GNU Emacs Brevarium

INTRODUCTION

On a text-only terminal, Emacs occupies the whole screen. On windowing systems, it creates its own windows to use. The term `frame` means an entire text-only screen or an entire window used by Emacs. Emacs usually starts with one frame, but you can add others.

When you start Emacs, the entire `frame` except for the first and last lines is devoted to the text you are editing. This area is called the `window`. The first line of the `frame` is the `menu bar` and the last line is the `echo area`.

You can subdivide the large text `window` horizontally or vertically into multiple text windows, each of which can be used for a different file. `window` always refers to the subdivisions of a `frame`.

The `window` that the cursor is in is the *selected* `window`, in which the editing takes place. Most Emacs commands implicitly apply to the text in the selected `window`.

Each text `window`'s last line is a `mode line`, which displays status information such as what `buffer` is being displayed above it in the `window`, what major and minor modes are in use, and whether the buffer contains unsaved changes.

Within Emacs, the cursor shows the location at which editing commands will take effect. This location is called the `point`. You can move `point` with keyboard commands or by clicking the mouse. While the cursor appears to point *at* a character, you should think of `point` as being *between* two characters; the `point` exists *before* the character that appears under the cursor and after the character preceding the cursor.

The `mark` can be set to remember a location between two characters in the `buffer`. The `region` contains all of the characters between `point` and `mark`.

The area at the bottom of the `frame` below the `mode line` is the `echo area`. It is used to display small amounts of text for various purposes.

Unlike `vi`, Emacs is a modeless editor. Emacs functions are `bound`, or assigned, to keys on the keyboard. Pressing a key or key-sequence invokes the function bound to that key or key-sequence. `C-a` means hold down the CTL key *while* pressing the a key. The CTL key is *always* held down while pressing another key. On some systems, ESC functions the same way as CTL, in that it is held down *while* pressing another key. On some systems, ESC must be pressed and relased, followed by the remaining key(s). Some systems support both modes of operation. The `M` in the sequence `M-f` comes from the word META, which is an older name for ESC.

Introduction *continues in the next column…*

Introduction *continues from the previous column…*

ENTERING AND EXITING

The usual way to invoke Emacs is with the shell command `emacs`. If you run Emacs from a shell window, run it in the background with `emacs&`.

When Emacs starts up, it makes a buffer named `**scratch**`, which is the buffer you start out in. At this point, you will probably want to visit one or more files (e.g., via `C-x C-f` as described below).

The two most common ways to exit Emacs are:

`C-z` `suspend-emacs`
If running in a graphical display, iconify the current frame. Otherwise, suspend execution and return to the shell prompt; resume with `fg` or `%emacs`.

`C-x C-c` `save-buffers-kill-emacs`
Offer to save each buffer, then kill this Emacs process.

INVOKING COMMANDS

Most of the time, you invoke a command via its key binding (e.g., Pressing `C-f` invokes the command `forward-char`). However, there will be times when you wish to invoke a command that is not bound to a key or for which you do not know its binding. In those situations, the key sequence `M-x` can be used to invoke the desired command by typing the command's name.

`M-x` `execute-extended-command`
command RET [*arguments* RET]
Read *command*, which is the name of an Emacs command, then read arguments for *command*, if any, then call *command*.

Commands in GNU Emacs are implemented as Lisp functions. In the documentation, and elsewhere, you may see *command* and *function* used interchangeably.

COMMAND DOCUMENTATION

The bulk of this reference comprises entries describing Emacs commands, their key bindings (if any), their additional input (if any), and the command's behavior. Each entry describing a command is a row in a table of commands; each entry is organized in the following manner:

`keyseq` ① `command` ②
additional input ③
description of command ④

① `keyseq`
The key sequence on the keyboard that `command` is bound to. Typing the key sequence will invoke the command `command`. If `keyseq` is ⟦`M-x`⟧ (`M-x` enclosed in double brackets), the the key sequence is *not* bound to `command`, rather, the command must be invoked manually by typing `M-x`, followed by `command`, followed by RET.

② `command`
The name of the command to invoke.

③ *additional input*
If `command` accepts or requires additional keyboard input (other than a *prefix argument* - see below), that additional input is described here. If `command` does not accept additional keyboard input, *additional input* is omitted and *description of command* is moved up one line to begin here.

④ *description of command*
A description of the command's behavior.

HELP

GNU Emacs has an extensive help system. It's worth spending a little time with it to understand the resources it offers. All of the command documentation in this document was obtained using `describe-function`.

`C-h b` `describe-bindings`
Display a buffer showing a list of all defined keys, in order of precedence.

Help *continues in the next column…*

Help *continues from the previous column…*

`C-h c` `describe-key-briefly`
keyseq
Prompt for a key sequence and describe its binding in the echo area.

`C-h f` `describe-function`
command RET
Prompt for a command name and and display its description and keyboard binding(s), if any.

`C-h k` `describe-key`
keyseq
Prompt for a key sequence and display the command name that is bound to the sequence.

`C-h m` `describe-mode`
display bindings for current major mode, the currently enabled minor modes, and the names of all hooks the mode calls

`[M-x]` `apropos`
pattern RET
Show all meaningful Lisp symbols whose names match *pattern*.

EDITING TEXT

Inserting Text

To insert printing characters into the text you are editing, just type them. Each character that you type is inserted at `point`, moving the cursor forward. To delete what you have just inserted, use DEL. DEL deletes the character before the cursor (actually the character to the left of `point`). To end a line and start a new one, type RET.

Emacs can split lines automatically when they become too long, if you turn on a special *minor mode* called *Auto Fill* mode (see below).

If you prefer to have characters replace (overwrite) existing text rather than shove it to the right, you can enable the *Overwrite* minor mode.

Direct insertion works for printing characters and SPC, but other characters act as editing commands and do not insert themselves. If you need to insert a control character, use `quoted-insert`.

char `self-insert-command`
Keys for printed characters, such as `a-z, A-Z, 0-9`, punctuation, symbols, etc., are bound to `self-insert-command`. When the key is pressed, `self-insert-command` is invoked and the character associated with the key is inserted at `point`. The numeric prefix `N` says how many times to repeat the insertion. For example, pressing the `a` key calls `self-insert-command` which inserts the character `a` into the buffer at point.

`C-q` `quoted-insert`
Read the next character and insert it. Useful for inserting control characters.

Changing the Location of Point

`C-a` `beginning-of-line`
Move point to beginning of line.

`C-e` `end-of-line`
Move point to end of line.

`C-f` `forward-char`
Move point one character forward.

`C-b` `backward-char`
Move point one character backward.

`M-f` `forward-word`
Move point forward one word.

`M-b` `backward-word`
Move point backward one word.

`C-n` `next-line`
Move point down one line.

`C-p` `previous-line`
Move point up one line.

`M-r` `move-to-window-line`
Move point to left margin, vertically centered in the windows. Text does not move on the screen.

`M-<` `beginning-of-buffer`

Editing Text *continues in the next column…*

Editing Text *continues from the previous column…*

Move point to the top of the buffer.

`M->` `end-of-buffer`
Move point to the end of the buffer.

`M-g c` `goto-char`
number RET
Read a number and move point to the *number*th character in the buffer.

`M-g g` `goto-line`
number RET
Read a number and move point to line *number*.

`C-v` `scroll-up-command`
Scroll text upward by nearly a full screen.

`M-v` `scroll-down-command`
Scroll text downward by nearly a full screen.

Erasing Text

`C-d` `delete-backward-character`
Delete the character before `point`. Usually bound to DEL also.

`C-k` `kill-line`
Kill forward from point to end of line.

`M-k` `kill-sentence`
Kill forward from point to end of sentence.

`C-x` DEL `backward-kill-sentence`
Kill backward from point to beginning of sentence.

`C-M-k` `kill-sexp`
Kill s-expression

`M-d` `kill-word`
Kill forward from point to end of next word.

`M-`DEL `backward-kill-word`
Kill backward from point to end of next word.

`C-w` `kill-region`
Kill text between point and mark, saving deleted text in `killring`. `C-y` can retrieve the text from the `killring`.

`M-w` `kill-ring-save`
Save region as last killed text without actually killing it.

`M-^` `delete-indentation`
join previous line to current line

`M-`SPC `just-one-space`
remove all but one space at point

`M-z` `zap-to-char`
char
Kill up to and including, the *ARG*th occurrence of *char*.

`[M-x]` `zap-up-to-char`
char
Kill up to, but not including, the *ARG*th occurrence of *char*. To bind this to `M-z`, see the elisp section.

Undoing Changes

`C-x u` `undo`
Undo one batch of changes, usually one command.

`C-_` `undo`
The same. Can be easier if you require multiple undos.

`[M-x]` `revert-buffer`
Replace current buffer text with the text of the visited from on disk. Undoes all changes since last visit or save.

Files

`C-x C-f` `find-file`
filename RET
Switch to buffer visiting *filename*, creating one if none exits.

`C-x C-s` `save-buffer`
Save current buffer in visited file, if modified.

`C-x C-v` `find-alternate-file`
filename RET
Find *filename*, select its buffer, kill the previous buffer.

`C-x` RET `f` `set-buffer-file-encoding-system`
encoding RET
Prompts for desired encoding in mini-buffer then changes buffer's encoding and marks buffer as modified. `C-x C-s` will save the buffer.

Editing Text *continues in the next column…*

Editing Text continues from the previous column…

Blank Lines

`C-o` `open-line`
Insert one or more blank lines at `point`.

`C-x C-o` `delete-blank-lines`
Delete all but one of many blank lines.

Cursor Position and Other Information

⟦`M-x`⟧ `what-line`
Print line number at `point`.

⟦`M-x`⟧ `line-number-mode`
Toggle automatic display of line number.

`M-=` `count-lines-region`
Print number of lines in current `region`.

`C-x =` `what-cursor-position`
Print character code of character after `point`, character position and column of `point`. Handy to identify Unicode characters.

⟦`M-x`⟧ `describe-char`
Describe the character at point. Opens a window containing extensive information about the character.

`C-x l` `count-lines-page`
display the number of lines in the buffer as well as the number of lines before and after mark

COMMAND ARGUMENTS

You can give any Emacs command a *numeric argument* (also called a *prefix argument*). Some commands interpret the argument as a repetition count. For example, `C-f` with an argument of 10 moves point forward by ten characters instead of one. With these commands, the absence of an argument is equivalent to an argument of 1.

If your keyboard has a META key, the easiest way to specify a numeric argument is to type `M-1-0 C-f`.

Another way of specifying an argument is to use the `C-u universal-argument` command followed by the digits of the argument. with `C-u` you can type the argument digits without holding down META.

`C-u` followed by a character which is neither a digit nor a minus sign has the special meaning of "multiply by four." `C-u` twice multiplies it by sixteen, etc. This is a good way to "move fast," since `C-u C-u C-f` moves forward about a fifth of a line. `C-u C-u C-o` makes a lot of blank lines. `C-u C-k` kills four lines.

THE REGION

⟦`M-x`⟧ `transient-mark-mode`
Turns on highlighting of `region` when using a windowing system. In this mode, a `region` "lasts" only temporarily. Handy for visualizing where the `region` is.

`C-`SPC `set-mark-command`
Set the `mark` to where `point` is.

`C-x C-x` `exchange-point-and-mark`
Interchange `point` and `mark`.

`C-w` `kill-region`
Kill the contents of `region`.

YANKING

"Yanking" means inserting previously killed text. Note that text removed by DEL (`delete-backward-character`) is *deleted* text, not *killed* text, and therefore is can not be inserted by yanking.

`C-y` `yank`
Yank the last killed text.

`M-y` `yank-pop`
Replace text just yanked with an earlier batch of killed text.

`C-M-y` `append-next-kill`
Append next kill to last batch of killed text.

Searching and Replacing continues in the next column…

Searching and Replacing continues from the previous column…

SEARCHING AND REPLACING

The principal search command is *incremental*; it begins to search before you have finished typing the search string.When you have typed enough characters to identify the place you want, you can stop. You may or may not need to terminate the command with RET. If you make a mistake in typing the search string, you can cancel characters with DEL.

Incremental searches generally ignore the case of the text they are searching through, *if* you specify the text in lower case. An upper-case letter anywhere in the incremental search string makes the search case-sensitive. There are also non-incremental search commands.

Incremental Search

`C-s` `isearch-forward`
Incremental search forward.

`C-r` `isearch-backward`
Incremental search backward.

Non-Incremental Search

`C-s` `isearch-forward`
RET *string* RET
Search forward for *string*.

`C-r` `isearch-forward`
RET *string* RET
Search backward for *string*.

Regular Expression Search

A *regular expression* (*regexp* for short) is a pattern that denotes a class of alternative strings to match. If you type a SPC in incremental regexp search, it matches any sequence of whitespace characters, including newlines. If you want to match just a space, type `C-q` SPC.

Non-incremental regexp search is done by `re-search-forward` and `re-search-backward`. You can invoke these via `M-X`, bind them to keys, or invoke them by way of incremental regexp search with `C-M-s` RET and `C-M-r` RET (see below).

`C-M-s` `isearch-forward-regexp`
regexp
Reads a regexp search string incrementally, searching forward. If no *regexp* is provided and RET is pressed, prompt for another *regexp* which is used in performing a non-incremental regexp search.

`C-M-r` `isearch-backward-regexp`
regexp
Reads a regexp search string incrementally, searching backward. If no *regexp* is provided and RET is pressed, prompt for another *regexp* which is used in performing a non-incremental regexp search.

Replacement Commands

⟦`M-x`⟧ `replace-string`
string RET *newstring* RET
Replace every occurrence of *string* with *newstring*.

⟦`M-x`⟧ `replace-regexp`
regexp RET *newstring* RET
Replace every match for *regexp* with *newstring*.

`M-%` `query-replace`
string RET *newstring* RET
Replace some occurrences of *string* with *newstring*.

Other Search and Loop Commands

⟦`M-x`⟧ `occur`
regexp RET
Display a list showing each line in buffer that contains a match for *regexp*.

FIXING TYPOS

Killing Mistakes

DEL `delete-backward-character`
Delete last character.

`M-`DEL `backward-kill-word`
Kill last word.

`C-x` DEL `backward-kill-sentence`

Fixing Typos continues in the next column…

Fixing Typos continues from the previous column…

Kill to beginning of sentence.

Transposing Text

`C-t` `transpose-chars`
Transpose the characters on either side of `point`.

`M-t` `transpose-words`
Transpose two words.

`C-x C-t` `transpose-lines`
Transpose two lines.

Case Conversion

`C-x C-l` `downcase-region`
Convert `region` to lowercase.

`C-x C-u` `upcase-region`
Convert `region` to uppercase.

Check and Correct Spelling

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

`M-` TAB `ispell-complete-word`
Complete the word before point using spelling dictionary.

`[M-x]` `ispell-buffer`
Check and correct spelling of buffer.

`[M-x]` `ispell-region`
Check and correct spelling of `region`.

INDENTING

TAB `indent-for-tab-command`
Indent the current line or Region, or insert a tab, as appropriate. The function called to actually indent the line or insert a tab is given by the variable `indent-line-function`.

If a prefix argument is given, after this function indents the current line or inserts a tab, it also rigidly indents the entire balanced expression which starts at the beginning of the current line, to reflect the current line's indentation.

If `transient-mark-mode` is turned on and the region is active, this function instead calls `indent-region`. In this case, any prefix argument is ignored.

`C-M-\` `indent-region`
Indent each non blank line in the region. A numeric prefix argument specifies a column: indent each line to that column. With no prefix argument, the command chooses one of three methods and indents all the lines with it. See the function documentation for more details.

`C-x` TAB `indent-rigidly`
Indent all lines starting in `region`.

If called interactively with no prefix argument, activate a transient mode in which the indentation can be adjusted interactively by typing ← and →. Typing any other key deactivates the transient mode.

If called interactively with prefix argument `N`, indent all lines in `region` to column `N`.

`[M-x]` `indent-code-rigidly`
Indent all lines of code, starting in the region, sideways by ARG columns. Does not affect lines starting inside comments or strings, assuming that the start of the region is not inside them.

FILLING

`[M-x]` `auto-fill-mode`
Enable or disable Auto Fill mode. When enabled, lines are broken automatically at spaces when they get longer than the desired width. The desired width is set in `fill-column`.

`M-q` `fill-paragraph`
Fill paragraph at or after point. If the Transient Mark mode is enabled and the `mark` is active, call `fill-region` to fill each of the paragraphs in the active region, instead of just filling the current paragraph.

***Filling** continues in the next column…*

***Filling** continues from the previous column…*

`C-x f` `set-fill-column`
If `ARG` was provided, set `fill-column` to `ARG`. Otherwise, set `fill-column` to the current column.

RECTANGLES

The rectangle commands operate on rectangular areas of text: all characters between a certain pair of columns, in a certain range of lines. When you specify a rectangle for a command to work on, you do it by putting `mark` at one corner and `point` at opposite corner. The rectangle thus specified is the `region-rectangle` because you control it in the same way that you control a `region`. If `point` and `mark` are in the same column, the `region-rectangle` they define is empty; however, the `region` they define will be empty only if `point` and `mark` are at the same location.

`[M-x]` `clear-rectangle`
Clear the `region-rectangle` by replacing its contents with spaces.

`C-x r k` `kill-rectangle`
Kill the text of the `region-rectangle`, saving its contents as the `kill-rectangle`.

`C-x r d` `kill-rectangle`
Delete the text of the `region-rectangle`.

`C-x r y` `yank-rectangle`
Yank the last killed rectangle with its upper left corner at `point`.

`C-x r o` `open-rectangle`
Insert blank space to fill the space of `region-rectangle`. This pushes the previous contents of the `region-rectangle` rightward.

`[M-x]` `string-rectangle` RET *string* RET
Insert *string* on each line of `region-rectangle`.

MACROS

An intro/summary would be helpful here.

`C-x (` `start-kbd-macro`

`C-x (` `end-kbd-macro`

`C-x e` `call-last-kbd-macro`

`C-/` `undo`
Undo any changes you made during failed attempt at macro definition.

`C-g` `keyboard-quit`
abort macro definition

`C-x e` `call-last-kbd-macro`

RET `newline`
Use RET to exit incremental search that is embedded within macro

COMMON TO MANY PROGRAMMING MODES

`M-;` `comment-dwim`
Call the appropriate comment command (dwim = Do What I Mean) for the subject lines. If the `region` is set, "comment out" the lines in `region`, unless `region` contains only comments, in which case the lines in `region` are "uncommented". If `region` is not set, "comment out" or "uncomment" the current line, as appropriate.

PYTHON MODE

`C-c <` `python-indent-shift-left`
Shift lines contained in `region` by *COUNT* columns to the left. *COUNT* defaults to `python-indent-offset`. If `region`isn't active, the current line is shifted.

***Python Mode** continues in the next column…*

Python Mode continues from the previous column…

`C-c >` `python-indent-shift-right`
Shift lines contained in `region` by *COUNT* columns to the right. *COUNT* defaults to `python-indent-offset`. If `region`isn't active, the current line is shifted.

REST MODE

Adornment

`C-c C-a C-a` `rst-adjust`
Cycle through all possible adornments for current content.

`C-c C-a C-d` `rst-display-adornments-hierarchy`
Display current hierarchy of adornments in document.

`C-c C-a C-s` `rst-straighten-adornments`
Homogenize all adornments in document.

Lists

`C-c C-l C-b` `rst-bullet-list-region`
Convert each block of text in region to bullet list.

`C-c C-l C-e` `rst-enumerate-region`
convert each block of text in region to enumerated list

`C-c C-l C-c` `rst-convert-bullets-to-enumeration`

`C-c C-l C-i` `rst-insert-list`
insert a list item, prompt for type and optionally number

`C-c C-l C-s` `rst-straighten-list`
Homogenize list to use identical bullet types.

ELISP

An intro/summary would be helpful here.

`C-u M-!` `eval-expression`
expression RET
Read an elisp expression, evaluate it, and display result in echo area.

`M-n` `forward-list`
During `eval-expression`, move to next expression in history.

`M-p` `previous-list`
During `eval-expression`, move to previous expression in history.

`C-x M-:` `repeat-complex-command`
last-command RET
Edit and re-eval the last complex command, or the ARGth from the last.

`C-x C-e` `eval-last-sexp`
Eval the s-expression before `point`, display value in echo area. With prefix argument, place output in current buffer.

SHELL

An intro/summary would be helpful here.

`M-:` `shell-command`
command RET
Run a shell command and insert its output in the buffer.

`C-x M-|` `shell-command-on-region`
command RET
Run a shell command on a region, replacing the region with the output of the shell command.

USEFUL UNBOUND FUNCTIONS

`-` `list-colors-display`
List all available colors, with samples, in a new buffer.

`-` `customize-themes`
Display and select available themes.

TODO

TODO continues in the next column…

TODO continues from the previous column…

- Add content to Python Mode, espec. shell interaction.
- Consider a .emacs section.
- Cleanup lengths and other defs.
- Everything from Macros on is just early first draft. These need work.
- Interactive shells
- Would be nice if all rows in a command's documentation were not split across pages
- Would be nice if column headers were always accurate
- Lots of extra space a pagebottoms.
- Useful Unbound Functions is lame.
- Commands in each section could bear some reorganization to better highlight commonality (e.g., the symmetry between `C-k` and `M-k`.).

EMACS LISP - *ELISP*

How to change the default binding for `M-z` to `zap-up-to-char` in `.emacs`:

```
;; Replace zap-to-char with zap-up-to-char
(autoload 'zap-up-to-char "misc"
 "Kill up to, but not including ARGth occurrence of CHAR.

  \(fn arg char)" 'interactive)
(global-set-key "\M-z" 'zap-up-to-char))
```

Kenneth East, Austin, Texas December 22, 2015