometimes I may forget to submit a file, or edit, to my version control system, and it'd be nice to be able to see a local automatic backup. Whenever 'I need space,' then I simply empty the backup directory, if ever. That the backups are numbered is so sweet ˆ‿ˆ

Like package installations, my backups are not kept in any version control system, like git; only locally.

Let's use an elementary diff system for backups.

```
(use-package backup-walker
  :commands backup-walker-start)
```

In a buffer that corresponds to a file, invoke `backup-walker-start` to see a visual diff of changes *between* versions. By default, you see the changes 'backwards': Red means delete these things to get to the older version; i.e., the red '-' are newer items.

5.2 Highlighting TODO-s

```
;; NOTE that the highlighting works even in comments.
(use-package hl-todo
  :init

  ;; Adding a new keyword: TEST.
  (add-to-list 'hl-todo-keyword-faces '("TEST" . "#dc8cc3"))

  (add-hook 'text-mode-hook (lambda () (hl-todo-mode t)))
)
```

Default todos include: TODO NEXT THEM PROG OKAY DONT FAIL DONE NOTE KLUDGE HACK TEMP FIXME and any sequence of X's or ?'s of length at least 3: XXX, XXXX, XXXXX, . . . , ???, ????, ????, ...

5.3 Taking a tour of one's edits

This package allows us to move around the edit points of a buffer *without* actually undoing anything. We even obtain a brief description of what happend at each edit point. This seems useful for when I get interrupted or lose my train of thought: Just press `C-c e ,` to see what I did recently and where.

```
(use-package goto-chg
    ;; Give me a description of the change made at a particular stop.
  :init (setq glc-default-span 0)
  :bind (("C-c e ," . goto-last-change)
         ("C-c e ." . goto-last-change-reverse)))
```

Compare this with `C-x u`, or `undo-tree-visualise`, wherein undos are actually performed.

5.4   Edit as Root

From an emacs-fu blog post:

```
(defun find-file-as-root ()
  "Like 'ido-find-file, but automatically edit the file with
root-privileges (using tramp/sudo), if the file is not writable by
user."
  (interactive)
  (let ((file (ido-read-file-name "Edit as root: ")))
    (unless (file-writable-p file)
      (setq file (concat "/sudo:root@localhost:" file)))
    (find-file file)))


(bind-key "C-x F" 'find-file-as-root)
```

5.5   FIXME Default Compilation Commands

Emacs' compile command allows us to execute arbitrary Elisp when `M-x recompile` is invoked. One of my habits is to append most of my files with the following:

   Since nearly every file I work with is or can be coerced into being in org mode, I usually have a section `* footer` that contains something like the above.

   Let's remove repeated matter.

```
;; Silently save before compiling.
(setq compilation-ask-about-save nil)

;; Silently kill previous compilation process before starting a new one.
(setq compilation-always-kill t)

;; Scroll as compilation output is procuded in *Compilation* buffer; e.g., pdflatex
;; Use 'first-error to stop scrolling on the first error encountered.
(setq compilation-scroll-output t)

;; My global compile command
(setq compile-command
  '(async-shell-command (concat "open " (org-latex-export-to-pdf))))

;; Bind 'recompile' to 'C-c C-m' ''m'' for ''m''ake
(global-set-key (kbd "C-c C-m") 'recompile)

;; Also a helpful quick f-key.
(global-set-key (kbd "<f7>") 'recompile)
```

5.6   Enabling CamelCase Aware Editing Operations

Subword movement lets us treat "EmacsIsAwesome" as three words "Emacs", "Is", and "Awesome" which is desirable since such naming is common among coders. Now, for example, `M-f` moves along each subword.

```
(global-subword-mode 1)
```

5.7   Mouse Editing Support

```
;; Text selected with the mouse is automatically copied to clipboard.
(setq mouse-drag-copy-region t)
```

5.8 Dimming Unused Windows

Let's dim windows, and even the whole Emacs frame, when not in use.

```
(use-package dimmer
  :config (dimmer-mode))
```

5.9 Having a workspace manager in Emacs

I've loved using XMonad as a window tiling manager. I've enjoyed the ability to segregate my tasks according to what 'project' I'm working on; such as research, marking, Emacs play, etc. With perspective, I can do the same thing :-)

That is, I can have a million buffers, but only those that belong to a workspace will be visible when I'm switching between buffers, for example.

```
(use-package perspective)


;; Activate it.
(persp-mode)


;; In the modeline, tell me which workspace I'm in.
(persp-turn-on-modestring)
```

All commands are prefixed by `C-x x`; main commands:

`s, n/→, p/←` 'S'elect a workspace to go to or create it, or go to 'n'ext one, or go to 'p'revious one.

`c` Query a perspective to kill.

`r` Rename a perspective.

`A` Add buffer to current perspective & remove it from all others.

As always, since we've installed `which-key`, it suffices to press `C-x x` then look at the resulting menu

5.10 Jump between windows using Cmd+Arrow & between recent buffers with Meta-Tab

```
(use-package windmove
  :config
  ;; use command key on Mac
  (windmove-default-keybindings 'super)
  ;; wrap around at edges
  (setq windmove-wrap-around t))
```

The docs for the following have usage examples.

```
(use-package buffer-flip
  :bind
  (:map buffer-flip-map
   ("M-<tab>" . buffer-flip-forward)
   ("M-S-<tab>" . buffer-flip-backward)
   ("C-g" . buffer-flip-abort))
  :config
    (setq buffer-flip-skip-patterns
        '("^\\*helm\\b"))
)
;; key to begin cycling buffers.
(global-set-key (kbd "M-<tab>") 'buffer-flip)
```

5.11 Completion Frameworks

Helm provides possible completions and also shows recently executed commands when pressing `M-x`.

Extremely helpful for when switching between buffers, `C-x b`, and discovering & learning about other commands! E.g., press `M-x` to see recently executed commands and other possible commands!

Try and be grateful.

```
(use-package helm
 :diminish
 :init (helm-mode t)
 :bind
  ("C-x C-r" . helm-recentf)       ; search for recently edited

  ;; Helm provides generic functions for completions to replace
  ;; tab-completion in Emacs with no loss of functionality.
  ("M-x" . 'helm-M-x)
  ;; ("C-x b". 'helm-buffers-list) ;; Avoid seeing all those *helm* mini buffers!
  ("C-x b". 'helm-mini) ;; see buffers & recent files; more useful.
  ("C-x r b" .'helm-filtered-bookmarks)
  ("C-x C-f" . 'helm-find-files)

   ;; Show all meaningful Lisp symbols whose names match a given pattern.
   ;; Helpful for looking up commands.
   ("C-h a" . helm-apropos)

   ;; Look at what was cut recently & paste it in.
   ("M-y" . helm-show-kill-ring)
)
;; (global-set-key (kbd "M-x") 'execute-extended-command) ;; Default "M-x"

;; Yet, let's keep tab-completetion anyhow.
(define-key helm-map (kbd "TAB") #'helm-execute-persistent-action)
(define-key helm-map (kbd "<tab>") #'helm-execute-persistent-action)
;; We can list 'actions' on the currently selected item by C-z.
(define-key helm-map (kbd "C-z")  'helm-select-action)
```

When `helm-mode` is enabled, even help commands make use of it. E.g., `C-h o` runs `describe-symbol` for the symbol at point, and `C-h w` runs `where-is` to find the key binding of the symbol at point. Both show a pop-up of other possible commands.

Let's ensure `C-x b` shows us: Current buffers, recent files, and bookmarks as well as the ability to create bookmarks, which is via `C-x r b` manually. For example, I press `C-x b` then type any string and will have the option of making that a bookmark referring to the current location I'm working in, or jump to it if it's an existing bookmark, or make a buffer with that name, or find a file with that name.

```
(setq helm-mini-default-sources '(helm-source-buffers-list
                                  helm-source-recentf
                                  helm-source-bookmarks
                                  helm-source-bookmark-set
                                  helm-source-buffer-not-found))
```

Incidentally, helm even provides an interface for the top program via `helm-top`. It also serves as an interface to popular search engines and over 100 websites such as `google, stackoverflow`, and `arxive`.

```
;; (shell-command "brew install surfraw &")
;;
;; Invoke helm-surfraw
```

If we want to perform a google search, with interactive suggestions, then invoke `helm-google-suggest` –which can be acted for other serves, such as Wikipedia or Youtube by `C-z`. For more google specific options, there is the `google-this` package.

Let's switch to a powerful searching mechanism – helm-swoop. It allows us to not only search the current buffer but also the other buffers and to make live edits by pressing `C-c C-e` when a search buffer exists. Incidentally, executing `C-s` on a word, region, will search for that particular word, region; then apply changes by `C-x C-s`.

```
(use-package helm-swoop
  :bind
  (
   ("C-s"    . 'helm-swoop)          ;; search current buffer
   ("C-M-s"  . 'helm-multi-swoop-all) ;; Search all buffer
   ;; Go back to last position where 'helm-swoop' was called
   ("C-S-s" . 'helm-swoop-back-to-last-point)
  )
 :config ;; Following from helm-swoop's github page.
  ;; Give up colour for speed.
  (setq helm-swoop-speed-or-color nil)
  ;; If this value is t, split window inside the current window
  (setq helm-swoop-split-with-multiple-windows nil)

)
```

Press `M-i` after a search has executed to enable it for all buffers.

We can also limit our search to org files, or buffers of the same mode, or buffers belonging to the same project!

Note that on the Mac, I can still perform default Emacs search using *Cmd+f*.

Finally, let's enable "complete anything" mode –it ought to start in half a second and only need two characters to get going, which means word suggestions are provided and so I need only type partial words then tab to get the full word!

```
(use-package company
  :diminish
  :config
    (setq company-dabbrev-other-buffers t
          company-dabbrev-code-other-buffers t

          ;; Allow (lengthy) numbers to be eligible for completion.
          company-complete-number t

          ;; M-num to select an option according to its number.
          company-show-numbers t

          ;; Only 2 letters required for completion to activate.
          company-minimum-prefix-length 2

          ;; Do not downcase completions by default.
          company-dabbrev-downcase nil

          ;; Even if I write something with the 'wrong' case,
          ;; provide the 'correct' casing.
          company-dabbrev-ignore-case t

          ;; Immediately activate completion.
          company-idle-delay 0
```

```
      )

    (global-company-mode 1)
)
;; So fast that we don't need this.
;; (global-set-key (kbd "C-c h") 'company-complete)
```

Note that `M-/` goes through a sequence of completions. Note that besides the arrow keys, we can also use `C-` or `M-` with `n, p` to navigate the options. Note that by default company mode does not support completion for phrases containing hyphens –this can be altered, if desired.

Besides boring word completition, let's add support for emojis.

```
(use-package company-emoji)
(add-to-list 'company-backends 'company-emoji)
```

For example: .

On a new line, write : then any letter to have a tool-tip appear. All emoji names are lowercase.

The libraries `emojify, emojify-logos` provides cool items like :haskell: :emacs: :org: :ruby: :python:. Unfortunately they do not easily export to html with org-mode, so I'm not using them.

Let documentation pop-up when we pause on a completion. This is very useful when editing in a particular coding language, say via `C-c '` for org-src blocks.

```
(use-package company-quickhelp
 :config
   (setq company-quickhelp-delay 0.1)
   (company-quickhelp-mode)
)
```

5.12   Snippets – Template Expansion

It is common that there is a sequence of text that we tend to repeat often, possibly with a name or some other parameter altered. Such a snippet could be written once then provided by a simple Lisp insert command with the parameters being queried. Luckily, others have written such pleasant utilities.

Yasnippet is a pleasant utility for template expansion with the alluring feature to allow arbitrary Lisp code to be executed during expansion. The declaration of templates is verbose, requiring a particular file hierarchy, as such I utilise Yankpad which allows me to employ an Org-mode approach: Each template corresponds to an org heading of the form `Key:Words:For:Expansion:Here:  name of snippet here` and the template body is then the body of the org heading. Any of `Key, Words, For, Expansion, Here` will rewrite into the body of the org tree. This is much more terse, and I even don't bother with that; instead preferring to tangle my templates using yankpad as a mere interface.

```
;; Yet another snippet extension program
(use-package yasnippet
  :diminish yas-minor-mode
  :config
    (yas-global-mode 1)
    ;; respect the spacing in my snippet declarations
    (setq yas-indent-line 'fixed)
)


;; Nice "interface" to said program
(use-package yankpad
  ;; :if company-mode ;; load & initialise only if company-mode is defined
  :demand t
  :init
```

```
    ;; Location of templates
    (setq yankpad-file "~/.emacs.d/yankpad.org")
    (setq yankpad-category "Category: Default")
  :config
    ;; If you want to complete snippets using company-mode
    ;; (add-to-list 'company-backends #'company-yankpad)
    ;; If you want to expand snippets with hippie-expand
    (add-to-list 'hippie-expand-try-functions-list #'yankpad-expand)
    ;; Load the snippet templates -- useful after yankpad is altered
    ;; (add-hook 'after-init-hook 'yankpad-reload)
)


;; Elementary textual completion backend.
(setq company-backends
    (add-to-list 'company-backends 'company-dabbrev))
;;
;; Add yasnippet support for all company backends
;; https://emacs.stackexchange.com/a/10520/10352
;;
(defvar company-mode/enable-yas t
  "There can only be one main completition backend, so let's
   enable yasnippet/yankpad as a secondary for all completion backends.")

(defun company-mode/backend-with-yas (backend)
  (if (or (not company-mode/enable-yas)
          (and (listp backend) (member 'company-yankpad backend)))
      backend
    (append (if (consp backend) backend (list backend))
            '(:with company-yankpad))))

(setq company-backends (mapcar #'company-mode/backend-with-yas company-backends))
```

With these settings, along with the `company` backend, I may type a keyword then "tab" it into expansion.

Yankpad requires we have an org file that contains our templates, so we tangle such a file `~/.emacs.d/yankpad.org`, and have all of our templates be globally accessible.

Here's an example of a common template I perform by hand –no more! I have the expected habit of copying a URL from someplace then forming a link to it by writing `[[URL] [description]]`, since the URL & syntax are already known, let's expand those and place the cursor at the only unknown –the description.

What's going on here?

1. This template is expanded with the keyword `my-org-insert-link`, then "tab".

2. The cursour lands at position `$1`, which has default text being the result of evaluating (`clipboard-yank`).

   We may evaluate Lisp code anywhere by enclosing it in backticks.

3. If we're satisfied with the current field, we simply tab to the next field. Otherwise, we simply write text –which overwrites the default text.

4. After enough tabbing we complete the template and the cursour lands at position `$0`.

Having default or mirrored text for `$2` would not allow me to see the URL field, lest I wish to change it or at least confirm it's what I want. Hence, the `$2` field has no default.

Let's overwrite the usual way to insert such links, via `C-c C-l`.

```
(cl-defun org-insert-link ()
  "Makes an org link by inserting the URL copied to clipboard
  and prompting for the link description only.

  Type over the shown link to change it, or tab to move to the description
  field.

  This overrides Org-mode's built-in 'org-insert-link' utility.
  "
  (interactive)
  (insert "my-org-insert-link")
  (yankpad-expand)
)
```

The Yasnippe manual is an accessible read, as is the Yankpad manual, and showcases many other utilities; such as having certain snippets being activated enabled only in particular modes or on demand. Of note is that field `$n` can be accessed in code with the invocation (`yas-field-value n`).

The rest of this section is other templates, not much for now, concluding with actually loading this snippet mechanism globally.

5.12.1 Org-mode Templates –A reason I "generate" templates ;)

This produces a pop-up list of org-mode block types, if `src` is selected, then a list of my commonly used languages pops-up. Alternatively, ignore the pop-up menu and write any block or language name. In this case, `yas-text` is equivalent to (`yas-field-value 1`); it generally refers to the value of the field being mirrored with `${n:  yas-text}`.

However, going through pop-ups takes precious time. Let's introduce a template for my most utilised kind of language blocks.

However, doing this for each language I want is a waste of time and textual space. Why? The purpose of templates is to reduce repetition, yet the above block would be repeated with only 3 parts 'unknown': The expansion keyword, the description, and the org-mode source block name. Whence,

```
(defun make-lang-template (key lang)
  "We make an org-mode source block snippet template.

  'key' is the expansion word key for the language 'lang';
  the description for the snippet is also 'lang'.
  "
  (s-join "\n" '(
    ,(concat "** " key ": " lang)
    ,(concat "#+begin_src " lang)
    "$0"
    "#+end_src"
    "\n"
  ))
)
```

The template *text* is then generated by the following simple loop –whose source block is named `my-org-lang-templates`.

```
(let (result) ;; ensure result is initially empty
  (dolist (x '( ("s_el" . "emacs-lisp")
                ("s_org" . "org")
                ("s_hs" . "haskell")
                ("s_ag" . "agda2")
                ("s_c" . "c")
```

46

```
            ("s_lx" . "latex")
          ) result)
    (setq result (concat result "\n" (make-lang-template (car x) (cdr x))))
  )
)
```

The *resulting text* of this block, generated below, is tangled to our yankpad by utilising a noweb source block invocation:

Now `s-`, due to company mode, brings up a list of languages that I can then simply scroll down through, then "enter" upon to expand. Moreover, the prefix `s-` means that the key is mostly irrelevant, since I needn't remember it because company-mode immediately lists possible completions. Super neat stuff :-)

Ain't this reminiscent of meta-programming ;-)

Incidentally, I originally wrote an Elisp script to temporarily open the yankpad, then append the desired text. However, such an approach is brittle: I have to manually execute said script.

In contrast, using noweb invocations to tangle the results is more flexible: Any time the tangling is performed, the yankpad is kept up to date –no personal intervention from myself.

Observe that noweb could be utilised in a similar fashion to share key-value pairs across different source files.

5.12.2 Elisp Templates

5.12.3 Equational Templates

5.12.4 Misc Templates

Where my local use contains `#+MACRO: remark @@latex:  \fbox{\textbf{Comment:  $1 }}@@`.

5.12.5 Re-Enabling Templates

After our yankpad templates are generated, we need to load it.

```
;; After init hook; see above near use-package install.
(yankpad-reload)
```

6 Helpful Utilities & Shortcuts

Here is a collection of Emacs-lisp functions that I have come to use in other files.

Disclaimer: I wrote much of the following *before* I learned any lisp; everything below is probably terrible.

Let's save a few precious seconds,

```
;; change all prompts to y or n
(fset 'yes-or-no-p 'y-or-n-p)


;; Enable 'possibly confusing commands'
(put 'downcase-region 'disabled nil)
(put 'upcase-region 'disabled nil)
(put 'narrow-to-region 'disabled nil)
(put 'narrow-to-page 'disabled nil)
```

6.1 Bind `recompile` to `C-c C-m` – "m" for "m"ake

```
(defvar my-keys-minor-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "C-c C-m") 'recompile)
    map)
  "my-keys-minor-mode keymap.")
```

47

```
(define-minor-mode my-keys-minor-mode
  "A minor mode so that my key settings override annoying major modes."
  :init-value t
  :lighter " my-keys")

(my-keys-minor-mode)

(diminish 'my-keys-minor-mode) ;; Don't show it in the modeline.
```

6.2   Reload buffer with `f5`

I do this so often it's not even funny.

```
(global-set-key [f5] '(lambda () (interactive) (revert-buffer nil t nil)))
```

In Mac OS, one uses `Cmd-r` to reload a page and Emacs binds buffer reversion to `Cmd-u` –in Emacs, Mac's `Cmd` is referred to as the 'super key' and denoted `s`.

Moreover, since I use Org-mode to generate code blocks and occasionally inspect them, it would be nice if they automatically reverted when they were regenerated –Emacs should also prompt me if I make any changes!

```
;; Auto update buffers that change on disk.
;; Will be prompted if there are changes that could be lost.
(global-auto-revert-mode 1)
```

6.3   Kill to start of line

Dual to `C-k`,

```
;; M-k kills to the left
(global-set-key "\M-k" '(lambda () (interactive) (kill-line 0)) )
```

6.4   `file-as-list` and `file-as-string`

Disclaimer: I wrote the following *before* I learned any lisp; everything below is probably terrible.

```
(defun file-as-list (filename)
  "Return the contents of FILENAME as a list of lines"
  (with-temp-buffer
    (insert-file-contents filename)
    (split-string (buffer-string))))

(defun file-as-string (filename)
  "Return the contents of FILENAME as a list of lines"
  (with-temp-buffer
    (insert-file-contents filename)
    (buffer-string)))
```

6.5   `kill-other-buffers`

```
(defun kill-other-buffers ()
  "Kill all other buffers."
  (interactive)
  (mapc 'kill-buffer (delq (current-buffer) (buffer-list))))
```

6.6 Switching from 2 horizontal windows to 2 vertical windows

I often find myself switching from a horizontal view of two windows in Emacs to a vertical view. This requires a variation of `C-x 1 RET C - x 3 RET C-x o X-x b RET`. Instead I now only need to type `C-|` to make this switch.

```elisp
(defun ensure-two-vertical-windows ()
  "I used this method often when programming in Coq."
 (interactive)
 (other-window 1)                       ;; C-x 0
 (let ((otherBuffer (buffer-name)))
   (delete-window)                      ;; C-x 0
   (split-window-right)                     ;; C-x 3
   (other-window 1)                     ;; C-x 0
   (switch-to-buffer otherBuffer)        ;; C-x b RET
 )
 (other-window 1)
)
(global-set-key (kbd "C-|") 'ensure-two-vertical-windows)
```

6.7 re-replace-in-file

Disclaimer: I wrote the following *before* I learned any lisp; everything below is probably terrible.

```elisp
(defun re-replace-in-file (file regex whatDo)
   "Find and replace a regular expression in-place in a file.

   Terrible function ... before I took the time to learn any Elisp!
   "

   (find-file file)
   (goto-char 0)
   (let ((altered (replace-regexp-in-string regex whatDo (buffer-string))))
     (erase-buffer)
     (insert altered)
     (save-buffer)
     (kill-buffer)
   )
)
```

Example usage:

```elisp
;; Within mysite.html we rewrite: <h1.*h1>      <h1.*h1>\n NICE
;; I.e., we add a line break after the first heading and a new word, ''NICE''.
(re-replace-in-file "mysite.html"
                    "<h1.*h1>"
                    (lambda (x) (concat x "\n NICE")))
```

6.7.1 mapsto: Simple rewriting for current buffer

```elisp
(defun mapsto (this that)
  "In the current buffer make the regular expression rewrite: this  that."
  (let* ((current-location (point))
       ;; Do not alter the case of the <replacement text>.
       (altered (replace-regexp-in-string this (lambda (x) that) (buffer-string) 'no-fixed-case))
       )
      (erase-buffer)
```

```
    (insert altered)
    (save-buffer)
    (goto-char current-location)
  )
)
```

6.8   Obtaining Values of `#+KEYWORD` Annotations

Org-mode settings are, for the most part, in the form `#+KEYWORD: VALUE`. Of notable interest are the `TITLE` and `NAME` keywords. We use the following `org-keywords` function to obtain the values of arbitrary `#+THIS : THAT` pairs, which may not necessarily be supported by native Org-mode –we do so for the case, for example, of the `CATEGORIES` and `IMAGE` tags associated with an article.

```
;; Src: http://kitchingroup.cheme.cmu.edu/blog/2013/05/05/Getting-keyword-options-in-org-files/
(defun org-keywords ()
  "Parse the buffer and return a cons list of (property . value) from lines like: #+PROPERTY: value"
  (org-element-map (org-element-parse-buffer 'element) 'keyword
                   (lambda (keyword) (cons (org-element-property :key keyword)
                                           (org-element-property :value keyword)))))


(defun org-keyword (KEYWORD)
  "Get the value of a KEYWORD in the form of #+KEYWORD: value"
  (cdr (assoc KEYWORD (org-keywords))))
```

Note that capitalisation in a "#+KeyWord" is irrelevant.
See here on how to see the abstract syntax tree of an org file and how to manipulate it.

6.9   Quickly pop-up a terminal, run a command, close it

```
(cl-defun toggle-terminal (&optional (name "*eshell-pop-up*"))
  "Pop up a terminal, do some work, then close it using the same command.

  The toggle behaviour is tied into the existence of the pop-up buffer.
  If the buffer exists, kill it; else create it.
  "
  (interactive)
  (cond
    ;; when the terminal buffer is alive, kill it.
    ((get-buffer name)  (kill-buffer name)
                        (ignore-errors (delete-window)))
    ;; otherwise, set value to refer to a new eshell buffer.
    (t                  (split-window-right)
                        (other-window 1)
                        (eshell)
                        (rename-buffer name))
  )
)


(global-set-key "\C-t" 'toggle-terminal)
```

6.10   `C-x k` kills current buffer

By default `C-x k` prompts to select which buffer should be selected. I almost always want to kill the current buffer, so let's not waste time making such a tedious decision.

```elisp
;; Kill current buffer; prompt only if
;; there are unsaved changes.
(global-set-key (kbd "C-x k")
  '(lambda () (interactive) (kill-buffer (current-buffer)))))
```

6.11 Publishing articles to my personal blog

I try to blog occasionally, so here's a helpful function to quickly publish the current article to my blog.

```elisp
(define-key global-map "\C-cb" 'my/publish-to-blog)


(cl-defun my/publish-to-blog (&optional (draft nil) (local nil))
  "
  Using 'AlBasmala' setup to publish current article to my blog.
  Details of AlBasmala can be found here:
  https://alhassy.github.io/AlBasmala/

  Locally: ~/alhassy.github.io/content/AlBasmala.org

  A 'draft' will be produced in about ~7 seconds, but does not re-produce
  a PDF and the article has a draft marker near the top. Otherwise,
  it will generally take ~30 seconds due to PDF production, which is normal.
  The default is not a draft and it takes ~20 seconds for the live
  github.io page to update.

  The 'local' optiona indicates whether the resulting article should be
  viewed using the local server or the live webpage. Live page is default.

  When 'draft' and 'local' are both set, the resulting page may momentarily
  show a page-not-found error, simply refresh.
  "

  (load-file "~/alhassy.github.io/content/AlBasmala.el")

  ;; --MOVE ME TO ALBASMALA--
  ;; Sometimes the file I'm working with is not a .org file, so:
  (setq file.org (buffer-name))

  (preview-article :draft draft)
  (unless draft (publish))
  (let ((server (if local "http://localhost:4000/" "https://alhassy.github.io/")))
    (async-shell-command (concat "open " server NAME "/") "*blog-post-in-browser*"))
)
```

6.12 Excellent PDF Viewer

Let's install the pdf-tools library for viewing PDFs in Emacs.

```elisp
;; First: (async-shell-command "brew install --HEAD dunn/homebrew-emacs/pdf-tools")

;; Then:
(use-package pdf-tools
  :ensure t
  :config
  (custom-set-variables
```

```
  '(pdf-tools-handle-upgrades nil))
(setq pdf-info-epdfinfo-program "/usr/local/bin/epdfinfo"))

;; Finally:
(pdf-tools-install)

;; Now PDFs opened in Emacs are in pdfview-mode.
```

Besides the expected PDF viewing utilities, such as search, annotation, and continuous scrolling; with a simple mouse right-click, we can even select a 'midnight' rendering mode which may be easier on the eyes. For more, see the brief pdf-tools-tourdeforce demo.