Rock Your Emacs Tutorial

Ben Sturmfels <ben@sturm.com.au>

LCA 2014, 8 January 2014

Do you love Emacs, but have never understood the strange code with lots of brackets? You're missing out on one of the great joys of Emacs — customising it to work exactly the way you want. It turns out that Emacs is little more than an interpreter for Lisp code interpreter, and once you know a little Emacs Lisp, almost anything is possible.

After attending this tutorial, you will know how to:

- read basic Emacs Lisp code
- modify existing Lisp code and write new
- make Emacs your own with persistent customisations
- customise behaviour for distinct modes
- bind your favourite commands to keys
- answer your own questions with the amazing documentation system
- use built-in Emacs Lisp development tools like the debugger
- write your own reusable extensions modules for Emacs customisations

The tutorial will have some interactive components, so bringing a laptop is recommended. Sharing with a friend will work well too though.

This tutorial will be enjoyed most if you already have a little programming experience.

1 Tutorial overview

Requirements

1. Copy of talk notes (PDF/HTML/hardcopy)
   - http://sturm.com.au/2014/talks/rock-your-emacs-lca/
2. GNU Emacs 24
3. Emacs Lisp (.el) source files, eg. `emacs24-el`
4. Emacs documentation in Info format, eg. `emacs24-common-non-dfsg`
   - Includes *Emacs Manual*, *Emacs Lisp Intro* and *Emacs Lisp Reference Manual*

Tutorial is aimed at

- frequent Emacs users
- Emacs Lisp “copy and pasters”
- some programming experience

Motivation

- make Emacs yours
- a little Lisp makes Emacs more fun
- barrier to your own customisations is lower than you expect
- Lisp is an interesting language
- how has Emacs attained this longevity and love?

Tutorial format

- talking + demonstration with Emacs + questions
- please type along
- bonus material depending on time
- function ideas list at back of these notes

Content outline

1. some Emacs lisp examples
2. tools for reading/writing Lisp
3. making persistent changes
4. dive deeply into the language and syntax
5. **5 min break about 2pm**
6. example customisations/extensions
7. built-in help and tools

This is not

- how to use Emacs
- memorising lots of lisp — use help instead
- heavy on Computer Science
- preaching Lisp for general purpose programming

2 Let's look at some Lisp

Lisp syntax

- simple syntax eg. `(+ 1 2)`
- prefix notation for function and arguments
- lots of parentheses, but tools help
  - auto-indenting, visual matching

Evaluating code (from anywhere)

- `M-x eval-expression (C-:)`
- `M-x eval-last-sexp (C-x C-e)`
- `M-x eval-region`
- `M-x eval-buffer`

Quick help (most useful first)

- `M-x help-for-help` (`C-h ?`)
- `M-x describe-variable` (`C-h v`)
- `M-x describe-function` (`C-h f`)
- `M-x info` (`C-h i`)
- `M-x describe-key` (`C-h k`)
- `M-x describe-mode` (`C-h m`)
- `M-x view-echo-area-messages` (`C-h e`)

Example 1: Send a message to user

```
(message "Hello Perth!")
(message (concat "Hello " user-full-name))
```

Example 2: What happens if you make a typo?

```
(massage "Hello Perth!")
(message (concat "Hello " user-fool-name))
```

Example 3: Change an existing global variable

```
(setq user-full-name "Charlie Parker")
(compose-mail)
```

Example 4: Define a new global variable

```
(setq user-favourite-food "mango")

;; Better still, use ‘defvar’.
(defvar user-favourite-food "mango"
  "Favourite food of the user logged in.")
```

Example 5: Define a new function

```
(defun insert-heart ()
  "Insert a heart symbol."
  (interactive)
  (insert ""))
```

- **demo**: this is now part of Emacs, via help functions
- write, eval, test, repeat - tight feedback loop

3 How Lisp fits into Emacs

Emacs is just a Lisp interpreter

- C for fundamental features/performance
- mostly Lisp — most actions run a Lisp function
- **demo**: `find-file` using Lisp
- don’t worry about all the keybindings
  - interactive commands are an interface tweak
  - binding keys to commands are an interface tweak

There’s no “extension API”

- Lisp provides full control over Emacs
- your extensions are indistinguishable from primitives
  - unique and powerful!
- ultimately can replace built-in functions with your own

Example 6: Don’t do this, it’s silly

```
(defun find-file ()
  "Break the ‘find-file’ command."
  (interactive)
  (error "Computer says: no."))
```

4 Tools for reading and writing Lisp

Don't memorise, use the awesome help

- **demo**: Emacs suggests more appropriate functions (eg. `next-line`)

Browse source

- **demo**: browse source
  - essence of free software, freedoms 1 & 2

Major modes

- *Lisp Interaction mode* for initial scratch buffer (`lisp-interaction-mode`)
  - `C-j` is `eval-print-last-sexp`
- *Emacs Lisp mode* for editing programs (`emacs-lisp-mode`)
- *Inferior Emacs Lisp mode* for shell interface (`ielm`)

Minor modes

- turn on *Eldoc mode* (`eldoc-mode`)
- turn on *Show Paren mode* (`show-paren-mode`)

5 Making your extensions persistent

Emacs initialisation

1. (mine at least) loads `/usr/share/emacs/site-lisp/debian-startup.el`
2. loads `site-run-file` — empty by default
3. looks for `~/.emacs`, `~/.emacs.el` or `~/.emacs.d/init.el`
   - your extensions are usually loaded here

Example 7: A snippet of your .emacs file.

```
;; Highlight matching parentheses.
(show-paren-mode)
```

Looking after your .emacs

- put it in version control – it's a program after all
- share across computers
- mine is a symlink to `~/dotfiles/.emacs`

Resolving .emacs errors

- whoops, I made a mistake
  - use `emacs --debug-init`
- **demo**: fixing startup errors

Loading things external to .emacs

- make a separate directory for your `.el` files
- add the directory to `load-path`
  - eg. `(add-to-list 'load-path "~/.elisp")`
  - avoid adding `"~/.emacs.d"= to =load-path`
- `(load "filename")` for regular files
- `(require 'blah)` if file contains `(provides 'blah)`
  - mechanism to avoid loading a feature twice

Example 8: Loading an external Lisp file

```
;; In ~/.elisp/dates.el:
(defun insert-iso-date ()
  "Insert the current date in YYYY-MM-DD format."
  (interactive)
  (let ((iso-date (format-time-string
                   "%Y-%m-%d" (current-time))))
    (insert iso-date)))

;; In ~/.emacs:
(add-to-list 'load-path "~/.elisp")
(load "dates")
```

6 Overview of Emacs Lisp for programmers

What's Lisp?

- Lisp developed in late 1950s
- originated in "computer science" but very pragmatic
- simple and elegant syntax
  - code and data in same syntax
  - fully featured
- rich history, many talented programmers, good writers

Emacs Lisp language

- inspired by Maclisp (MIT 1960s) and Common Lisp (standardised 1980s)
  - features excluded/simplified to reduce memory
- a "Lisp-2", meaning separate namespaces for functions and variables
- no earmuffs on globals, `*global-var*`

Lisp vocab 1

symbol eg. `find-file`

- name for something — bound/unbound
- case-sensitive, but use lowercase

symbolic expression/sexp/expression eg. `(+ 1 2)`

list eg. `'(apple orange pear)`

- made up of nested cons cells, eg. `(cons 1 (cons 2 (cons 3 nil)))`
- like linked lists
- `car`, `cdr` — unfortunate names

quote take as written, don't eval

- eg. `'find-file` or `'(+ 1 2)`
- shorthand for `(quote (+ 1 2))`

Lisp vocab 2

truth values `nil`, `t`

- everything is true except `nil` or `()`
- zero is true
- `nil` and `()` are the same, use in context

numbers integer `43`, float `3.33`

text character `?a`, string `"abc"`

Lisp vocab 3

functions

- first-class types
- lambda anonymous functions
  - eg. `(lambda (x) (1+ x))`
- **demo**: functions as arguments
  - `(mapcar (lambda (x) (1+ x)) '(1 2 3))`
- optional arguments with `&optional`
- variable number of arguments with `&rest`

special form not many of these

- builtin, eg. `eval`
- macro, eg. `dolist`

predicates boolean functions , eg. `listp` or `buffer-modified-p`

Some composite data types

alist association list

- eg. `'((loc . "Perth") (temp . 39))`
- substitute for hash table to a point, using `assoc`
- list of cons cells

plist property list

hash table use an alist if you can

vector use a list if you can

Emacs-specific vocab

interactive makes a function available to user interface

command commands are functions with `(interactive)` applied

point your cursor location

mark other end of selection

region between point and mark

marker data type of point and mark

Other language features

- functions return last expression in body
- global namespace, no modules
  - use prefix, eg. `rmail-`
- optional function parameters: `&optional ...`
- variable number of parameters: `&rest`
- no named function parameters

Slightly more obscure features

- macros: change the language yourself
- no tail-call recursion optimisation
- symbols starting with colon, eg. `:group` are "keywords"
  - evaluate to themselves, much like numbers, strings and arrays
- quasi-quoting: `` `(a b ,user-full-name) ``
  - or use `(list 'a 'b user-full-name)`

7 Break

8 Tweaking Emacs

Customize

- interactive tool for configuring variables
- **demo**: customize `erc-nick`
- good for exploring global vars and their values
- can't define new functions
  - a little constraining

Manually setting variables

- **demo**: translate between customization and own Lisp
- changes made on the fly, no restart (of course)

Changing key bindings

- can bind/rebind any command to any keystroke
- `C-c [single-letter]` reserved for you
- `<f5>` to `<f12>` handy too
- `"\C-ch"` or `[?\C-c ?h]` are internal lisp representations
  - neater to use `kbd`, eg. `(kbd "C-c h")`, `(kbd "<f5>")`
- global (everywhere) or local (one buffer)

Global key bindings (everywhere)

- `global-set-key`:
  - local bindings shadow global (useful)
- unset with `global-unset-key`
- these set/unset keys in `current-global-map`

Example 9: Global key binding for `insert-heart`

```
(global-set-key (kbd "C-c h") 'insert-heart)
```

Mode-specific key bindings

- `define-key`: you need to specify the keymap
- or use `local-set-key` in a mode hook

Example 10: Mode-specific key binding for `insert-iso-date`

```
(define-key python-mode-map (kbd "C-c .") 'insert-iso-date)
```

9 Other help

Emacs Lisp Reference Manual

- thorough and well written
- **highly recommended**

Info mode tips

- up: `u`
- last: `l`
- forward page: `SPC`
- follow link: `RET`
- navigate menu: `m`
- incremental search handy too, ie. `C-s`

Problem of finding an unknown variable/function

- writing your own code
- does the function/variable you want already exist?
  - how could you find it?

Quick help

1. guess variable/function name using `describe-function` or `describe-variable`
2. try *Emacs Lisp Reference Manual*
   - browse the main contents and/or incremental search

Deeper help

1. search function/variable comments: `apropos-documentation`
2. search all info manuals: `info-apropos`
3. EmacsWiki search
4. Web search
5. IRC: Ask for help in `#emacs` on Freenode (`erc-tls`)

Let's practise with help

1. Look up "formatting strings" in the reference manual
2. Figure out what function is bound to `C-x r d`

10 More substantial Emacs extensions

Customising modal behaviours with hooks

- like event handlers
- normal hooks, eg. `foo-hook`
- abnormal hooks take arguments/return something, eg. `foo-functions`
- hooks are everywhere, see `Standard Hooks` in manual
- major-modes: `mymodename-hook` runs in last steps of initialisation
- use `add-hook`

Example 11: Using hook functions

```
;; Enable ‘eldoc’ when using Emacs Lisp mode.
(add-hook 'emacs-lisp-mode-hook 'eldoc-mode)

;; Highlight whitespace when programming
(add-hook 'prog-mode-hook
          (lambda ()
            (setq whitespace-style
                  '(face tabs trailing lines-tail))
            (whitespace-mode)))
```

Interactive functions

- **demo**: `insert-heart` as both interactive and non-interactive
- generally for side-effect, rather than return value
- `(interactive)`:
  - makes function available through user interface
  - maintains undo
  - makes Emacs supply or prompt for function arguments
  - suppresses the return value
- see manual for codes for `interactive`
  - can use Lisp functions instead

Example 12: Redact region

```
(defun redact-region (beg end char)
  "Replace region from BEG to END with character CHAR."
  (interactive "r\ncRedact character: ")
  (save-excursion
    (goto-char beg)
    (while (< beg end)
      (if (eq (char-after) ?\n)
          (forward-char 1)
        (progn
          (delete-char 1)
          (insert char)))
      (setq beg (point)))))
```

Example 13: Redact region (without codes)

```
(defun redact-region (beg end char)
  "Replace region from BEG to END with character CHAR."
  (interactive
   (list (region-beginning) (region-end)
         (read-char "Redact character: ")))
  (save-excursion
    (goto-char beg)
    (while (< beg end)
      (if (eq (char-after) ?\n)
          (forward-char 1)
        (progn
          (delete-char 1)
          (insert char)))
      (setq beg (point)))))
```

Writing a minor mode

- use `define-minor-mode` macro and fill in the blanks
- loads of possibilities, but low barrier to entry

Example 14: Presentation mode with large font

```
(define-minor-mode presentation-mode
  "Toggle Presentation mode.
When Presentation mode is enabled, the default faces
are larger for easy reading."
  nil
  " Pres"
  :global t
  (if presentation-mode
      (set-face-attribute 'default nil :height 158)
    (set-face-attribute 'default nil :height 98)))
```

11 Debugging

Edebug source level debugger

- there's an "always on" debugger — gives backtrace upon errors
- `edebug-defun` (or use menu)
- add breakpoints
- you can add permanent breakpoint with `(edebug)` (still needs to be instrumented)

Example 15: Try debugging with Edebug

```
(defun greet ()
  (interactive)
  (message (concat "Hello " user-full-name)))
```

12 Summary

Further reading

Introduction to Emacs Lisp
- good first few chapters, plus "Emacs Initialization" and "Debugging"
- difficulty and relevance varies
- don't hesitate to skip ahead

EMACS: The Extensible, Customizable Display Editor
- high-level discussion of the design of Emacs by Richard Stallman

- a little out of date, but very interesting

The Land of Lisp, Conrad Barski
- about Common Lisp, but has lots of relevant background

Lisp in other free software

Guile (Scheme Lisp dialect) used in some GNU software, including:

GnuCash uses Guile

Gimp uses Script-Fu (Scheme)

GNU Guix new functional package manager uses Guile

Conclusion

- hope brackets are less scary now
  - you understand some lisp
  - comfortable with built-in help
- text editors are personal
  - distance from thought to change is low here
  - make Emacs suit **your** needs

13 Bonus sections

Bonus: Some tips

- `eq` for comparing symbols, `equal` for everything else
- `if`, `when` or `unless` can only contain one expression — use `progn` or `cond`
- use `let` to create local variables, take care when using `setq`
- Lisp and especially Scheme are big on recursion
  - less useful in Emacs due to low default stack limit and no tail-call optimisation
- Lispers often prefer a functional style, but aren't pedantic
  - Emacs Lisp is all about the side-effects

Bonus: Common Lisp functions

- "Lisp" in the web is usually Common Lisp
- found some code that uses `loop`
- library `cl-lib` emulates some Common Lisp features, guilt free

```
(require 'cl-lib)
(cl-remove-if-not 'cl-oddp '(1 2 3 4))
```

Bonus: Saving keyboard macros

- record, save and bind to keys
  - `kmacro-name-last-macro`, then `insert-kbd-macro`
- **demo**: linkify a list lines
- no conditionals

Bonus: Useful functions

Not a complete list, just some ideas. See the reference manual.

Strings `concat`, `substring`, `format`

Lists `car`, `cdr`, `cons`, `push`, `pop`, `add-to-list`, `assoc`, `mapcar`, `mapc`, `reverse`

Interacting with the user `message`, `read-string`, `read-file-name`

Comparing things `eq`, `equal`, `and`, `or`, `not`, `version<`

Loops `while`, `dolist`, `dotimes`

Conditionals `if`, `when/unless`, `cond`

Working with/in buffers `point`, `mark`, `insert`, `save-excursion`, `with-current-buffer`, `save-restriction`, `forward-char`, `forward-line`, `kill-region`, `move-to-left-margin`, `move-end-of-line`, `filter-buffer-substring`

Searching `search-forward`, `re-search-forward`, `looking-at`

License