Emacs Lisp Cookbook

This page contains snippets of code that demonstrate basic Emacs Lisp programming operations in the spirit of O'Reilly's Cookbook series of books. For every task addressed, a worked-out solution is presented as a short, focused, directly usable piece of code.

All this stuff can be found elsewhere, but it is scattered about in libraries, manuals, etc. It would be helpful to have it here in one spot.

These recipes should be pastable into the ***scratch*** buffer so that users can hit **C-j** and evaluate them step by step.

Contents

Strings

Note: the Melpa package s.el (https://github.com/magnars/s.el) provides a modern string manipulation library.

The empty string (zero-length string, null string, ...):

```
(zerop (string-match "" "")) ;; O(n)
==> t

(string-equal "" "") ;; O(n)?
==> t

(equal "" "") ;; O(n)?
==> t

(zerop (length "")) ;; O(1)
==> t

(eq "" "") ;; O(1)
==> t
```

As a space and performance optimization, Emacs keeps an intern-ed copy of the empty string as a single object

```
(eq "" (purecopy ""))
==> t

(eq "" (propertize "" 'face 'italic))
==> t
```

Strings vs buffer content

While it is quite common in other programming languages to work on strings contained in variables in Emacs it is even more idiomatic to work on strings in buffers. That's why the following contains examples of both.

Processing characters

For string manipulation, you must use s.el (https://github.com/magnars/s.el#s-trim-s).

Reversing a string:

```
;; with s.el
(s-reverse "ab xyz") ;; => "zyx ba"
;; otherwise
(string-to-list "foo")
==> (102 111 111)
```

```
(reverse (string-to-list "foo"))
==> (111 111 102)
(apply 'string (reverse (string-to-list "foo")))
==> "oof"
```

See CharacterProcessing and StringModification. See tr for an example if you sometimes need to mix strings and characters.

Looking at characters in buffers:

```
(with-temp-buffer
  (insert "abcdefg")
  (goto-char (point-min))
  (while (not (= (char-after) ?b))
    (forward-char))
  (point))
==> 2
```

Trim whitespace

Trim whitespace from the end of a string:

With s.el, see tweak whitespace (https://github.com/magnars/s.el#tweak-whitespace).

```
;; with s.el
(s-trim " this") ;; => "this"
;; or
(setq test-str "abcdefg  ")
(when (string-match "[ \t]*$" test-str)
  (message (concat "[" (replace-match "" nil nil test-str) "]")))
```

Trim whitespace from a string with a Perl-like chomp function:

```
(defun chomp (str)
  "Chomp leading and tailing whitespace from STR."
  (while (string-match "\\`\n+\\|^\\s-+\\|\\s-+$\\|\n+\\'"
                       str)
    (setq str (replace-match "" t t str)))
  str)
```

Splitting strings

With s.el (https://github.com/magnars/s.el), see **s-split**, **s-truncate** and many more.

The 'split-string' function is defined in 'subr.el' as

```
(defun split-string (string &optional separators omit-nulls)
 ...)
```

where 'separators' is a regular expression describing where to split the string. 'separators' defaults to white-space characters (spaces, form feeds, tabs, newlines, carriage returns, and vertical tabs). If 'omit-nulls' is set as 't' then zero-length strings are deleted from output.

```
(split-string "1 thing 2 say 3 words 4 you" "[1-9]")
==> ("" " thing " " say " " words " " you")
```

Omitting nulls:

```
(split-string "1 thing 2 say 3 words 4 you" "[1-9]" t)
(" thing " " say " " words " " you")
```

Joining strings

With s.el, use **s-join**.

Or use **mapconcat** to join a list into a string using a separator ("glue") between elements in the string.

Example:

```
(s-join "+" '("abc" "def" "ghi")) ;; => "abc+def+ghi"
;; or
(mapconcat 'identity '("" "home" "alex " "elisp" "erc") "/")
==> "/home/alex /elisp/erc"
```

Serialization

The basic idea is to convert forms to strings with `prin1-to-string' and convert it back from a string with `read'.

```
(length (read (prin1-to-string (make-list 1000000 '(x)))))
==> 1000000

(read (prin1-to-string "Hello World!"))
==> "Hello World!"
```

This only works in the simplest cases. Unfortunately, this doesn't work for all Emacs data types for programming or the editor.

```
(read (prin1-to-string (make-hash-table))) ;; Error before Emacs 23.
==> #s(hash-table size 65 test eql rehash-size 1.5 [...] data ())

(read (prin1-to-string (current-buffer)))
==> Lisp error: (invalid-read-syntax "#")
```

Formatting

Killing text

As the Emacs Lisp Manual says, "Most of the kill commands are primarily for interactive use [...] When you need to delete text for internal purposes within a Lisp function, you should normally use deletion functions, so as not to disturb the kill ring contents."

The following mimic the `kill-' commands but without disturbing the kill ring.

Delete region

The Lisp equivalent of `kill-region' (`C-w') but without kill ring side effects::

```
(delete-region (region-beginning) (region-end))
```

According to the ElispManual, "Few programs need to use the `region-beginning' and `region-end' functions." This is because Lisp code should not rely on nor "alter the mark unless altering the mark is part of the user-level functionality of the command. (And, in that case, this effect should be documented.) To remember a location for internal use in the Lisp program, store it in a Lisp variable. For example: [...]"

```
(let ((beg (point)))
  (forward-line 1)
  (delete-region beg (point)))
```

Delete line

The equivalent of `kill-line' (`C-k') but without kill ring side effects:

```
(let ((beg (point)))
  (forward-line 1)
  (forward-char -1)
  (delete-region beg (point)))
```

Alternatively, replacing the `let' with `save-excursion'.

```
(delete-region (point)
               (save-excursion
                 (forward-line 1)
                 (forward-char -1)
                 (point)))
```

Or simplest of all,

```
(delete-region (point) (line-end-position))
```

The examples with `forward-line' are shown because the paradigm is used later, see below.

Delete line backwards

The equivalent of killing the line backwards (`C-0 C-k') but without kill ring side effects:

```
(let ((beg (point)))
  (forward-line 0)
  (delete-region (point) beg))
```

Alternatively, replacing the `let' with `save-excursion'.

```
(delete-region (save-excursion
                 (forward-line 0)
                 (point))
               (point))
```

Or simplest of all,

```
(delete-region (line-beginning-position) (point))
```

Delete line to next line

The equivalent of killing the line and the newline (`C-1 C-k') but without kill ring side effects:

```
(let ((beg (point)))
  (forward-line 1)
  (delete-region beg (point)))
```

Alternatively, replacing the `let' with `save-excursion'.

```
(delete-region (point)
               (save-excursion
                 (forward-line 1)
                 (point)))
```

Delete whole line

The equivalent of `kill-whole-line' (`C-S-DEL') but without kill ring side effects:

```
(let ((beg (progn (forward-line 0)
                  (point))))
  (forward-line 1)
  (delete-region beg (point)))
```

Alternatively, replacing the `let' with `save-excursion'.

```
(delete-region (save-excursion
                 (forward-line 0)
                 (point))
               (save-excursion
                 (forward-line 1)
                 (point)))
```

Or simplest of all,

```
(delete-region (line-beginning-position) (line-end-position))
```

Delete word

The equivalent of `kill-word' (`M-d') but without kill ring side effects:

```
(let ((beg (point)))
  (forward-word 1)
  (delete-region beg (point)))
```

Alternatively, replacing the `let' with `save-excursion'.

```
(delete-region (point)
               (save-excursion
                 (forward-word 1)
                 (point)))
```

Delete sentence

The equivalent of `kill-sentence' (`M-k') but without kill ring side effects:

```
(let ((beg (point)))
  (forward-sentence 1)
  (delete-region beg (point)))
```

Alternatively, replacing the `let' with `save-excursion'.

```
  (delete-region (point)
                 (save-excursion
```

```
                    (forward-sentence 1)
                    (point)))
```

Search and Replace

Searching and replacing text is a fundamental editing need. Emacs has separate facilities for both interactive and scripted search and replace.

Interactive Use

The **replace-regexp** function provides a way to replace text interactively. This function supports embedded emacs lisp statements in the second arguement (the replacement expression). By default **replace-regexp** replaces every match after the cursor location until it reaches the end of the buffer. A more useful method is to mark a region for replacement.

For example: calling

`M-x replace-regexp RET \([A-Z]\) RET \,(downcase \1)`

on the marked region **AABBCC** will convert it to **aabbcc**. The first arguement is a regular expression matching any capital letter and saving it as the first match, while the **\,** indicates embedded emacs lisp code (which calls the `downcase` function on the matched pattern).

Scripted Use

A cleaner solution while scripting is to combine **search-forward-regexp** with **replace-match**. Every time the search is successful, the results are implicitly saved in to **match-string**. This short function replaces every pattern in a marked region with a new string drawn from its components:

```
(defun camelCase-to_underscores (start end)
  "Convert any string matching something like aBc to a_bc"
  (interactive "r")
  (save-restriction
    (narrow-to-region start end)
    (goto-char 1)
    (let ((case-fold-search nil))
      (while (search-forward-regexp "\\([a-z]\\)\\([A-Z]\\)\\([a-z]\\)" nil t)
        (replace-match (concat (match-string 1)
                               "_"
                               (downcase (match-string 2))
                               (match-string 3))
                       t nil)))))
```

Note: to toggle between underscore, CamelCase and uppercase styles, you can use the string-inflection (http://melpa.milkbox.net/#/string-inflection) package.

Note: to search for text on buffers, have a look to the m-buffer (https://github.com/phillord/m-buffer-el) library. For example, to get all strings matching a regexp in the current buffer, do

```
(m-buffer-match-string-no-properties
    (m-buffer-match (current-buffer) "[a-z]*"))
```

Dates and times

Get today's date

```
(format-time-string "%d %B %Y")
```

or

```
(eshell/date)
```

Conversions

Read a date from a string.

```
(let ((time (date-to-time "Tue, 27-Sep-83 12:35:59 EST")))
  (set-time-zone-rule t) ;; Use Universal time.
  (prog1 (format-time-string "%Y-%m-%d %T UTC" time)
    (set-time-zone-rule nil))) ;; Reset to default time zone.
==> "1983-09-27 17:35:59 UTC"
```

Decode a time object.

```
(decode-time (date-to-time "Tue, 27-Sep-83 12:35:59 EST"))
==> (59 35 13 27 9 1983 2 t -14400)
```

Get the seconds from the unix epoch.

```
(let ((time (date-to-time "13 Feb 2009 23:31:30 UTC")))
  (float-time time))
==> 1234585890.0
```

Find the date for seconds from the unix epoch.

```
(format-time-string "%Y-%m-%d %T UTC" (seconds-to-time 1234585890))
==> "2009-02-13 23:31:30 UTC"
```

Find the date 30 seconds in the future.

```
(format-time-string "%Y-%m-%d %T UTC" (time-add (current-time)
                                               (seconds-to-time 30)))
==> "2012-02-13 10:07:11 UTC"
```

Formatting elapsed time in years, days, hours, minutes and seconds.

```
(format-seconds "%Y %D %h:%m:%s" (1- (* 367 24 3600)))
==> "1 year 1 day 23:59:59"
```

Find the days between two dates.

```
(let ((days1 (time-to-days (date-to-time "Tue, 27-Sep-83 12:35:59 EST")))
      (days2 (time-to-days (date-to-time "2009-02-13 23:31:30 UTC"))))
  (- days2 days1))
==> 9271
```

Getting the day in the year.

```
(time-to-day-in-year (current-time))
==> 44
```

Build a date based on the day of the year.

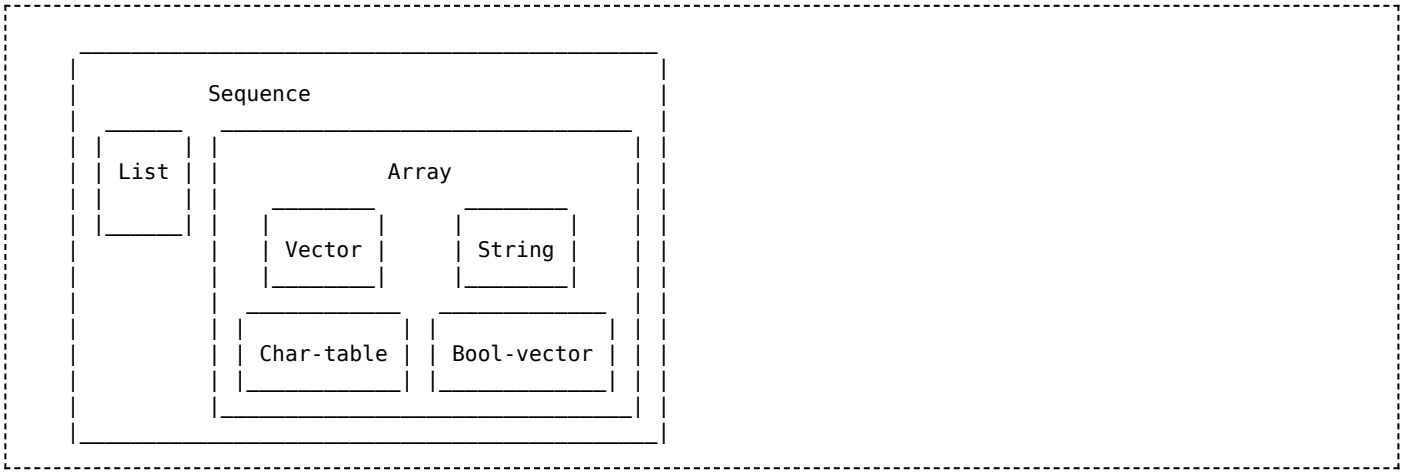
```
(format-time-string "%j"
                    (encode-time 0 0 0 44 1 2012))
==> "044"
```

Timers

TODO

Sequences

Datatypes used to represent sequences of things:



Lists

List basics are explained on ListStructure. Lists can shrink and grow, but access to elements towards the end of the list is slow if the list is long.

Use `cons' to append a new element to the front of a list. Use `nth' to access an element of the list.

```
(let ((words '("fight" "foo" "for" "food!")))
  (when (string= "foo" (nth 1 words))
    (setq words (cons "bar" words)))
  words)
==> ("bar" "fight" "foo" "for" "food!")
```

See ListModification for more ways of changing a list.

Iteration:

```
(let ((result))
  (dolist (word '("fight" "foo" "for" "food!"))
    (when (string-match "o" word)
      (setq result (cons word result))))
  (nreverse result))
==> ("foo" "for" "food!")
```

Note how `cons' adds an element to the front of the list, so that usually the list has to be reversed after the loop. `nreverse' is particularly efficient because it does this destructively by swiveling pointers around. See DestructiveOperations for more about this.

Copying:

Use `copy-sequence' to make a copy of a list that won't change the elements of the original.

```
    (let* ((orig '((1 2) (3 4)))
           (copy (copy-sequence orig)))
      (setcdr copy '((5 6)))
      (list orig copy))
    ==> (((1 2) (3 4)) ((1 2) (5 6)))
```

However, the elements in the copy are still from the original.

```
    (let* ((orig '((1 2) (3 4)))
           (copy (copy-sequence orig)))
      (setcdr (cadr copy) '(0))
      (list orig copy))
    ==> (((1 2) (3 0)) ((1 2) (3 0)))
```

The function `copy-tree' is the recursive version of `copy-sequence'.

```
    (let* ((orig '((1 2) (3 4)))
           (copy (copy-tree orig)))
      (setcdr (cadr copy) '(0))
      (list orig copy))
    ==> (((1 2) (3 4)) ((1 2) (3 0)))
```

Filtering:

Emacs Lisp doesn't come with a `filter' function to keep elements that satisfy a conditional and excise the elements that do not satisfy it. One can use `mapcar' to iterate over a list with a conditional, and then use `delq' to remove the `nil' values.

```
  (defun my-filter (condp lst)
    (delq nil
          (mapcar (lambda (x) (and (funcall condp x) x)) lst)))
```

Therefore,

```
  (my-filter 'identity my-list)
```

is equivalent to

```
  (delq nil my-list)
```

For example:

```
  (let ((num-list '(1 'a 2 "nil" 3 nil 4)))
    (my-filter 'numberp num-list))
  ==> (1 2 3 4)
```

Actually the package cl-seq contains the functions `remove-if' and `remove-if-not'. The latter can be used instead of `my-filter'.

```
  (let ((num-list '(1 'a 2 "nil" 3 nil 4)))
    (remove-if-not 'numberp num-list))
  ==> (1 2 3 4)

  (let ((num-list '(1 'a 2 "nil" 3 nil 4)))
    (remove-if 'numberp num-list))
  ==> ((quote a) "nil" nil)
```

As an example here is the quick sort algorithm:

```
(defun quicksort (lst)
  "Implement the quicksort algorithm."
  (if (null lst) nil
    (let* ((spl (car lst))
           (rst (cdr lst))
           (smalp (lambda (x)
                  (< x spl))))
      (append (quicksort (remove-if-not smalp rst))
              (list spl)
              (quicksort (remove-if smalp rst))))))

(quicksort '(5 7 1 3 -9 8 7 -4 0))
==> (-9 -4 0 1 3 5 7 7 8)
```

Tranposing:

Convert multiple lists into a list

```
((lambda (&rest args)
   (mapcar (lambda (n)
             (delq nil (mapcar (lambda (arg) (nth n arg)) args)))
           (number-sequence 0 (1- (apply 'max (mapcar 'length args))))))
 '(1 2 3) '(a b c) '(A B C))
 ==> ((1 a A) (2 b B) (3 c C))
```

A more concise version is possible with the the higher-arity version of mapcar available with the `cl' library.

```
((lambda (&rest args)
   (apply (function mapcar*) (function list) args))
 '(1 2 3) '(a b c) '(A B C))
==> ((1 a A) (2 b B) (3 c C))
```

Searching:

Simply checking for existence of a value in a list can be done with `member' or `memq'.

```
(let ((words '("fight" "foo" "for" "food!")))
  (car (member "for" words)))
==> "for"

(let ((re "\\wo\\b")
      (words '("fight" "foo" "for" "food!")))
  (consp (memq t
           (mapcar (lambda (s) (numberp (string-match re s))) words))))
==> t
```

In the latter, a more efficient algorithm would use a loop (a non-local exit).

Vectors

Vectors are fixed in size but elements can be accessed in constant time.

```
(let ((words ["fight" "foo" "for" "food!"]))
  (when (string= "foo" (aref words 1))
    (aset words 1 "bar"))
  words)
==> ["fight" "bar" "for" "food!"]
```

Hashes

Hashes map keys to values. In a way they are similar to alists, except they are more efficient for a large number of keys.

More info is available on the HashMap page.

Storing and retrieving keys and values

By default, hash tables use `eql' to compare keys. This is not appropriate for strings: ##(eql "alex" "alex")## ==> nil. Thus, use `equal' in these cases:

```
    (let ((nick-table (make-hash-table :test 'equal)))
      (puthash "kensanata" "Alex Schroeder" nick-table)
      (gethash "kensanata" nick-table))
    ==> "Alex Schroeder"
```

Iterate:

```
    (let ((nick-table (make-hash-table :test 'equal))
          nicks)
      (puthash "kensanata" "Alex Schroeder" nick-table)
      (puthash "e1f" "Luis Fernandes" nick-table)
      (puthash "pjb" "Pascal J. Bourguignon" nick-table)
      (maphash (lambda (nick real-name)
                 (setq nicks (cons nick nicks)))
               nick-table)
      nicks)
      ==> ("pjb" "e1f" "kensanata")
```

Sorting keys

Use `maphash' to build up a list of keys, sort it, and then loop through the list:

```
    (let ((nick-table (make-hash-table :test 'equal))
          nicks)
      (puthash "kensanata" "Alex Schroeder" nick-table)
      (puthash "e1f" "Luis Fernandes" nick-table)
      (puthash "pjb" "Pascal J. Bourguignon" nick-table)
      (maphash (lambda (nick real-name)
                 (setq nicks (cons nick nicks)))
               nick-table)
      (mapcar (lambda (nick)
                (concat nick " => " (gethash nick nick-table)))
              (sort nicks 'string<)))
      ==> ("e1f => Luis Fernandes"
           "kensanata => Alex Schroeder"
           "pjb => Pascal J. Bourguignon")
```

Files

Read

Processing a file is usually done with a temporary buffer:

```
  (defun process-file (file)
    "Read the contents of a file into a temp buffer and then do
  something there."
    (when (file-readable-p file)
      (with-temp-buffer
        (insert-file-contents file)
        (goto-char (point-min))
        (while (not (eobp))
        ;; do something here with buffer content
          (forward-line)))))
```

On the chance that a buffer may already be actively visiting the file, consider using `find-file-noselect'

```
(defun file-string (file)
  "Read the contents of a file and return as a string."
  (with-current-buffer (find-file-noselect file)
    (buffer-string)))
```

Write

To write something to a file you can create a temporary buffer, insert the things to write there and write the buffer contents to a file. The following example read a string and a filename (with completion, but doesn't need to exist, see InteractiveCodeChar F) and write the string to that file.

```
(defun write-string-to-file (string file)
  (interactive "sEnter the string: \nFFile to save to: ")
  (with-temp-buffer
    (insert string)
    (when (file-writable-p file)
      (write-region (point-min)
                    (point-max)
                    file))))
```

Searching

If you don't have grep, then you may need to write some Lisp which can find a match in a file.

```
;; Visit file unless its already open.
(with-current-buffer (find-file-noselect "~/.emacs")
  (save-excursion ;; Don't change location of point.
    (goto-char (point-min)) ;; From the beginning...
    (if (re-search-forward ".*load-path.*" nil t 1)
        (match-string-no-properties 0)
      (error "Search failed"))))
==> "(add-to-list 'load-path \"/usr/share/emacs/site-lisp/\")"
```

Filter

Locking

Stat

An interface to the kernel's stat(2) is provided by the function file-attributes. The way times are represented may be a bit unexpected, though.

Deleting

```
(if (file-exists-p filename)
    (delete-file filename))
```

Copy, move and rename

Directories

Traversing

```
    (defun walk-path (dir action)
       "walk DIR executing ACTION with (dir file)"
       (cond ((file-directory-p dir)
              (or (char-equal ?/ (aref dir(1- (length dir))))
                  (setq dir (file-name-as-directory dir)))
              (let ((lst (directory-files dir nil nil t))
                    fullname file)
                (while lst
                  (setq file (car lst))
                  (setq lst (cdr lst))
                  (cond ((member file '("." ".."))
                         (t
                          (and (funcall action dir file)
                               (setq fullname (concat dir file))
                               (file-directory-p fullname)
                               (walk-path fullname action)))))))))
             (t
              (funcall action
                       (file-name-directory dir)
                       (file-name-nondirectory dir)))))

    (defun walk-path-visitor (dir file)
       "Called by walk-path for each file found"
       (message (concat  dir file)))

    (walk-path "~/" 'walk-path-visitor)
```

Note: see also *f-entries* of f.el (https://github.com/rejeep/f.el#f-entries-path-optional-fn-recursive) to find all files and directories in a path.

Path splitting

Splitting the path can be done with `split-string' and with the slash. Previously, Emacs would determine the character separating directory names with `directory-sep-char'. However, the variable is obselete with Emacs 21.1.

```
 (split-string default-directory "/")
 ==> ("" "usr" "share" "emacs" "22.2" "lisp" "")
```

For splitting a path variable, Emacs already has the `parse-colon-path' function.

```
 (parse-colon-path (getenv "PATH"))
 ==> ("/usr/lib/qt-3.3/bin/" "/usr/kerberos/bin/" "/usr/local/bin/"
 "/usr/bin/" "/bin/" "/usr/local/sbin/" "/usr/sbin/" "/sbin/")
```

Processes

Running a program

Run a command without caring about its output.

```
   (async-shell-command "emacs")
```

Run a command and put its output in the current buffer.

```
(shell-command "seq 8 12 | sort" t)
10
11
12
8
9
```

Run a command and put its output in a new buffer.

```
(shell-command "seq 8 12 | sort"
               (get-buffer-create "*Standard output*"))
```

Run a command return its output as a string.

```
(shell-command-to-string "seq 8 12 | sort")
```

XEmacs also comes with `exec-to-string'.

Handling signals

Sockets

Tcp client

Tcp server

Perhaps EmacsEchoServer and EmacsDaytimeServer can be useful here.

Keyboard events

- Call function bound to key

```
(funcall (key-binding (kbd "M-TAB")))
```

or

```
(call-interactively (key-binding (kbd "M-TAB")))
```