ldlework's init.el

setup

The following sections take care of bootstrapping `straight.el` and defining some utility elisp that is used throughout the rest of the configuration.

straight.el

straight.el is an alternative to package.el with many advantages including the ability to integrate with use-package and installing packages from git or github.

bootstrap

```
(let ((bootstrap-file (concat user-emacs-directory "straight/repos/straight.el/bootstrap.el"
      (bootstrap-version 3))
  (unless (file-exists-p bootstrap-file)
    (with-current-buffer
      (url-retrieve-synchronously
       "https://raw.githubusercontent.com/raxod502/straight.el/develop/install.el"
       'silent 'inhibit-cookies)
      (goto-char (point-max))
      (eval-print-last-sexp)))
  (load bootstrap-file nil 'nomessage))
```

use-package integration

```
(setq straight-use-package-by-default t)
(straight-use-package 'use-package)
(use-package git) ;; ensure we can install from git sources
```

dependencies

The following package dependencies are used throughout the rest of the configuration. They provide contemporary APIs for working with various elisp data structures.

```
(use-package dash :config (require 'dash))    ;; lists
(use-package ht :config (require 'ht))        ;; hash-tables
(use-package s :config (require 's))          ;; strings
(use-package a :config (require 'a))          ;; association lists
```

boilerplate

The following functions are required by some of the snippets in Emacs-nougat.

```
;; lets user override default bindings
(defun global-kbd-or (&rest args)
  (let* ((pairs (-partition 2 args)))
    (cl-loop for pair in pairs
           for target = (car pair)
           for default = (cadr pair)
           for target-name = (symbol-name target)
           for override-name = (format "kbd-%s" target-name)
           for override-symbol = (make-symbol override-name)
           if (boundp override-symbol)
           do (global-set-key (kbd (eval override-symbol)) target)
           else
           do (global-set-key (kbd default) target))))

(defun define-kbd-or (&rest args)
  (let* ((triplets (-partition 3 args)))
    (cl-loop for triplet in triplets
           for mode = (nth 0 triplet)
           for target = (nth 1 triplet)
           for default = (nth 2 triplet)
           for target-name = (symbol-name target)
           for override-name = (format "kbd-%s" target-name)
           for override-symbol = (make-symbol override-name)
           if (boundp override-symbol)
           do (define-key mode (kbd (eval override-symbol)) target)
           else
           do (define-key mode (kbd default) target))))

(defun kbd-or (target default)
  (if (boundp target)
      (kbd (eval target))
    (kbd default)))

(defun var-or (target default)
  (if (boundp target)
      (eval target)
    default))
```

personal helpers

Some helper function's I've defined to help generate paths relevant to my system and data.

my/org-path-name

```
(defvar my/org-path-name
  (expand-file-name "~/org/")
  "Root path-name for org-mode files")
```

my/org-file-name

```
(defun my/org-file-name (file-name)
  "Create file-name relative to my/org-path-name"
  (concat my/org-path-name file-name))
```

my/notes-file-name

```
(defvar my/notes-file-name
  (my/org-file-name "notes.org")
  "Main notes file-name")
```

my/template-directory

```
(setq-default my/template-directory (concat my/org-path-name "templates/"))
```

my/template-file-name

```
(defun my/template-file-name (file-name)
  "Create file-name relative to my/template-directory"
  (concat my/template-directory file-name))
```

my/project-root

```
(setq-default my/project-root (expand-file-name "~/src/"))
```

my/project-directory

```
(defun my/project-directory (name)
  (concat my/project-root name))
```

setup external browser

```
(setq browse-url-browser-function 'browse-url-chrome)
(setq browse-url-chrome-program "google-chrome-unstable")
```

nixos

NixOS ZSH module drops some PATH modification stuff in `~/.config/zsh/.zshrc` which causes the following message on startup:

```
You appear to be setting environment variables ("PATH") in your .bashrc or .zshrc:
those files are only read by interactive shells, so you should instead set
environment variables in startup files like .profile, .bash_profile or .zshenv.
Refer to your shell's man page for more info.

Customize 'exec-path-from-shell-arguments' to remove "-i" when done, or disable
'exec-path-from-shell-check-startup-files' to disable this message.
```

The following line prevents the warning above:

```
(setq exec-path-from-shell-check-startup-files nil)
```

badger-theme

My favorite theme, available here.

```
(use-package badger-theme
  :config (load-theme 'badger t))
```

default font

```
(set-face-attribute 'default nil :family "Source Code Pro" :height 145 :weight 'light)
```

blend in the fringes

```
(set-face-attribute 'fringe nil :background nil)
```

helper for getting colors

```
(defun badger-color (name)
  (let ((name (format "badger-%s" name)))
    (cdr (assoc name badger-colors-alist))))

;; (badger-color "bg")
```

emacs

The following sections customize core Emacs settings.

autosaves

Auto-save will periodically save files to backup while you editing. This is great if something goes catastrophically wrong to Emacs!

autosave every buffer that visits a file

```
(setq auto-save-default t)
```

save every 20 secs or 20 keystrokes

```
(setq auto-save-timeout 20
      auto-save-interval 20)
```

store autosaves in a single place

```
(defvar emacs-autosave-directory
  (concat user-emacs-directory "autosaves/"))

(unless (file-exists-p emacs-autosave-directory)
    (make-directory emacs-autosave-directory))

(setq auto-save-file-name-transforms
      `((".*" ,emacs-autosave-directory t)))
```

backups

Backups are created everytime a buffer is saved. This is really useful for recovering work that takes place between version-control commits or on unversioned files.

store backups with the autosaves

```
(setq backup-directory-alist
      `((".*" . ,emacs-autosave-directory)))
```

keep 10 backups

```
(setq kept-new-versions 10
      kept-old-verisons 0)
```

delete old backups

```
(setq delete-old-versions t)
```

copy files to avoid various problems

```
(setq backup-by-copying t)
```

backup files even if version controlled

```
(setq vc-make-backup-files t)
```

backup every save

```
(use-package backup-each-save
  :config (add-hook 'after-save-hook 'backup-each-save))
```

cursor

box style

```
(setq-default cursor-type 'box)
```

blinking

```
(blink-cursor-mode 1)
```

disable

Turn off various UI features to achieve a minimal, distraction free experience. Additionally, all configuration should live inside version-controlled files so the Emacs customizations file is also disabled.

menubar

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

toolbar

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

scrollbar

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

startup message

```
(setq inhibit-startup-message t
      initial-scratch-message nil)
```

customizations file

```
(setq custom-file (make-temp-file ""))
```

editing

use spaces

```
(setq-default indent-tabs-mode nil)
```

visual fill-column

```
(use-package visual-fill-column
  :config (global-visual-fill-column-mode))
```

fill at 85

```
(setq-default fill-column 85)
```

autofill text-mode

```
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

ssh for tramp

Default method for transferring files with Tramp.

```
(setq tramp-default-method "ssh")
```

key-bindings

meta n & p

```
(global-set-key (kbd "M-p") 'backward-paragraph)
(global-set-key (kbd "M-n") 'forward-paragraph)
```

minor-modes

whitespace-mode

```
(use-package whitespace
  :diminish global-whitespace-mode
  :init
  (setq whitespace-style
      '(face tabs newline trailing tab-mark space-before-tab space-after-tab))
  (global-whitespace-mode 1))
```

prettify-symbols-mode

Various symbols will be replaced with nice looking unicode glyphs.

```
(global-prettify-symbols-mode 1)
```

electric-pair-mode

Matching closed brackets are inserted for any typed open bracket.

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

rainbow-delimeters-mode

```
(require 'color)
(defun gen-col-list (length s v &optional hval)
  (cl-flet ( (random-float () (/ (random 10000000000) 10000000000.0))
        (mod-float (f) (- f (ffloor f))) )
    (unless hval
      (setq hval (random-float)))
    (let ((golden-ratio-conjugate (/ (- (sqrt 5) 1) 2))
        (h hval)
        (current length)
        (ret-list '()))
      (while (> current 0)
      (setq ret-list
            (append ret-list
                    (list (apply 'color-rgb-to-hex (color-hsl-to-rgb h s v)))))
      (setq h (mod-float (+ h golden-ratio-conjugate)))
      (setq current (- current 1)))
      ret-list)))

(defun set-random-rainbow-colors (s l &optional h)
  ;; Output into message buffer in case you get a scheme you REALLY like.
  ;; (message "set-random-rainbow-colors %s" (list s l h))
  (interactive)
  (rainbow-delimiters-mode t)

  ;; Show mismatched braces in bright red.
  (set-face-background 'rainbow-delimiters-unmatched-face "red")

  ;; Rainbow delimiters based on golden ratio
  (let ( (colors (gen-col-list 9 s l h))
       (i 1) )
    (let ( (length (length colors)) )
      ;;(message (concat "i " (number-to-string i) " length " (number-to-string length)))
      (while (<= i length)
      (let ( (rainbow-var-name (concat "rainbow-delimiters-depth-" (number-to-string i) "-fa
             (col (nth i colors)) )
        ;; (message (concat rainbow-var-name " => " col))
        (set-face-foreground (intern rainbow-var-name) col))
      (setq i (+ i 1))))))

(use-package rainbow-delimiters :commands rainbow-delimiters-mode :hook ...
  :init
  (setq rainbow-delimiters-max-face-count 16)
  (set-random-rainbow-colors 0.6 0.7 0.5)
  (add-hook 'prog-mode-hook 'rainbow-delimiters-mode))
```

show-paren-mode

```
(show-paren-mode 1)
(setq show-paren-delay 0)
(require 'paren)
(set-face-background 'show-paren-match nil)
(set-face-background 'show-paren-mismatch nil)
(set-face-foreground 'show-paren-match "#ff0")
(set-face-foreground 'show-paren-mismatch "#f00")
(set-face-attribute 'show-paren-match nil :weight 'extra-bold)
```

which-key-mode

```
(use-package which-key
  :diminish which-key-mode
  :config
  ;; sort single chars alphabetically P p Q q
  (setq which-key-sort-order 'which-key-key-order-alpha)
  (setq which-key-idle-delay 0.8)
  (which-key-mode))
```

company-mode

```
(use-package company
  :config (add-hook 'after-init-hook 'global-company-mode))
```

ispell-minor-mode

```
(setq ispell-program-name (expand-file-name "~/.nix-profile/bin/aspell"))
```

exec-path from shell

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

eyeliner for modeline

```
(use-package eyeliner
  ;; :straight (eyeliner :type git :host github :repo "dustinlacewell/eyeliner")
  :straight (eyeliner :local-repo "~/src/eyeliner")

  :init
  (setq eyeliner/warm-color (badger-color "salmon"))
  (setq eyeliner/cool-color (badger-color "blue"))
  (setq eyeliner/plain-color "#FFFFFF")

  :config
  (require 'eyeliner)
  (spaceline-helm-mode 1)
  (custom-set-faces
   `(powerline-active2
     ((t (:background ,(badger-color "bg")))))
   `(powerline-inactive2
     ((t (:background ,(badger-color "bg"))))))
  (eyeliner/install))
```

org-mode

Org-mode is the absolute pinnacle in personal organization and planning systems. Including outlining, task tracking with states, priorities, scheduling and deadlines. The features go on-and-on.

straight.el fixes

There are some issues with straight.el and org. These the following boilerplate fixes all that until that is resolved.

fix-org-git-version

```
(defun fix-org-git-version ()
  "The Git version of org-mode.
  Inserted by installing org-mode or when a release is made."
  (require 'git)
  (let ((git-repo (expand-file-name
                   "straight/repos/org/" user-emacs-directory)))
    (string-trim
     (git-run "describe"
              "--match=release\*"
              "--abbrev=6"
              "HEAD"))))
```

fix-org-release

```
(defun fix-org-release ()
  "The release version of org-mode.
  Inserted by installing org-mode or when a release is made."
  (require 'git)
  (let ((git-repo (expand-file-name
                   "straight/repos/org/" user-emacs-directory)))
    (string-trim
     (string-remove-prefix
      "release_"
      (git-run "describe"
               "--match=release\*"
               "--abbrev=0"
               "HEAD")))))
```

installation

```
(use-package org
  :mode ("\\.org\\'" . org-mode)
  :config
  ;; This forces straight to load the package immediately in an attempt to avoid the
  ;; Org that ships with Emacs.
  (require 'org)

  ;; these depend on the 'straight.el fixes' above
  (defalias #'org-git-version #'fix-org-git-version)
  (defalias #'org-release #'fix-org-release)

  ;; Enable org capture
  (require 'org-capture)

  ;; Enable templates like <s
  (require 'org-tempo)

  (define-kbd-or
    org-mode-map 'outline-next-visible-heading "M-n"
    org-mode-map 'outline-previous-visible-heading "M-p"
    org-mode-map 'org-insert-heading "M-RET"))
```

look

The following sections change how Org-mode documents look.

indent by header level

Hide the heading asterisks. Instead indent headings based on depth.

```
(with-eval-after-load 'org
  (add-hook 'org-mode-hook #'org-indent-mode))
```

pretty heading bullets

Use nice unicode bullets instead of the default asterisks.

```
(use-package org-bullets
  :after (org)
  :config
  (add-hook 'org-mode-hook 'org-bullets-mode))
```

pretty todo states

Instead of the default `TODO` and `DONE` states, use some interesting unicode characters. Use `C-c C-t` to cycle through states.

```
(with-eval-after-load 'org
  (setq org-todo-keywords '((sequence "…" "⏩" "⏸" "✔"))))
```

pretty priority cookies

Instead of the default `[#A]` and `[#C]` priority cookies, use little unicode arrows to indicate high and low priority. `[#B]`, which is the same as no priority, is shown as normal.

```
(with-eval-after-load 'org
  (defun nougat/org-pretty-compose-p (start end match)
    (if (or (string= match "[#A]") (string= match "[#C]"))
      ;; prettify asterisks in headings
      (org-match-line org-outline-regexp-bol)
      ;; else rely on the default function
      (funcall #'prettify-symbols-default-compose-p start end match)))

  (global-prettify-symbols-mode)

  (add-hook
   'org-mode-hook
   (lambda ()
     (setq-local prettify-symbols-compose-predicate #'nougat/org-pretty-compose-p)
     (setq-local prettify-symbols-alist
                 '(("[#A]" . ?↟)
                   ("[#C]" . ?↡))))))
```

pretty heading ellipsis

```
(with-eval-after-load 'org
  (setq org-ellipsis " ▿"))
```

theme customizations

```
(use-package org-beautify-theme
  :after (org)
  :config
  (setq org-fontify-whole-heading-line t)
  (setq org-fontify-quote-and-verse-blocks t)
  (setq org-hide-emphasis-markers t)
  (let* ((background-color (face-background 'default nil 'default))
       (padding nil))
    (custom-theme-set-faces
     'org-beautify
     `(org-document-title
       ((t (:inherit org-level-1
                     :height 2.0
                     :underline nil
                     :box ,padding))))
     `(org-level-1
       ((t (:height 1.5 :
                    box ,padding))))
     `(org-level-2
       ((t
       (:height 1.25
                :box ,padding))))
     `(org-level-3
       ((t (:box ,padding))))
     `(org-ellipsis
       ((t (:inherit org-level-faces))))
     `(org-list-dt
       ((t
       (:inherit default
                 :height 2.0))))
     `(org-meta-line
       ((t (:slant italic
                 :height 0.9
                 :foreground "#777777"))))
     `(org-agenda-structure
       ((t (:inherit default
                     :height 2.0
                     :underline nil))))
     `(org-document-info-keyword
       ((t (:inherit default
                     :height 0.8
                     :foreground "#AA7777"))))
     `(org-checkbox
       ((t (:box (:color "#93a1a1"
                         :style "released-button")))))
     `(org-block
       ((t (:background "#373737"
                        :box nil
                        :height 0.8
                        :family "Menlo"))))
     `(org-block-begin-line
       ((t (:height 0.8
```

```
                :foreground "#777777"
                :background "#222222"))))
    `(org-block-end-line
      ((t (:inherit org-block-begin-line))))
    `(org-quote
      ((t (:slant italic
                  :height 1.1)))))))
```

feel

The following sections change how it feels to use Org-mode.

don't fold blocks on open

```
(with-eval-after-load 'org
  (setq org-hide-block-startup nil))
```

auto-fill paragraphs

```
(with-eval-after-load 'org
  (add-hook 'org-mode-hook 'turn-on-auto-fill))
```

respect content on heading insert

If you try to insert a heading in the middle of an entry, don't split it in half, but instead insert the new heading after the end of the current entry.

```
(with-eval-after-load 'org
  (setq org-insert-heading-respect-content nil))
```

ensure one-line between headers

When you save, this section will ensure that there is a one-line space between each heading. This helps with the background color of code-blocks not showing up on folded headings.

```
(with-eval-after-load 'org
  (defun org-mode--ensure-one-blank-line ()
    (save-excursion
      (goto-char (point-min))
      (while (re-search-forward "#\\+[a-z_]+\\s-\\*" nil t)
      (replace-match "#+end_src

*")
      (call-interactively 'org-previous-visible-heading)
      (call-interactively 'org-cycle)
      (call-interactively 'org-cycle))
      (org-save-outline-visibility t
      (org-mode))))

  (add-hook
   'org-mode-hook
   (lambda () (add-hook
               'before-save-hook
               'org-mode--ensure-one-blank-line
              nil 'make-it-local))))
```

org-babel

add languages to babel

```
(with-eval-after-load 'org
  (org-babel-do-load-languages
   'org-babel-load-languages
   '((shell . t)
     (emacs-lisp . t))))
```

set default header args

```
(with-eval-after-load 'org
  (setq org-babel-default-header-args
      '((:session . "none")
        (:results . "silent")
        (:exports . "code")
        (:cache . "no")
        (:noweb . "no")
        (:hlines . "no")
        (:tangle . "no"))))
```

disable code evaluation prompts

```
(with-eval-after-load 'org
  (setq org-confirm-babel-evaluate nil)
  (setq org-confirm-shell-link-function nil)
  (setq org-confirm-elisp-link-function nil))
```

org-capture

set default notes file

```
(with-eval-after-load 'org
  (setq org-default-notes-file
      (expand-file-name (var-or 'nougat-capture-target "~/org/notes.org"))))
```

bind a key for capture

```
(with-eval-after-load 'org
  (global-set-key (kbd "C-c c") 'org-capture))
```

automatically visit new capture

```
(with-eval-after-load 'org
  (add-to-list 'org-capture-after-finalize-hook 'org-capture-goto-last-stored))
```

projectile

Projectile offers a number of features related to project interaction. It can track the root directories and sibling files of files you edit automatically. Combined with Helm, you can very quickly navigate related files.

Projectile's default prefix is `C-c p`

```
(use-package projectile
  :config
  (setq projectile-enable-caching t)
  (projectile-mode t))
```

project discovery

Customize `nougat-project-root` to set the location of the majority of your projects/repositories.

```
(projectile-discover-projects-in-directory
 (file-name-as-directory
  (expand-file-name
   (var-or 'nougat-project-root "~/src"))))
```

helm

Helm is a fast completion/selection framework for Emacs. It pops up a buffer with choices which you narrows by fuzzy search. This can be used for finding files, switching buffers, etc.

The following keys are bound by default:

default key	override symbol	description
M-x	kbd-helm-M-x	execute commands
C-h f	kbd-helm-apropos	get help for any symbol
C-x C-f	kbd-helm-find-files	find files
C-c y	kbd-helm-show-kill-ring	view kill ring history
C-x C-r	kbd-helm-recentf	open recently viewed files

```
(use-package helm
  :config
  (helm-mode 1)
  (require 'helm-config)
  (global-kbd-or 'helm-M-x "M-x"
                 'helm-apropos "C-h f"
                 'helm-find-files "C-x C-f"
                 'helm-mini "C-x b"
                 'helm-show-kill-ring "C-c y"
                 'helm-recentf "C-x C-r"))
```

ace-jump-helm-line

Use (`M-;` / `kbd-helm-ace-jump`) to show a unique letter combination next to each Helm candidate. Pressing a combination instantly selects that candidate.

```
(with-eval-after-load 'helm
  (use-package ace-jump-helm-line
    :commands ace-jump-helm-line
    :config (define-key helm-map
          (or-kbd kbd-helm-ace-jump "M-;")
          'ace-jump-helm-line)))
```

helm-bookmarks

Use (`C-x C-b` / `kbd-helm-bookmarks`) to manage bookmarks with Helm.

```
(with-eval-after-load 'helm
  (require 'helm-bookmark)
  (global-kbd-or 'helm-bookmark "C-x C-b"))
```

helm-descbinds

Use (`C-h b` / `kbd-helm-descbinds`) to inspect current bindings with Helm.

```
(use-package helm-descbinds
  :after (helm)
  :commands helm-descbinds
  :config
  (global-kbd-or 'helm-descbinds "C-h b"))
```

helm-flyspell

With `flyspell-mode` on, use (`C-;` / `kbd-helm-flyspell`) after a word to correct it with Helm.

```
(use-package helm-flyspell
  :after (helm)
  :commands helm-flyspell-correct
  :config (global-kbd-or 'helm-flyspell-correct "C-;"))
```

helm-org-rifle

Use (`M-r` / `kbd-helm-org-rifle`) to rifle through the current org-mode buffer or all open org-mode buffers if one is not focused.

```
(use-package helm-org-rifle
  :after (helm org)
  :commands helm-org-rifle-current-buffer
  :config
  (define-kbd-or
    org-mode-map 'helm-org-rifle-current-buffer "M-r"))
```

helm-projectile

Use (`C-x c p` / `kbd-helm-projectile`) to view the buffers and files in the current Projectile project.

```
(use-package helm-projectile
    :after (helm projectile)
    :commands helm-projectile
    :config
    (global-kbd-or 'helm-projectile "C-x c p"))
```

theme customizations

```
(set-face-attribute
 'helm-selection nil
 :inherit t
 :background (badger-color "bg+1")
 :foreground nil
 :height 1.0
 :weight 'ultra-bold
 :inverse-video nil)

(set-face-attribute
 'helm-source-header nil
 :inherit nil
 :underline nil
 :background (badger-color "bg")
 :foreground (badger-color "dull-red")
 :height 1.9)

(set-face-attribute
 'helm-header nil
 :inherit nil
 :height 0.8
 :background (badger-color "bg")
 :foreground (badger-color "teal"))

(set-face-attribute
 'helm-separator nil
 :height 0.8
 :foreground (badger-color "salmon"))

(set-face-attribute
 'helm-match nil
 :weight 'bold
 :foreground (badger-color "olive"))
```

magit

Use (`C-x g` / `kbd-magit-status`) to open the best front-end to Git there is!

```
(use-package magit
  :config
  (require 'magit)
  (global-kbd-or 'magit-status "C-x g"))
```

elfeed

boilerplate

```
(defun advice-unadvice (sym)
  "Remove all advices from symbol SYM."
  (interactive "aFunction symbol: ")
  (advice-mapc (lambda (advice _props) (advice-remove sym advice)) sym))

(defun elfeed-font-size-hook ()
  (buffer-face-set '(:height 1.35)))

(defun elfeed-visual-fill-hook ()
  (visual-fill-column-mode--enable))

(defun elfeed-show-refresh-advice (entry)
  (elfeed-font-size-hook)
  (visual-fill-column-mode 1)
  (setq word-wrap 1)
  (elfeed-show-refresh))

(defun elfeed-show ()
  (interactive)
  (elfeed)
  (delete-other-windows))
```

setup

```
(use-package elfeed
  :bind (("C-x w" . elfeed-show))
  :config
  (add-hook 'elfeed-search-update-hook 'elfeed-font-size-hook)
  (advice-unadvice 'elfeed-show-entry)
  (advice-add 'elfeed-show-entry :after 'elfeed-show-refresh-advice))

(use-package elfeed-org
  :after (elfeed)
  :config
  (elfeed-org)
  (setq rmh-elfeed-org-files (list "~/org/notes.org")))
```

language support

applescript

```
(use-package apples-mode
  :config
  (add-to-list 'auto-mode-alist '("\\.as\\'" . apples-mode)))
```

Elisp

context-help

Use (`C-c h` / `kbd-toggle-context-help`) to turn on a help-window that will automatically update to display the help of the symbol before point.

```
(defun toggle-context-help ()
  "Turn on or off the context help.
Note that if ON and you hide the help buffer then you need to
manually reshow it. A double toggle will make it reappear"
  (interactive)
  (with-current-buffer (help-buffer)
    (unless (local-variable-p 'context-help)
      (set (make-local-variable 'context-help) t))
    (if (setq context-help (not context-help))
      (progn
         (if (not (get-buffer-window (help-buffer)))
             (display-buffer (help-buffer)))))
    (message "Context help %s" (if context-help "ON" "OFF"))))

(defun context-help ()
  "Display function or variable at point in *Help* buffer if visible.
Default behaviour can be turned off by setting the buffer local
context-help to false"
  (interactive)
  (let ((rgr-symbol (symbol-at-point))) ; symbol-at-point http://www.emacswiki.org/cgi-bin/w
    (with-current-buffer (help-buffer)
     (unless (local-variable-p 'context-help)
       (set (make-local-variable 'context-help) t))
     (if (and context-help (get-buffer-window (help-buffer))
       rgr-symbol)
       (if (fboundp  rgr-symbol)
         (describe-function rgr-symbol)
       (if (boundp  rgr-symbol) (describe-variable rgr-symbol)))))))

(defadvice eldoc-print-current-symbol-info
  (around eldoc-show-c-tag activate)
  (cond
      ((eq major-mode 'emacs-lisp-mode) (context-help) ad-do-it)
      ((eq major-mode 'lisp-interaction-mode) (context-help) ad-do-it)
      ((eq major-mode 'apropos-mode) (context-help) ad-do-it)
      (t ad-do-it)))

(global-kbd-or 'toggle-context-help "C-c h")
```

lispy-mode

```
(use-package lispy
  :config
  (add-hook 'emacs-lisp-mode-hook (lambda () (lispy-mode 1)))
  (add-hook 'lisp-interaction-mode-hook (lambda () (lispy-mode 1))))
```

markdown-mode

All the internet uses it.

```
(use-package markdown-mode
  :commands (markdown-mode gfm-mode)
  :mode (("README\\.md\\'" . gfm-mode)
       ("\\.md\\'" . markdown-mode)
       ("\\.markdown\\'" . markdown-mode))
  :config (setq markdown-command "multimarkdown"))
```

python

elpy

```
(use-package elpy)
```

jedi

Jedi is an auto-completion server for Python.

```
(use-package jedi
  :init
  (progn
    (add-hook 'python-mode-hook 'jedi:setup)
    (setq jedi:complete-on-dot t)))
```

Typescript

```
(use-package typescript-mode)

(defun setup-tide-mode ()
  (interactive)
  (tide-setup)
  (flycheck-mode +1)
  (setq flycheck-check-syntax-automatically '(save mode-enabled))
  (eldoc-mode +1)
  (tide-hl-identifier-mode +1)
  (company-mode +1))

(use-package tide
  :config
  (add-hook 'before-save-hook 'tide-format-before-save)
  (add-hook 'typescript-mode-hook #'setup-tide-mode))
```

jsx

```
(use-package web-mode
  :config
  (require 'web-mode)
  (add-to-list 'auto-mode-alist '("\\.tsx\\'" . web-mode))
  (add-hook 'web-mode-hook
          (lambda ()
            (when (string-equal "tsx" (file-name-extension buffer-file-name))
              (setup-tide-mode))))
  ;; enable typescript-tslint checker
  (flycheck-add-mode 'typescript-tslint 'web-mode))
```

yaml

yaml-mode

```
(use-package yaml-mode
  :config
  (require 'yaml-mode)
  (add-to-list 'auto-mode-alist '("\\.yml\\'" . yaml-mode)))
```

f#

```
(use-package fsharp-mode
  :config
  (add-to-list 'auto-mode-alist '("\\.fs[iylx]?$" . fsharp-mode)))
```

```
(with-eval-after-load 'fsharp-mode
  (add-to-list 'exec-path "/nix/var/nix/profiles/default/bin")
  (add-to-list 'exec-path (expand-file-name "~/.nix-profile/bin"))
  (add-to-list 'auto-mode-alist '("\\.fs[iylx]?$" . fsharp-mode)))
```

tooling support

docker

```
(use-package dockerfile-mode
  :config
  (require 'dockerfile-mode)
  (add-to-list 'auto-mode-alist '("Dockerfile\\'" . dockerfile-mode)))
```

nix

nix-mode

```
(use-package nix-mode
  :init (add-to-list 'auto-mode-alist '("\\.nix?\\'" . nix-mode)))
```

nix-sandbox

```
(use-package nix-sandbox)
```

hydra

Hydra provides an easy way to create little pop-up interfaces with a collection of related single-key bindings.

```
(use-package hydra)
```

pretty-hydra

Pretty-hydra provides a macro that makes it easy to get good looking Hydras.

```
(use-package pretty-hydra
  :straight (pretty-hydra :type git :host github
                          :repo "jerrypnz/major-mode-hydra.el"
                          :files ("pretty-hydra.el"))
  :config
  (require 'pretty-hydra))
```

major-mode-hydra

Major-mode-hydra provides an macro for defining major-mode specific Hydras.

```
(use-package major-mode-hydra
  :straight (major-mode-hydra
             :type git :host github
             :repo "jerrypnz/major-mode-hydra.el"
             :files ("major-mode-hydra.el"))
  :config
  (require 'major-mode-hydra)
  (global-kbd-or 'major-mode-hydra "C-<f19>"))
```

hera

Hera provides for a few Hydra niceties including an API that allows your Hydras to form a stack.

```
(use-package hera
  :straight (hera :type git :host github :repo "dustinlacewell/hera")
  :config
  (require 'hera))
```

nougat-hydra

This is the main macro for defining Hydras in Nougat.

```
(defun nougat--inject-hint (symbol hint)
  (-let* ((name (symbol-name symbol))
        (hint-symbol (intern (format "%s/hint" name)))
        (format-form (cadr (eval hint-symbol)))
        (string-cdr (nthcdr 1 format-form))
        (format-string (string-trim (car string-cdr)))
        (amended-string (format "%s\n\n%s" format-string hint)))
    (setcar string-cdr amended-string)))

(defun nougat--make-head-hint (head default-color)
  (-let (((key _ hint . rest) head))
    (when key
      (-let* (((&plist :color color) rest)
            (color (or color default-color))
            (face (intern (format "hydra-face-%s" color)))
            (propertized-key (propertize key 'face face)))
      (format " [%s]: %s" propertized-key hint)))))

(defun nougat--make-hint (heads default-color)
  (string-join
   (cl-loop for head in heads
          for hint = (nougat--make-head-hint head default-color)
          do (pp hint)
          collect hint) "\n"))

(defun nougat--clear-hint (head)
  (-let* (((key form _ . rest) head))
    `(,key ,form nil ,@rest)))

(defun nougat--add-exit-head (heads)
  (let ((exit-head `(,(var-or 'kbd-hera-pop "SPC") (hera-pop) "to exit" :color blue)))
    (append heads `(,exit-head))))

(defun nougat--add-heads (columns extra-heads)
  (let* ((cell (nthcdr 1 columns))
       (heads (car cell))
       (extra-heads (mapcar 'nougat--clear-hint extra-heads)))
    (setcar cell (append heads extra-heads))))

(defmacro nougat-hydra (name body columns &optional extra-heads)
  (declare (indent defun))
  (-let* (((&plist :color default-color :major-mode mode) body)
        (extra-heads (nougat--add-exit-head extra-heads))
        (extra-hint (nougat--make-hint extra-heads default-color))
        (body (plist-put body :hint nil))
        (body-name (format "%s/body" (symbol-name name)))
        (body-symbol (intern body-name))
        (mode-support
         `(when ',mode
            (setq major-mode-hydra--body-cache
                  (a-assoc major-mode-hydra--body-cache ',mode ',body-symbol)))))
    (nougat--add-heads columns extra-heads)
```

```
    (when mode
      (remf body :major-mode))
    `(progn
       (pretty-hydra-define ,name ,body ,columns)
       (nougat--inject-hint ',name ,extra-hint)
       ,mode-support)))

;; (nougat-hydra hydra-test (:color red :major-mode fundamental-mode)
;;   ("First"
;;    (("a" (message "first - a") "msg a" :color blue)
;;     ("b" (message "first - b") "msg b"))
;;    "Second"
;;    (("c" (message "second - c") "msg c" :color blue)
;;     ("d" (message "second - d") "msg d"))))
```

hydra-bookmarks

Manage your bookmarks. Depends on `helm`.

```
(defun hydra--set-bookmark (&optional name)
  (interactive)
  (let ((file-name (file-name-nondirectory (buffer-file-name)))
      (path-parts (split-string (file-name-directory (buffer-file-name))))
      (name (substring (buffer-file-name))(format "%s" ))))
  (save-excursion
    (beginning-of-buffer))
  (let ((filename (or name (format "%s:%s" (buffer-name) (line-number-at-pos)))))
    (bookmark-set filename)))

(nougat-hydra hydra-bookmarks (:color blue)
  ("Bookmarks" (("a" (hydra--set-bookmark) "add this")
              ("n" (hydra--set-bookmark (read-from-minibuffer "Name")) "add named")
              ("l" (helm-bookmarks) "list all"))))
```

hydra-help

Many of the Emacs help facilities at your fingertips!

```
(nougat-hydra hydra-help (:color blue)
  ("Describe"
   (("c" describe-function "function")
    ("p" describe-package "package")
    ("m" describe-mode "mode")
    ("v" describe-variable "function"))
   "Keys"
   (("k" describe-key "key")
    ("K" describe-key-briefly "brief key")
    ("w" where-is "where-is")
    ("b" helm-descbinds "bindings"))
   "Search"
   (("a" helm-apropos "apropos")
    ("d" apropos-documentation "documentation")
    ("s" info-lookup-symbol "symbol info"))
   "Docs"
   (("i" info "info")
    ("n" helm-man-woman "man")
    ("h" helm-dash "dash"))
   "View"
   (("e" view-echo-area-messages "echo area")
    ("l" view-lossage "lossage")
    ("c" describe-coding-system "encoding")
    ("I" describe-input-method "input method")
    ("C" describe-char "char at point"))))
```

hydra-mark

```
(defun unpop-to-mark-command ()
  "Unpop off mark ring. Does nothing if mark ring is empty."
  (when mark-ring
    (setq mark-ring (cons (copy-marker (mark-marker)) mark-ring))
    (set-marker (mark-marker) (car (last mark-ring)) (current-buffer))
    (when (null (mark t)) (ding))
    (setq mark-ring (nbutlast mark-ring))
    (goto-char (marker-position (car (last mark-ring))))))

(defun push-mark ()
  (interactive)
  (set-mark-command nil)
  (set-mark-command nil))

(nougat-hydra hydra-mark (:color pink)
  ("Mark"
   (("m" push-mark "mark here")
    ("p" (lambda () (interactive) (set-mark-command '(4))) "previous")
    ("n" (lambda () (interactive) (unpop-to-mark-command)) "next")
    ("c" (lambda () (interactive) (setq mark-ring nil)) "clear"))))
```

hydra-projectile

```
(defun projectile-dwim ()
  (interactive)
  (if (string= "-" (projectile-project-name))
      (helm-projectile-switch-project)
    (hydra-projectile/body)))

(nougat-hydra hydra-projectile (:color blue)
  ("Open"
   (("f" helm-projectile-find-file "file")
    ("r" helm-projectile-recent "recent")
    ("p" helm-projectile-switch-project "project")
    ("d" helm-projectile-find-dir "directory"))
   "Search"
   (("o" projectile-multi-occur "occur")
    ("a" projectile-ag))
   "Buffers"
   (("b" helm-projectile-switch-to-buffer "switch")
    ("k" helm-projectile-kill-buffers "kill"))
   "Cache"
   (("C" projectile-invalidate-cache "clear")
    ("x" projectile-remove-known-project "remove project")
    ("X" projectile-cleanup-known-projects "cleanup"))))
```

hydra-registers

```
(nougat-hydra hydra-registers (:color pink)
  ("Point"
   (("r" point-to-register "save point")
    ("j" jump-to-register "jump")
    ("v" view-register "view all"))
   "Text"
   (("c" copy-to-register "copy region")
    ("C" copy-rectangle-to-register "copy rect")
    ("i" insert-register "insert")
    ("p" prepend-to-register "prepend")
    ("a" append-to-register "append"))
   "Macros"
   (("m" kmacro-to-register "store")
    ("e" jump-to-register "execute"))))
```

hydra-window

```
(use-package ace-window)
(winner-mode 1)

(nougat-hydra hydra-window (:color red)
  ("Jump"
   (("h" windmove-left "left")
    ("l" windmove-right "right")
    ("k" windmove-up "up")
    ("j" windmove-down "down")
    ("a" ace-select-window "ace"))
   "Split"
   (("q" split-window-right "left")
    ("r" (progn (split-window-right) (call-interactively 'other-window)) "right")
    ("e" split-window-below "up")
    ("w" (progn (split-window-below) (call-interactively 'other-window)) "down"))
   "Do"
   (("d" delete-window "delete")
    ("o" delete-other-windows "delete others")
    ("u" winner-undo "undo")
    ("R" winner-redo "redo"))))
```

hydra-yank-pop

```
(nougat-hydra hydra-yank-pop (:color red)
  ("Yank/Pop"
   (("y" (yank-pop 1) "previous")
    ("Y" (yank-pop -1) "next")
    ("l" helm-show-kill-ring "list" :color blue))))

(global-set-key
 (kbd (var-or 'kbd-hydra-yank-pop "C-y"))
 (lambda () (interactive) (yank) (hydra-yank-pop/body)))
```

hydra-zoom

```
(nougat-hydra hydra-zoom (:color red)
  ("Zoom"
   (("i" text-scale-increase "in")
    ("o" text-scale-decrease "out"))))
```

emacs-lisp

```
(nougat-hydra hydra-elisp (:color blue :major-mode emacs-lisp-mode)
  ("Execute"
   (("d" eval-defun "defun")
    ("b" eval-current-buffer "buffer")
    ("r" eval-region "region"))
   "Debug"
   (("D" edebug-defun "defun")
    ("a" edebug-all-defs "all definitions" :color red)
    ("A" edebug-all-forms "all forms" :color red))))
```

org-mode

hydra-org-goto-first-sibling

```
(defun hydra-org-goto-first-sibling () (interactive)
       (org-backward-heading-same-level 99999999))
```

hydra-org-goto-last-sibling

```
(defun hydra-org-goto-last-sibling () (interactive)
       (org-forward-heading-same-level 99999999))
```

hydra-org-parent-level

```
(defun hydra-org-parent-level ()
  (interactive)
  (let ((o-point (point)))
    (if (save-excursion
          (beginning-of-line)
          (looking-at org-heading-regexp))
        (progn
          (call-interactively 'outline-up-heading)
          (org-cycle-internal-local))
      (progn
      (call-interactively 'org-previous-visible-heading)
      (org-cycle-internal-local)))
    (when (and (/= o-point (point))
               org-tidy-p)
      (call-interactively 'hydra-org-tidy))))
```

hydra-org-child-level

```
(defun hydra-org-child-level ()
  (interactive)
  (org-show-entry)
  (org-show-children)
  (when (not (org-goto-first-child))
    (when (save-excursion
          (beginning-of-line)
          (looking-at org-heading-regexp))
      (next-line))))
```

hydra

```
(nougat-hydra hydra-org (:color red :major-mode org-mode)
  ("Shift"
   (("K" org-move-subtree-up "up")
    ("J" org-move-subtree-down "down")
    ("h" org-promote-subtree "promote")
    ("l" org-demote-subtree "demote"))
   "Travel"
   (("p" org-backward-heading-same-level "backward")
    ("n" org-forward-heading-same-level "forward")
    ("j" hydra-org-child-level "to child")
    ("k" hydra-org-parent-level "to parent")
    ("a" hydra-org-goto-first-sibling "first sibling")
    ("e" hydra-org-goto-last-sibling "last sibling"))
   "Perform"
   (("r" (lambda () (interactive)
         (helm-org-rifle-current-buffer)
         (call-interactively 'org-cycle)
         (call-interactively 'org-cycle)) "rifle")
    ("v" avy-org-goto-heading-timer "avy")
    ("L" org-toggle-link-display "toggle links"))))
```

```
(global-set-key (kbd "C-<f19>") 'major-mode-hydra)
```

spotify

```
;; (use-package spot4e
;;   :straight (spot4e :type git :host github :local-repo "~/src/spot4e")
;;   :config
;;   (setq spot4e-refresh-token "AQDA5zMq9Juoi76685iFALQWp17uWpZVTCPCqTMI2rZFUwX-lmH7Z9idbhP
;;   (setq spot4e-access-token nil)
;;   (run-with-timer 0 (* 60 59) 'spot4e-refresh))

;; (spot4e-authorize)

;; (pretty-hydra-define hydra-spotify (:color red :hint nil)
;;   ("Player" (("h" (spot4e-player-next) "next")
;;              ("l" (spot4e-player-previous) "previous")
;;              ("j" (spot4e-player-pause) "pause")
;;              ("k" (spot4e-player-play) "play")
;;              ("v" (spot4e-set-volume) "volume"))

;;    "Search" (("t" (spot4e-helm-search-tracks) "tracks")
;;              ("a" (spot4e-helm-search-artists) "artists" :push t)
;;              ("A" (spot4e-helm-search-albums ) "albums")
;;              ("r" (spot4e-helm-search-recent-tracks ) "recent")
;;              ("c" (spot4e-helm-search-categories) "categories")))
;;   (("<f19>" (hera-pop) "nil" :color blue)
;;    ("R" (spot4e-refresh) "nil" :color red)))

;; (hydra-spotify/body)
```

entrypoint

ep-notes-file

```
(setq ep-notes-file (my/org-file-name "notes.org"))
```

ep-notes-find-file

```
(defun ep-notes-find-file () (find-file ep-notes-file))
```

ep-notes-visit

```
(defun ep-notes-visit (&rest olp) (org-olp-visit ep-notes-file olp))
;; (ep-notes-visit "Workiva" "Runbooks")
```

ep-notes-select-then-visit

```
(defun ep-notes-select-then-visit (&rest olp) (org-olp-select-then-visit ep-notes-file olp))
;; (ep-notes-select-then-visit "Workiva" "Tasks")
```

hydra-default

```
(nougat-hydra hydra-default (:color blue)
  ("Editing"
   (("p" (hera-push 'hydra-projectile/body) "projectile")
    ("r" (hera-push 'hydra-registers/body) "registers")
    ("m" (hera-push 'hydra-mark/body) "mark"))
   "UI"
   (("w" (hera-push 'hydra-window/body) "windows")
    ("z" (hera-push 'hydra-zoom/body) "zoom"))
   "Help"
   (("h" (hera-push 'hydra-help/body) "help"))))

(global-set-key (kbd "<f19>") (lambda () (interactive) (hera-start 'hydra-default/body)))
```