Skip to content

Follow:
RSS
Email
Twitter

Mastering Emacs
a blog about mastering the world's best text editor

- Home
- About
- Effective Editing
- Reading Guide
- All Articles
- For Beginners
- Quick Tips
- Tutorials

Tags

autocomplete, editing, elisp, eval, ielm, inspect, scratch

Evaluating Elisp in Emacs

by mickey on November 29th, 2010
Home > All Articles > Evaluating Elisp in Emacs

There are several ways of evaluating elisp code in Emacs, and picking the right approach will help you get your job done faster and more efficiently. If you're new to elisp you will quickly realize that Emacs has many shortcuts and features that makes writing, inspecting or debugging elisp code very easy.

Evaluating Chunks of Code

If you have a body of code that you want to evaluate, this approach is by far the simplest. To quickly evaluate any amount of code — from one to many *forms*, where a *form* is essentially something that Emacs can evaluate — simply yank the code into an empty buffer and run `M-x eval-buffer`.

If you want something less hamfisted you can tell Emacs to only evaluate the *region* by marking what you want to run and executing `M-x eval-region`.

Concerning Regions…

It goes without saying that I assume `transient-mark-mode` is active, which is it in newer Emacsen by default; but chances are it is enabled in your `.emacs` anyway, as *transient* makes the region visible on your screen, and not surprisingly that is what most people want.

If you are a disbeliever and prefer a simpler life without *transient* then you're familiar with narrowing and exchanging the point and the mark to get what you want anyway. I should mention, by the way, that the vanguard of Emacs conversatism, Richard Stallman himself, *does* use `transient-mark-mode`.

Evaluating Expressions

By S-Expression

This is where Emacs starts to shine, for it has special tools available to it to evaluate single *s-expressions*. The facilities that are in place in Emacs to do this depend on the major mode you are using (and, rarely, your mode's *syntax table*) but most modes still expose `C-x C-e`, which is bound to `eval-last-sexp`.

The `eval-last-sexp` command is very interesting, because it is one of few commands that are globally available to most modes, and it only works if you have the point at the **end** of an s-expression (*"sexp"*), which is anything enclosed by a pair of parentheses, like this: `(message "Hello, World!")|` where `|` is your point.

There are a couple of limitations that make `C-x C-e` a poor tool to use if you are writing and testing elisp code, because it cannot update variables declared with `defvar` or `defcustom`, and it goes with saying that having to move point to the end of the expression every time you want to eval the sexp is annoying as well.

By Form

Arguably the best way to evaluate a form is `eval-defun`, bound to `C-M-x`. This command does not suffer from the problems I mentioned before, as it is designed for a typical hacker's write-eval-test cycle. You can type `C-M-x` anywhere in a form, and it will evaluate the outer-most form, ensuring that you won't accidentally evaluate only a part of a larger form.

What makes it even better is its built-in support for *edebug*, Emacs' elisp debugger. Using the universal argument `C-u` Emacs will evaluate the form as normal, but enable debug instrumentation. The next time it is run it kicks off the debugger and lets you step through the code, with every evaluation showing its value (if any) in the echo area.

You can also use this functionality for exploratory purposes by enabling debugging on Emacs' own routines.

This is by far the best way of quickly re-evaluating code you've updated, and it should be muscle memory for any serious elisp hacker.

By Expression

Another quick way of evaluating an expression is to use `eval-expression`, bound to `M-:`. Evaluated expressions are printed to the echo area and the value prepended to the list variable `values`. It is important to note that as it takes *expressions* and not *sexps* or *forms* you can also print the value of any variable in Emacs — just be sure **not** to wrap the variable in parenthesis.

This command is particularly handy if you quickly want to change a setting or invoke an elisp function directly, as it evaluates code in the context of the buffer it was invoked from. If you use the universal argument `C-u`, the output of the expression will be inserted into the buffer. In a similar vein, you can tell Emacs to redo the previous command it did and let you edit the command beforehand by invoking `C-x M-:`. Another useful way of exploring Emacs.

One thing you cannot do is invoke multiple expressions; to get around this, you can wrap the expressions in the `progn` form, like so: `(progn (foo) (bar) (baz))`.

Interactive Evaluation

The Scratch Buffer

The Emacs `*scratch*` buffer comes with a few useful additions over the regular Emacs-Lisp mode that makes it useful for throw-away scripts and quick code evaluation. If you put point at the end of an sexp and type `C-j` Emacs will evaluate the expression and print the result straight into the buffer. *Note: If you use paredit, be aware that it rebinds the key!*

The scratch buffer is a poor substitute for a proper REPL, but thankfully Emacs does come with one of those too.

The Interactive Emacs-Lisp Mode

This is a fantastic, and hidden, gem in Emacs: a proper REPL. Type `M-x ielm` to run the elisp REPL. It comes with all the useful features you would expect from an interactive shell, like `*`, `**` and `***` to get the last three outputs from the shell. Multi-line commands are fully supported as well, of course.

Because `ielm` inherits from comint you get all the advantages that provides, like a command history and so on. You also get limited completion support with `TAB`, which is very nice.

Another useful feature is the concept of a *working buffer* – a buffer through which your changes are evaluated. If you type `C-c C-b` you can change ielm's working buffer to one of your choosing and then all the code you evaluate thereafter will be treated as if you executed it in the context of that buffer. This functionality comes in handy if you are dealing with buffer-local variables or changes that're specific to one buffer only. Very, very powerful.

Enabling Auto Complete in IELM

By default ielm is not supported in auto complete, but that is easily rectified. Add this to your .emacs:

```
(defun ielm-auto-complete ()
  "Enables `auto-complete' support in \\[ielm]."
  (setq ac-sources '(ac-source-functions
                     ac-source-variables
                     ac-source-features
                     ac-source-symbols
                     ac-source-words-in-same-mode-buffers))
  (add-to-list 'ac-modes 'inferior-emacs-lisp-mode)
  (auto-complete-mode 1))
(add-hook 'ielm-mode-hook 'ielm-auto-complete)
```

Honorable Mention: Eshell

You can evaluate elisp in Eshell directly from the prompt, with some limitations. I have covered this in greater detail in my article on Mastering Eshell.

Conclusion

That just about covers evaluating code in Emacs, from one-liners to entire buffers of code. If you're hacking elisp you want `C-M-x` for your everyday editing and quick debugging; if you're inspecting state or want to change something in Emacs real quick, use `M-:`; if you want to evaluate a buffer or region, use `eval-buffer` or `eval-region`; if you're exploring elisp and want an interactive environment, use `ielm`. The rest serve more specialized roles, but you cannot go wrong if you stick to these recommendations.

Share

From → All Articles, Tutorials

10 Comments →

1. Reynaldo permalink

   Great post!
   I didn't know about ielm nor C-M-x.
   Thanks

   Reply

2. Scott permalink

I use C-M-S-x to do what C-M-x does but for the the immediately enclosing form as opposed to the top-level one. I don't like going to the end and doing C-x C-e. I don't write much emacs lisp so I've only implemented this for slime.

```
(defun up-list+ ()
  (interactive)
  (if (in-string-p)
      (while (in-string-p)
        (forward-char))
    (up-list)))

(defun eval-parent-sexp ()
  "Cause sometimes you just want to eval just the immediate form. not the top level, but without going to the closing paren and evali
  (interactive)
  (save-excursion
    ;; get out of string if in it
    (dotimes (c (if (in-string-p) 2 1))
      (up-list+))
    (let ((cmd (key-binding (kbd "C-x C-e"))))
      (if (eq cmd 'slime-eval-last-expression)
          (funcall cmd)
        (funcall cmd '()))))))

(global-set-key (kbd "C-M-S-x") 'eval-parent-sexp)
```

Reply

3. 

Paul Hobbs permalink

It's worth mentioning that M-: is REALLY useful when doing keyboard macros.

Reply

- 

  Conor permalink

  Do you have an example macro or two where you use M-: ?

  I haven't used elisp much but have been using keyboard macros a lot lately.

  Just trying to get an idea of the kinds of things I could do with M-:

  Reply

4. 

Michael permalink

Hi Mickey, what do you mean by "useful features you would expect from an interactive shell, like *, ** and *** to get the last three outputs from the shell"? I never heard of this but it sounds interesting. What shell would that be? bash?

Reply

- 

  mickey permalink

  Oh, yes, perhaps I should clarify: the shell I am referring to is the ielm shell itself.

  Reply

5. 

Comrade Curious permalink

What is the difference between s-expression, form, and expression?

A form is any top-level s-expression?

"blah", 123, [1 2 3] are expressions that are not s-expressions?

Reply

- 

  mickey permalink

  Expressions here means that a variety of operations are tried in order to resolve your query to M-: — if you give it a string you'll get a string back as as strings and numbers evaulate to themselves. A variable will give you its value and a function its output.

  The difference between a form and an s-expression is pretty academic:

  A form is an s-expression, but an s-exp may not necessarily be a form; for instance, (+ 1 1) is a form but ("+" 1 1) is not as you cannot evaluate the string "+".

Reply

Trackbacks & Pingbacks

1. Tweets that mention Evaluating Elisp in Emacs | Mastering Emacs -- Topsy.com
2. Fun with Vimgolf 2: Sum your vimgolf rank | Mastering Emacs

Leave a Reply

Name: (required):

Email: (required):

Website:

Comment:

Note: XHTML is allowed. Your email address will **never** be published.

Subscribe to this comment feed via RSS

Submit Comment

- ## Tags

admin ansi-term autocomplete beginner buffer customization cygwin editing elisp eshell eval files guide ido ielm info inspect key bind mark multilingual navigation network news opinion productivity prompt python readability recentf regexp region repl scratch shell tabs term terminology tips tramp transient trash can twitter vimgolf whitespace workflow

- ## Pages

  - About
  - Effective Editing
  - Reading Guide

- ## Blogroll

  - Emacs on Reddit
  - Follow me on Twitter
  - Irreal's Emacs blog
  - The Emacs Wiki
  - WikEmacs

About

Hi, I'm Mickey and this is my blog about mastering Emacs. I've been using Emacs for eight years and I'd like to share with you all the things I've learned over the years.

The blog will cover all facets of Emacs and will be suitable for beginners and -- I hope -- experts alike.

- ## Categories

  - All Articles
  - For Beginners
  - Quick Tips
  - Tutorials

Search

Search