~PABLO

PABLOMURAD.COM



SPECIAL EDITION · EMACS AND ORG MODE

Nº 01

OCTOBER – NOVEMBER 2026

5 essays · emacs · tutorial · tools

A ZINE FROM PABLOMURAD.COM

CONTENTS

Nº 01
SPECIAL EDITION · EMACS AND ORG MODE
OCTOBER – NOVEMBER 2026

01 **Emacs for the curious: install it, survive it, keep it** 3
EMACS · LINUX · WINDOWS
Emacs is older than most of the people who use it. The first version showed up at MIT in 1976, as a set of macros for an editor called TECO, and GNU Emacs, the one we all mean today, came out in 1985.

02 **Org mode from zero: a complete, friendly guide** 14
EMACS · ORG-MODE · TUTORIAL
A complete, friendly guide to Org mode in Emacs: headings and folding, TODOs, dates and repeaters, the agenda, capture, links, tables with formulas, clocking, code blocks you can run, export to HTML and PDF, a starter config and a cheat sheet.

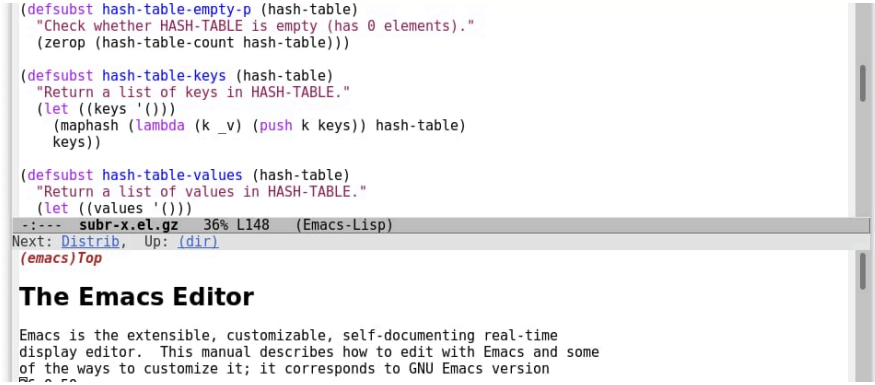
03 **Emacs beyond the basics, part I: feeds, gopher, the fediverse and mail** 31
EMACS · TUTORIAL · TOOLS
Emacs as a small operating system: read RSS with elfeed, browse gopher and Gemini with Elpher, follow the fediverse with mastodon.el, chat on IRC and Matrix, read the web in EWW and send mail. Keys, configs and a starter block.

04 **Emacs beyond the basics, part II: your first steps in Emacs Lisp** 38
EMACS · ELISP · LISP
A gentle first step into Emacs Lisp: read the syntax, write your own commands, bind keys, use hooks and advice, build a quick-notes command and learn to debug and read Emacs's own source. No programming background needed.

05 **Emacs beyond the basics, part III: a real programming setup** 47
EMACS · PROGRAMMING · TUTORIAL
Turn Emacs into a modern programming setup: projects, Eglot and language servers, tree-sitter, completion, Consult and Embark, compile and debug, Git without Magit, containers and a ready-to-paste config.

Emacs for the curious: install it, survive it, keep it

Emacs is older than most of the people who use it. The first version showed up at MIT in 1976, as a set of macros for an editor called TECO, and GNU Emacs, the one we all mean today, came out in 1985.



Emacs is older than most of the people who use it. The first version showed up at MIT in 1976, as a set of macros for an editor called TECO, and GNU Emacs, the one we all mean today, came out in 1985. Version 31.1 shipped in August 2026. Forty years of releases, and a plain install still opens in about a second, which is more than I can say for most chat apps.

People like to joke that Emacs is a great operating system lacking a decent text editor. It's funny because it's half true. Under the hood, Emacs is a Lisp machine that happens to edit text, and over the years people taught it to read email, manage tasks, browse files, run shells, talk to Git and play Tetris. You don't have to use any of that. But it's nice to know it's there.

This is the guide I'd give a friend who's curious and a little scared: how to install it on Linux and Windows, how to survive the first day, the

commands that matter, a small config to start with, and a pile of tricks that make people fall in love with the thing.

Installing on Linux

Every distribution has Emacs in its repositories, and that's the easiest way to get it. Pick your line:

```
sudo apt install emacs # Debian, Ubuntu, Mint
sudo dnf install emacs # Fedora
sudo pacman -S emacs # Arch, Manjaro
sudo zypper install emacs # openSUSE
guix package -i emacs # Guix
```

Check that it worked:

```
emacs --version
```

On Debian and Ubuntu there's also `emacs-nox`, a version without the graphical interface, perfect for servers. Even with the full version you can always run it inside the terminal with `emacs -nw` ("no window").

One thing to know: stable distributions often ship an older Emacs. That's fine for learning, everything in this post works on Emacs 29 and newer (with one exception I'll point out). If you want 31.1 right now, rolling distros like Arch and Fedora get it quickly, and Ubuntu has community Snap and Flatpak packages. You can also build it from source, which is less dramatic than it sounds.

Installing on Windows

Windows gets official builds too. Go to the GNU mirror (the download page at gnu.org/software/emacs sends you to the closest one), open the `windows` folder, pick the newest version and grab the file that ends in `-installer.exe`. Run it, click next a few times, and you'll have Emacs in your Start menu.

If you don't want to install anything, download the `.zip` from the same folder instead, unzip it anywhere and run `bin\runemacs.exe`. That works fine from a pen drive too.

And if you already live in MSYS2, it's one command:

```
pacman -S mingw-w64-x86_64-emacs
```

A Windows detail that confuses everybody: when Emacs says `~`, it usually doesn't mean `C:\Users\you`. Unless you've set a `HOME` environment

variable, it often points to `C:\Users\you\AppData\Roaming`. That's where your config folder will live. If you ever get lost, press `C-h v`, type `user-emacs-directory` and hit Enter. Emacs will tell you exactly where it's looking.

On a Mac, by the way, Homebrew has a cask for it (`brew install --cask emacs`). But this post is about Linux and Windows, so let's move on.

How to read Emacs keys

Before anything else, you need to read the notation, because every Emacs tutorial in the world uses it.

- `C-x` means hold **Ctrl** and press **x**.
- `M-x` means hold **Alt** and press **x**. The M is for "Meta," a key from old keyboards. If Alt doesn't work in your terminal, press **Esc** and then **x**.
- `C-x C-f` means Ctrl+x, then Ctrl+f. Two chords, one after the other.
- `C-x b` means Ctrl+x, let go, then just **b**.
- `RET` is Enter, `SPC` is the space bar.

And `M-x` is the most important key of all. It lets you run any command by name. Forgot a shortcut? `M-x`, type a piece of the name, Tab to complete, Enter. Everything in Emacs is a command, and every command has a name.

Surviving day one

Open Emacs and do this first:

```
C-h t
```

That's the built-in tutorial. It takes about half an hour, it's written by people who clearly enjoyed writing it, and it teaches you the basics by making you use them. It's the best Emacs tutorial ever written, and it's been sitting inside the program the whole time.

If you skip it anyway (I know), these are the keys that will keep you alive:

```
C-x C-f: open a file (or create a new one)
C-x C-s: save
C-x C-w: save as
C-/: undo
C-g: cancel whatever is happening
C-x C-c: quit Emacs
```

`C-g` deserves a moment. Whenever Emacs is waiting for something you didn't mean to ask for, a half-typed command, a weird prompt, a mode

you don't understand, `C-g` gets you out. Press it two or three times if you panic. It's the Esc key of Emacs, and you'll use it a lot in the first week.

Moving around

Arrow keys, Page Up and Home all work, so nobody is forcing you. But the Emacs keys keep your hands on the home row, and after a while they feel faster:

```
C-f / C-b: forward / back one character
M-f / M-b: forward / back one word
C-n / C-p: next / previous line
C-a / C-e: start / end of the line
C-v / M-v: page down / page up
M-< / M->: top / bottom of the file
M-g g: go to a line number
C-l: center the screen on the cursor (press again to move it to the top)
```

Notice the pattern: `C-` does something small, `M-` does the bigger version of the same thing. Character and word, line and sentence. Once you see it, a lot of the keys start to guess themselves.

Editing, copying and pasting

Emacs had copy and paste before Ctrl+C and Ctrl+V existed, so it uses its own words. Cutting is "killing," pasting is "yanking," and the clipboard is the "kill ring," because it remembers a whole history of things you cut.

```
C-SPC: start selecting (set the "mark"), then move the cursor
C-w: cut the selection
M-w: copy the selection
C-y: paste
M-y: right after C-y, cycle through older things you cut
C-k: cut from the cursor to the end of the line
M-d: cut the next word
```

`M-y` is the hidden gem here. Cut three different things, press `C-y`, then `M-y` a few times, and watch Emacs walk back through your clipboard history.

For searching and replacing:

```
C-s: search forward, as you type
C-r: search backward
M-%: search and replace, asking at each match
```

In `M-%`, Emacs asks about every match: `y` to replace, `n` to skip, `!` to replace all the rest, `q` to stop. There's also `M-x replace-regexp` when you need regular expressions.

If you're not ready for any of this, `M-x cua-mode` turns on the familiar Ctrl+C, Ctrl+V, Ctrl+X and Ctrl+Z. Purists will frown.

Buffers and windows

Here's the vocabulary that trips people up. A **buffer** is anything open in Emacs: a file, a shell, a help page. A **window** is a space on screen showing a buffer. And what the rest of the world calls a window, Emacs calls a **frame**. Yes, really.

```
C-x b: switch to another buffer
C-x C-b: list all buffers
C-x k: close (kill) a buffer
C-x 2: split into top and bottom
C-x 3: split side by side
C-x o: jump to the other window
C-x 1: keep only this window
C-x 0: close this window
```

You can open a file on the left, its tests on the right and a shell at the bottom, all in one frame, all driven from the keyboard. Emacs 31 even added commands to rotate, flip and transpose window layouts, for when you change your mind about who sits where.

Emacs explains itself

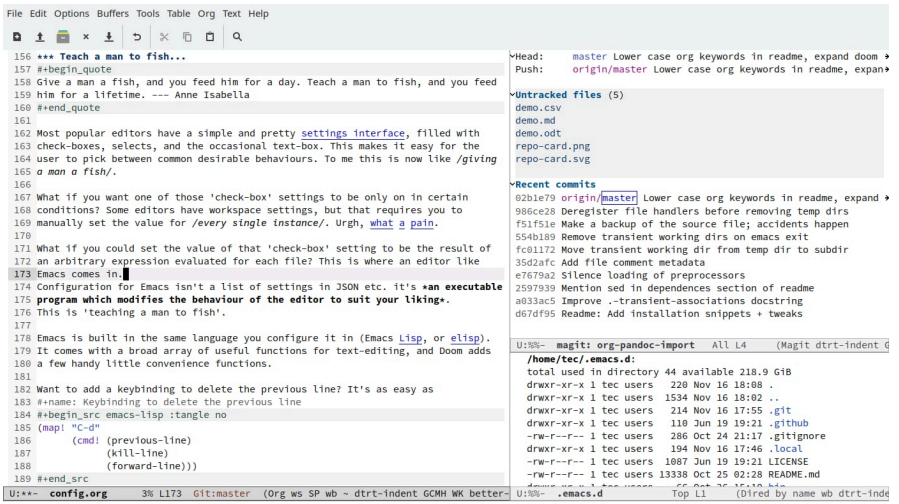
This is my favorite thing about Emacs, and it's the reason you'll never be stuck for long. Everything is documented from inside, and the help system knows exactly what's loaded right now.

```
C-h k: what does this key do? (press it after)
C-h f: describe a function
C-h v: describe a variable
C-h m: explain the modes active in this buffer
C-h a: search commands by keyword
C-h i: open the full manuals
```

Press `C-h k` and then `C-x C-s` and Emacs tells you which command runs, what it does and where it's defined. Click the link and you're reading its source code. No other editor I know lets you be this nosy.

The things that make people stay

Most people come to Emacs for the editing and stay for one of these.



■ *Org, Magit and Dired side by side. Image: Wikimedia Commons*

DIRED, A FILE MANAGER MADE OF TEXT

`C-x d` opens Dired, a directory listing you can edit like a file. Move with the usual keys, Enter opens a file, `d` marks for deletion and `x` deletes, `C` copies, `R` renames or moves, `+` creates a folder. The best trick is `C-x C-q`: it makes the listing editable, so you can rename forty files with a search and replace, then `C-c C-c` to apply.

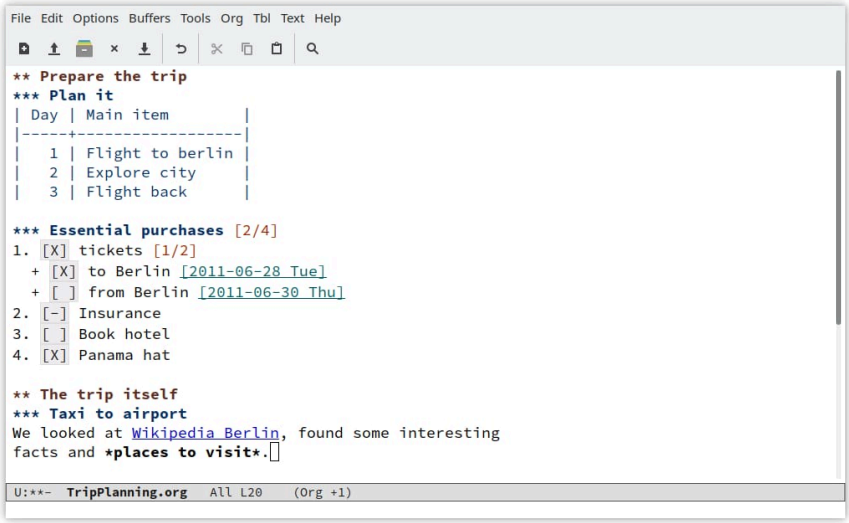
ORG MODE

Org is the reason a lot of people install Emacs in the first place. It's plain text with structure: outlines you fold and unfold with Tab, to-do lists, deadlines, tables that recalculate, links, agendas, and export to HTML, PDF (through LaTeX) or OpenDocument. The files are ordinary `.org` text files, readable anywhere, forever.

To try it, open a file ending in `.org` and type:

```
* Weekend
** TODO Fix the backup script
DEADLINE: <2026-11-01 Sun>
** DONE Buy cat food
* Ideas
- A guestbook that only accepts haiku
```

Press Tab on a heading to fold it. `C-c C-t` cycles a heading between TODO and DONE. `C-c C-d` adds a deadline with a little calendar. `M-↑` and `M-↓` move



A plain text Org file planning a trip. Image: Wikimedia Commons

whole sections around, children included. And `C-c C-e` opens the export menu. You can live in Org for years and keep finding new things.

MAGIT, GIT THAT MAKES SENSE

Magit isn't built in, you install it as a package (the config below does that for you), but almost everyone who uses Emacs and Git ends up with it. `C-x g` opens a status screen for your repository. Move to a file and press `s` to stage it, `u` to unstage, Tab to see the diff, `c c` to write a commit, `P p` to push. You can even stage single lines of a change. People who swear they hate Emacs have been known to keep it installed just for Magit.

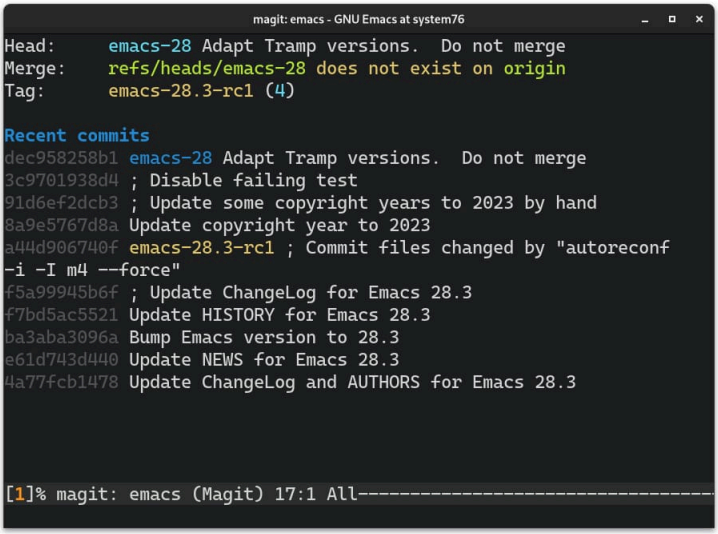
EDITING FILES ON OTHER MACHINES

TRAMP lets you open remote files as if they were local. Press `C-x C-f` and type:

```
/ssh:you@your-server:/etc/nginx/nginx.conf
```

Emacs connects over SSH, opens the file and saves it back when you press `C-x C-s`. Need root? Chain it:

```
/ssh:you@your-server|sudo:your-server:/etc/nginx/nginx.conf
```



■ *Magit, Git inside Emacs. Image: Wikimedia Commons*

Dired works over TRAMP too, so you can browse a server's folders like they were on your desk. On Windows, TRAMP can use `plink` from PuTTY instead of `ssh` (with `/plink:` instead of `/ssh:`) if you don't have OpenSSH installed.

A SHELL (OR THREE)

`M-x eshell` is a shell written in Emacs Lisp. It works the same on Linux and Windows and you can mix Lisp into your commands. `M-x shell` runs your system shell inside a buffer, so you can search and copy its output like any text. And `M-x term` runs a real terminal emulator for programs that need one.

PROGRAMMING

Modern Emacs comes with Eglot, a client for language servers (the same ones VS Code uses), built in since Emacs 29. Install the language server for your language, say `pyright` for Python or `gopls` for Go, open a file and run `M-x eglot`. You get completion, go to definition, rename, errors as you type. Emacs also ships tree-sitter modes for a bunch of languages, which means faster and more accurate syntax highlighting.

Your first config

Emacs reads its settings from a file called `init.el`. It lives in `~/.emacs.d/init.el` or, on newer setups, `~/.config/emacs/init.el` (on Windows, remember, `~` is probably that `AppData\Roaming` folder). Create the file, paste this in, save, and restart Emacs:

```elisp
;;; init.el --- a small, honest start

;; Quieter startup
(setq inhibit-startup-screen t)
(setq ring-bell-function 'ignore)
(tool-bar-mode -1) ; the menu bar stays, it helps while you learn

;; Comfort
(global-display-line-numbers-mode 1)
(column-number-mode 1)
(electric-pair-mode 1) ; close brackets and quotes for you
(global-auto-revert-mode 1) ; reload files changed outside Emacs
(recentf-mode 1) ; remember recent files
(savehist-mode 1) ; remember minibuffer history
(save-place-mode 1) ; reopen files where you left them
(which-key-mode 1) ; show possible keys as you type (Emacs 30+)

;; A good built-in theme (try modus-operandi for a light one)
(load-theme 'modus-vivendi t)

;; Keep backup files out of your projects
(setq backup-directory-alist
`(("." . ,(expand-file-name "backups" user-emacs-directory))))

;; Packages from MELPA
(require 'package)
(add-to-list 'package-archives '("melpa" . "https://melpa.org/packages/") t)

(use-package vertico ; better completion lists
:ensure t
:init (vertico-mode))

(use-package orderless ; type words in any order to find things
:ensure t
:custom (completion-styles '(orderless basic)))

(use-package marginalia ; little descriptions next to completions
:ensure t
:init (marginalia-mode))

(use-package magit
:ensure t
:bind ("C-x g" . magit-status))
```

The first start after this takes a minute, because Emacs downloads the packages. After that it's instant. If you're on Emacs 29, delete the

`which-key-mode` line, since which-key only became part of Emacs in version 30.

Two things worth knowing while you tinker. You don't need to restart after every change: `M-x eval-buffer` reloads the file you're editing. And if a change breaks something, start Emacs with `emacs -Q` to skip your config completely, then fix the file in peace.

Tricks I'd show a friend

Keyboard macros. Press `F3`, do something once (edit a line, move down), press `F4`. Now every `F4` repeats it. Fix two hundred lines of a CSV in ten seconds and feel like a wizard.

Repeat anything. `C-u 8 C-n` moves down eight lines. `C-u 40 #` types forty hashes. Almost every command takes a number this way.

Rectangles. Select a block, then `C-x r t` and type something: it gets inserted at the start of every line in the block. Perfect for commenting a bunch of lines or adding a prefix to a list. `C-x r k` cuts a rectangle instead.

Lazy autocomplete. `M-/` completes the word you're typing using words already in your open buffers. No setup, works everywhere.

Find every match at once. `M-x occur` lists every line matching a pattern in a separate buffer. Click a result to jump there.

Type any character. `C-x 8 RET`, then type a name like `snowman` or `em space`. Emacs knows the whole Unicode table.

Zoom. `C-x C-+` and `C-x C--` make the text bigger and smaller, `C-x C-0` goes back to normal.

Fill paragraphs. `M-q` rewraps a messy paragraph to a nice width. Great for plain text, commit messages and emails.

Instant startup. Run `emacs --daemon` once (or `M-x server-start` inside Emacs), then open files with `emacsclient -c file.txt`. The window shows up immediately because Emacs is already running. On Windows, the matching program is `emacsclientw.exe`.

A calculator. `M-x calc` is a full RPN calculator with units, matrices and algebra. Type `2 RET 3 +` and you're in.

Games. `M-x tetris`. `M-x snake`. And `M-x butterfly`, an old joke from xkcd that someone implemented for real. Try it once, at least.

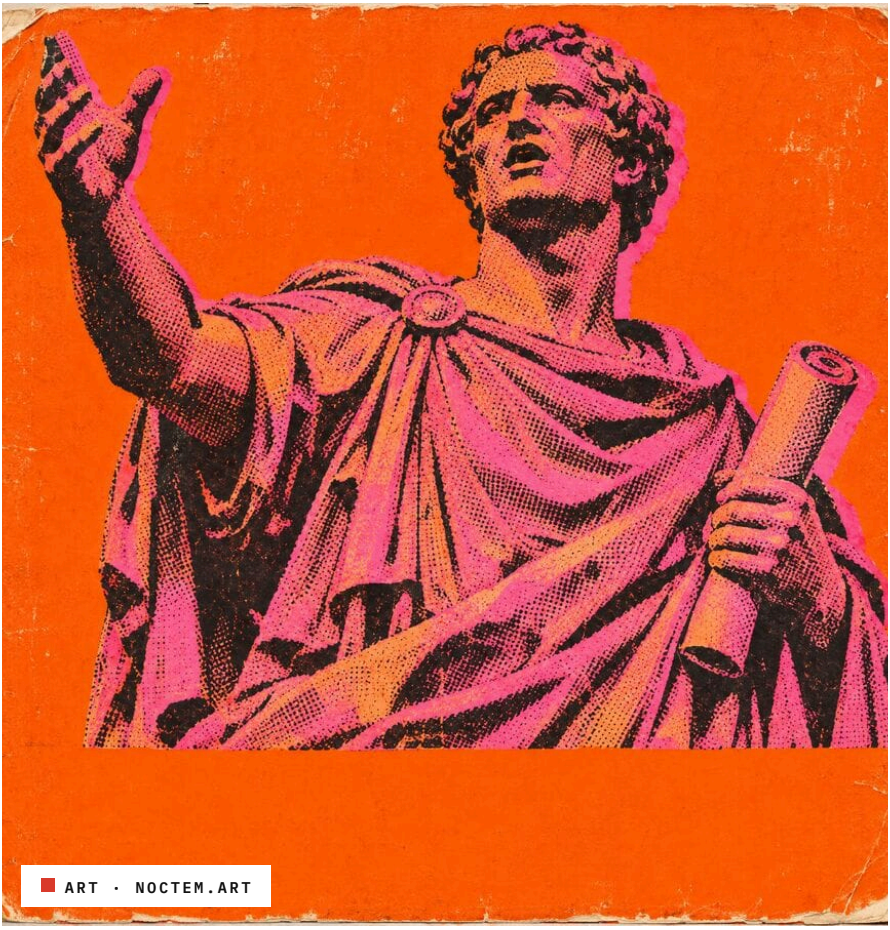
Where to go from here

Don't try to learn everything. Use Emacs for one thing first, notes in Org, or Git through Magit, or editing config files on your servers, and let the rest grow around it. Every time you catch yourself doing something

annoying twice, ask Emacs for help (`C-h a` and a keyword) and you'll almost always find a command that already does it.

For reading, everything is free and most of it is already on your disk. `C-h t` for the tutorial, `C-h i` for the Emacs manual, and inside that same menu, "An Introduction to Programming in Emacs Lisp," which is how people go from using Emacs to bending it. The manuals are also online at gnu.org/software/emacs/manual.

It's a forty-year-old program that still gets better every year, and it'll happily outlive whatever editor is trendy this month. Give it a weekend. Worst case, you learn that `C-g` gets you out of anything, and that's a pretty good life lesson too. ■



ART · NOCTEM.ART

02 EMACS · ORG-MODE · TUTORIAL 6 NOVEMBER 2026

Org mode from zero: a complete, friendly guide

A complete, friendly guide to Org mode in Emacs: headings and folding, TODOs, dates and repeaters, the agenda, capture, links, tables with formulas, clocking, code blocks you can run, export to HTML and PDF, a starter config and a cheat sheet.



Org mode is two things that grew up together. It's a way of writing plain text with a little structure (stars for headings, dashes for lists, pipes for tables), and it's the part of Emacs that knows what to do with that text. You write a file that any program on earth can open, and Emacs turns it into an outliner, a to-do list, a calendar, a spreadsheet, a notebook that runs code and a publishing tool.

It started in 2003 as a better way to fold outlines, and it's been part of Emacs since the mid-2000s. In 2026 it's on version 9.8 and still going. The files you write today will still open decades from now, in any text editor, with no app, no account and no export step, because they're just text.

This guide starts at the first star and goes all the way to agendas, code blocks and exporting a PDF. You don't need to read it in one sitting. Each section works on its own, and the cheat sheet at the end is there for the day you forget a key.

Before you start

You need Emacs. If you don't have it yet, my post Emacs for the curious covers installing it on Linux and Windows and surviving the first day. Org already comes inside Emacs, so there's nothing else to install. If you ever want the newest Org, it's on GNU ELPA, the official package archive.

A quick reminder of how Emacs keys are written, because this post is full of them:

```
C-c: hold Ctrl and press c
M-x: hold Alt and press x (or press Esc, then x)
C-c C-t: Ctrl+c, then Ctrl+t
S-TAB: Shift+Tab
M-RET: Alt+Enter
```

To start, open a file whose name ends in `.org`, for example `C-x C-f ~/org/inbox.org`. Emacs sees the extension and switches to Org mode on its own. Everything below happens in that file.

One star, two stars

A heading is a line that starts with one or more stars and a space. More stars, deeper level:

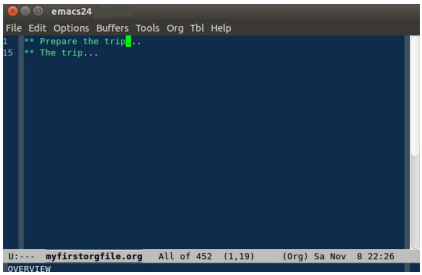
```
* Home
** Kitchen
*** Fix the dripping tap
** Garden
* Work
** Blog
*** Write the Org post
```

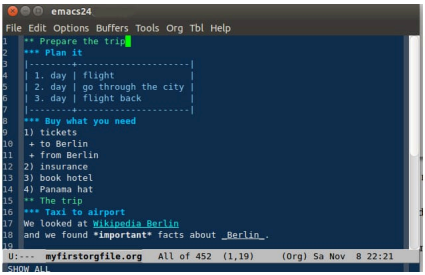
Everything under a heading, until the next heading of the same level, belongs to it. That's the whole idea of an outline, and it's what makes the next trick possible.

FOLDING WITH TAB

Put the cursor on a heading and press `TAB`. The heading closes and shows three dots, meaning there's something inside. Press `TAB` again and you see the children. Once more and everything under it opens. Org calls this cycling: folded, children, everything.

`S-TAB` does the same for the whole file at once. Press it a few times and the file goes from only top-level headings, to all headings, to every line of text.





This is the first thing that makes Org feel different from a normal text file. A file with three hundred lines can look like five lines, and you open only the part you're working on.

MOVING THINGS AROUND

You never need to cut and paste a section in Org. You grab it with the keyboard and move it:

```
M-RET: new heading at the same level
C-RET: new heading after the whole current section
M-left / M-right: promote or demote this heading
M-S-left / M-S-right: promote or demote it with all its children
M-up / M-down: move the whole section up or down, children included
```

Try `M-down` on "Kitchen" in the example above. The heading jumps below "Garden" and takes the dripping tap with it.

Writing inside headings

Under a heading you write normal paragraphs. Org gives you a handful of marks for inline styles:

```
*bold* /italic/ _underline_ =verbatim= ~code~ +strike-through+
```

Lists are as simple as they look. Dashes or plus signs for bullets, numbers for ordered lists, and a double colon for term and definition:

```
- eggs
- flour
1. preheat the oven
2. mix everything
- Org :: an outliner made of plain text
```

`M-RET` inside a list adds a new item, and `M-up` and `M-down` move items around, the same keys you use for headings. Org also renumbers ordered lists for you when you move things.

CHECKBOXES

A checkbox is a pair of square brackets with a space inside, placed right after the dash. The fastest way to make one is to let Org write it: put the cursor on a list item and press `C-u C-c C-c`. Then `C-c C-c` ticks it, and the empty box becomes `[X]`. In a list that already has checkboxes, `M-S-RET` adds a new item with an empty box.

Add a cookie to the heading above the list, `[/]` or `[%]`, and Org keeps count for you:

```
* Packing for the weekend [2/2]
- [X] passport
- [X] charger
```

Every time you tick or untick a box, the cookie updates. Untick the charger and the heading goes back to `[1/2]`. `[%]` shows a percentage instead.

TODO: turning notes into tasks

Any heading becomes a task when it starts with a keyword:

```
* TODO Renew the domain
* DONE Pay the electricity bill
```

`C-c C-t` cycles the heading through no keyword, TODO and DONE. `S-left` and `S-right` do the same thing in both directions.

YOUR OWN KEYWORDS

Two states are enough for a shopping list, but real work has more steps. Put a line like this at the top of a file:

```
#+TODO: TODO NEXT WAIT | DONE CANCELED
```

Everything before the bar is open, everything after it is finished. Press `C-c C-c` on that line so Org reloads it, and now `C-c C-t` walks through all five. The config at the end of this post sets the same keywords for every file.

REMEMBERING WHEN YOU FINISHED

Add this to your config and Org stamps the moment you close a task:

```
(setq org-log-done 'time)                                    ELISP
```

```
* DONE Pay the electricity bill
CLOSED: [2026-11-06 Fri 10:42]
```

It looks like a small thing until the day someone asks "when did we send that?" and the answer is sitting right under the heading.

PRIORITIES

A priority cookie goes right after the keyword:

```
* TODO [#A] Back up the server before the update
* TODO [#C] Reorganize the bookmarks
```

`A` is the highest and `C` the lowest. `S-up` and `S-down` on a heading change it, and `C-c ,` asks for it directly.

TAGS

Tags go at the end of the heading, between colons:

```
* TODO Call the accountant :work:phone:
* Trip to Ouro Preto :travel:
** TODO Book the hotel
```

`C-c C-q` adds or edits tags with completion. Children inherit their parents' tags, so "Book the hotel" counts as `:travel:` without saying it. That matters later, when you search by tag.

Dates: scheduled, deadline and plain

Org understands dates written between angle brackets:

```
<2026-11-10 Tue>
<2026-11-10 Tue 14:00>
```

You don't type them by hand. `C-c .` opens a little calendar and a prompt, and the prompt is smarter than it looks. You can type a date, but also things like these:

```
fri: next Friday
+3d: three days from today
+2w: two weeks from today
nov 20: the 20th of November
2pm: today at two in the afternoon
```

`S-left` and `S-right` on a date move it a day back or forward, and `S-up` and `S-down` change whatever part the cursor is on, year, month, day or hour.

SCHEDULED AND DEADLINE

Two kinds of dates mean something special to a task, and the difference is worth learning early:

```
* TODO Write the newsletter
SCHEDULED: <2026-11-09 Mon>
* TODO Send the tax documents
DEADLINE: <2026-11-20 Fri>
```

SCHEDULED means "this is when I plan to start working on it". DEADLINE means "this must be done by then". `C-c C-s` adds the first and `C-c C-d` the second. A deadline starts showing up in the agenda 14 days before the date, so it never catches you by surprise. You can change that window per task: `<2026-11-20 Fri -5d>` warns you only five days before.

THINGS THAT REPEAT

Add a repeater to a date and the task comes back after you mark it done:

```
* TODO Water the plants
SCHEDULED: <2026-11-09 Mon +1w>
* TODO Check the backups
SCHEDULED: <2026-11-09 Mon ++1w>
* TODO Change the water filter
SCHEDULED: <2026-11-09 Mon .+3m>
```

The three flavors behave differently when you're late:

```
+1w: moves exactly one week from the old date, even if that's still in
the past
++1w: moves forward in steps of one week until it lands in the future
.+3m: moves three months from the day you actually did it
```

`+` is good for bills, `++` for weekly rituals that should stay on the same weekday, and `.+` for chores where the interval matters more than the calendar, like a filter that lasts three months from the day you changed it.

DATES THAT DON'T NAG

Square brackets make an inactive date: `[2026-11-06 Fri]`. It's a note of when something happened, and the agenda ignores it. `C-c !` inserts one. Org itself uses them for the CLOSED stamp and for logs.

The agenda: every file, one view

Up to here everything lived in one file. The agenda is where Org starts to feel like an app. It reads all your Org files and builds views out of them: what's scheduled this week, every open task, everything tagged `:work:`.

First, the agenda needs a key that works everywhere in Emacs. The Org manual suggests these three, and the rest of this post assumes you have them:

```elisp
(global-set-key (kbd "C-c l") #'org-store-link)
(global-set-key (kbd "C-c a") #'org-agenda)
(global-set-key (kbd "C-c c") #'org-capture)
```

Second, the agenda needs to know which files to read. The simplest way is to keep all your Org files in one folder and point it there:

```elisp
(setq org-agenda-files '("~/org"))
```

You can also add the file you're in with `C-c [`. Now press `C-c a` and a menu appears. These are the views you'll use most:

```
a: the agenda for this week, with scheduled items and deadlines
t: every open task, from every file
m: match tags, like work or travel+urgent
s: search for words across all your files
```

LIVING IN THE AGENDA

The agenda is a live view, so you can work from inside it without opening the files:

```
f / b: next or previous week
.: jump back to today
d / w: day view or week view
t: change the TODO state of the item under the cursor
C-c C-s / C-c C-d: schedule or set a deadline
RET: open the item in its file
TAB: open it in the other window
I / O: clock in or out (more on that below)
r: refresh the view
s: save all Org files
q: close the agenda
```

The first time you press `C-c a a` on a Monday and see your week laid out, with deadlines from three different files already sorted by day, Org stops being a text format and starts being a planner.

YOUR OWN VIEWS

When a view becomes a habit, give it a key. This one shows today's agenda with every task marked NEXT under it:

```elisp
(setq org-agenda-custom-commands
'(("d" "Today and next"
((agenda "" ((org-agenda-span 'day)))
(todo "NEXT")))))
```

Now `C-c a d` is your morning page.

Capture: write it down before you forget

Ideas show up while you're doing something else. Capture lets you write them down without leaving what you're doing. Press `C-c c` from anywhere in Emacs, pick a template, type a line, press `C-c C-c`, and you're back where you were. The note went to the right file in the right shape.

Templates are short:

```elisp
(setq org-capture-templates
'(("t" "Task" entry (file "inbox.org")
"* TODO %?\n%U")
("n" "Note" entry (file "inbox.org")
"* %? :note:\n%U")
("j" "Journal" entry (file+olp+datetree "journal.org")
"* %<%H:%M> %?")))
```

The little codes are placeholders. `%?` is where the cursor waits for you, `%U` is the current date and time as an inactive date, and `%<%H:%M>` is the time in whatever format you like. `file+olp+datetree` files each entry under year, month and day headings, which turns `journal.org` into a diary that organizes itself. A relative file name like `inbox.org` lives inside `org-directory`, the folder you set in the config.

Inside the capture window you have three choices:

```
C-c C-c: save it and go back
C-c C-w: save it somewhere else (refile, see below)
C-c C-k: never mind, throw it away
```

Refile and archive: keeping it tidy

An inbox only works if it gets emptied. `C-c C-w` on any heading (refile) asks where it should live and moves it there, children and all. With this in your config, the list of destinations includes every heading up to three levels deep in your agenda files:

```elisp
(setq org-refile-targets '((org-agenda-files :maxlevel . 3)))
(setq org-refile-use-outline-path 'file)
(setq org-outline-path-complete-in-steps nil)
```

Finished projects can leave the stage too. `C-c C-x C-a` archives a heading: by default it moves it to an archive file next to the original, like `inbox.org_archive`. Nothing gets deleted, the agenda just stops seeing it.

One simple rhythm: capture everything into `inbox.org` during the week, then once a week open it, refile each item to its project, and archive what's done.

Links

Org links use double brackets, with an optional description:

```
[[https://orgmode.org][the Org website]]
[[file:~/org/projects.org][my projects]]
[[file:~/Pictures/rack.jpg]]
[[*Write the Org post]]
```

The last one points to a heading in the same file. When the description is there, Org shows only the description, underlined and clickable, and hides the brackets.

The keys:

```
C-c l: store a link to wherever you are (any file, any heading, even an
email in Emacs)
C-c C-l: insert a link, or edit the one under the cursor
C-c C-o: open the link under the cursor
C-c &: go back to where you were before you followed it
C-c C-x C-v: show linked images inside the buffer
```

`C-c l` and `C-c C-l` together are the trick. Store a link to a heading deep inside some project file, go to your journal, press `C-c C-l` and the stored link is right there in the prompt. That's how a pile of files turns into a web of notes.

Tables that align themselves

Start a line with a pipe, type the column names separated by pipes, and press `TAB`:

```
|Item|Qty|Price|Total|
|-
```

Org aligns the columns. The second line, a pipe and a dash, turns into a separator line when you press `TAB` on it. From then on, `TAB` jumps to the next cell and `RET` to the next row, and the table realigns itself as you type. You can write as messy as you want.

Moving and changing the table uses the same keys as headings, which is a nice touch:

```
M-left / M-right: move a column
M-up / M-down: move a row
M-S-right: insert a column
M-S-left: delete a column
M-S-down: insert a row
M-S-up: delete a row
C-c -: insert a separator line below
C-c ^: sort by the current column
```

FORMULAS

Tables can calculate. Here's a small shopping list:

```
| Item | Qty | Price | Total |
|-
| Coffee | 2 | 18 | |
| Bread | 3 | 7.5 | |
| Cheese | 1 | 32 | |
|-
| Total | | | |
#+TBLFM: $4=$2*$3::@>$4=vsum(@2..@-1)
```

Put the cursor on the `#+TBLFM` line and press `C-c C-c`. The Total column fills in with 36, 22.5 and 32, and the last row adds them up to 90.5.

The formula reads like this. `$4=$2*$3` says "column 4 is column 2 times column 3", for every row. After the double colon comes a second formula: `@>$4` is the last row, column 4, and `vsum(@2..@-1)` sums from row 2 to the row above. Change a price, press `C-c C-c` again, and everything updates.

You don't have to learn the syntax by heart. In any cell, `C-c =` asks for a formula for that column, and `C-c +` adds up the numbers in the column above and shows the result. For anything bigger, Org's tables can use Emacs Calc or Lisp in formulas, but the basics carry you a long way.

Properties and column view

Any heading can carry a drawer of key and value pairs:

```
* TODO Replace the NAS disks
:PROPERTIES:
:Effort: 2:00
:Cost: 1200
:END:
```

`C-c C-x p` adds a property with completion, and `C-c C-x e` sets the effort estimate. On their own, properties are just tidy metadata. Combined with column view, they become a spreadsheet of your outline. Add this line at the top of the file:

```
#+COLUMNS: %40ITEM %TODO %Effort %Cost
```

Then press `C-c C-x C-c` on a heading. The outline turns into a table, one row per heading, one column per property, and you can edit values right there. `q` goes back to normal.

Clocking time

Org can tell you where your hours went. On a task, press `C-c C-x C-i` to start the clock. The mode line at the bottom of Emacs shows the task and the running time. `C-c C-x C-o` stops it. Org writes the session under the heading:

```
* TODO Write the Org post
:LOGBOOK:
CLOCK: [2026-11-06 Fri 09:10]--[2026-11-06 Fri 11:25] => 2:15
:END:
```

A few related keys:

```
C-c C-x C-j: jump to the task that's being clocked
C-c C-x C-q: cancel the current clock
C-c C-x C-d: show the total time on every heading
M-x org-clock-report: insert a table with totals per task
```

The clock report is a block that updates with `C-c C-c`, so a monthly “where did my time go” is one key away. In the agenda, `I` and `O` clock the item under the cursor in and out.

Code blocks: a notebook in plain text

This is where Org goes past what a note app does. An Org file can hold code, run it and keep the output:

```
#+begin_src shell :results output
df -h /
#+end_src
```

Put the cursor inside and press `C-c C-c`. Org runs the command and writes the result right under the block, in a `#+RESULTS:` section. Run it again and the result is replaced.

Out of the box, only Emacs Lisp runs. To enable other languages, list them in your config:

```elisp
(org-babel-do-load-languages
'org-babel-load-languages
'((emacs-lisp . t)
(python . t)
(shell . t)))
```

Now Python works the same way:

```
#+begin_src python :results output
print(2 + 2)
#+end_src

#+RESULTS:
: 4
```

A few things that make code blocks pleasant:

```
C-c C-,: insert a block from a menu (src, quote, example and more)
C-c ': edit the block in a full buffer for that language, then C-c ' again
to come back
C-c C-c: run the block under the cursor
C-c C-v t: write all blocks marked with :tangle into their own files
```

That last key is the start of what people call literate configuration. You write your Emacs config, your shell scripts or your server notes as an Org document, with explanations between the code, and add `:tangle` to each block:

```
#+begin_src shell :tangle ~/bin/backup.sh
rsync -a ~/org/ server:/backups/org/
#+end_src
```

`C-c C-v t` writes the code out to the real files. The documentation and the code can't drift apart, because they're the same file.

If you prefer typing to menus, add `(require 'org-tempo)` to your config. Then `<s` followed by `TAB` expands into a source block, `<q` into a quote, `<e` into an example.

When Org asks "Evaluate this code block?" before running anything, that's on purpose. A file you download could contain code, and Org won't run it without asking.

Export: from text to HTML, PDF and friends

`C-c C-e` opens the export menu. Each letter picks a format, and a second letter picks what to do with it:

```
h h: HTML file
h o: HTML file, then open it in the browser
l p: PDF, through LaTeX
l o: PDF, then open it
o o: OpenDocument text (.odt), which LibreOffice and Word open
t U: plain text with Unicode, in a buffer
```

PDF export needs a LaTeX installation on your computer, such as TeX Live. HTML and ODT need nothing extra.

Markdown is not loaded by default. Add `(require 'ox-md)` to your config and `m m` appears in the menu.

A few lines at the top of a file control the export:

```
#+TITLE: A trip to Ouro Preto
#+AUTHOR: Your name
#+DATE: 2026-11-06
#+OPTIONS: toc:nil num:nil ^:{}
```

`toc:nil` removes the table of contents, `num:nil` removes the numbers in front of headings, and `^:{}` stops Org from turning `file_name` into "file" with a subscript "name", which gets in the way when you write about code. In the export menu, `C-s` limits the export to the heading under the cursor, handy for sending one section of a big file.

Code blocks export only the code by default. Add `:exports both` to a block to include its result, or `:exports results` to show only the output.

Small things worth knowing

Footnotes. `C-c C-x f` creates a footnote like `[fn:1]` and takes you to where its text goes. Press it again on the reference to jump back.

Focus on one section. `C-x n s` narrows the buffer to the current heading, so the rest of the file disappears. `C-x n w` brings it back. Great for writing long things.

Jump to any heading. `C-c C-j` lets you search the outline and go straight to a heading.

Sparse trees. `C-c /` opens a menu that folds the file to show only what matches. `C-c / t` shows only open tasks, `C-c / m` matches tags, `C-c / r` matches a regular expression. Everything else folds away until you press `S-TAB`.

Notes on a task. `C-c C-z` adds a dated note under the heading. Useful for "called them, they'll answer on Monday".

A cleaner look. `(setq org-startup-indented t)` indents text to match its heading, without changing the file. The stars of deeper levels become invisible except the last one.

Special characters. Type `\alpha` or `\rarr` and Org shows α and → if you set `(setq org-pretty-entities t)`. The file still holds the plain backslash version.

A starter config

Here's everything from this post in one place. Put it in your `init.el`, change the folder if you want, restart Emacs, and create `~/org` if it doesn't exist:

```elisp
;;; Org mode, a starting point

(require 'org)
(require 'org-tempo) ; <s TAB for code blocks
(require 'ox-md) ; Markdown export

;; Keys that work everywhere in Emacs
(global-set-key (kbd "C-c l") #'org-store-link)
(global-set-key (kbd "C-c a") #'org-agenda)
(global-set-key (kbd "C-c c") #'org-capture)

;; Where your files live
(setq org-directory "~/org")
(setq org-default-notes-file (expand-file-name "inbox.org" org-directory))
(setq org-agenda-files (list org-directory))

;; Task states: open ones before the bar, finished ones after
;; The letters are shortcuts; @ asks for a note when you pick that state
(setq org-todo-keywords
'((sequence "TODO(t)" "NEXT(n)" "WAIT(w@)" "|" "DONE(d)" "CANCELED(c@)")))

;; Record when tasks are done, and keep logs folded in a drawer
(setq org-log-done 'time)
(setq org-log-into-drawer t)

;; Capture
(setq org-capture-templates
'(("t" "Task" entry (file "inbox.org")
"* TODO %?\n%U")
("n" "Note" entry (file "inbox.org")
"* %? :note:\n%U")
("j" "Journal" entry (file+olp+datetree "journal.org")
"* %<%H:%M> %?")))

;; Refile anywhere up to three levels deep
(setq org-refile-targets '((org-agenda-files :maxlevel . 3)))
(setq org-refile-use-outline-path 'file)
(setq org-outline-path-complete-in-steps nil)

;; A custom agenda view: C-c a d
(setq org-agenda-custom-commands
'(("d" "Today and next"
((agenda "" ((org-agenda-span 'day)))
(todo "NEXT")))))

;; Looks
```

```
(setq org-startup-indented t)
(setq org-pretty-entities t)

;; Languages for code blocks
(org-babel-do-load-languages
'org-babel-load-languages
'((emacs-lisp . t)
(python . t)
(shell . t)))
```

With `TODO(t)` and friends in the keywords, `C-c C-t` stops cycling and shows a small menu instead: press `t`, `n`, `w`, `d` or `c` to jump straight to a state. If Python code blocks complain that `python` doesn't exist, your system probably calls it `python3`, and `(setq org-babel-python-command "python3")` fixes that.

A way to start

The worst way to learn Org is to set up everything on the first day. A gentler path:

1. **Week one: one file.** Keep a single `notes.org`. Use headings, `TAB`, lists and `M-up` / `M-down`. Nothing else.
2. **Week two: tasks.** Add TODO, `C-c C-t`, SCHEDULED and DEADLINE. Open the agenda with `C-c a a` every morning.
3. **Week three: capture.** Set up one template and use `C-c c` for every idea, task and link that shows up.
4. **Later, as needed.** Tables when you need numbers, code blocks when you need to run something, export when you need to send something to someone.

Every feature in Org is optional. A file with three headings and a TODO is already a perfectly good Org file.

Org outside Emacs

Plain text travels well, and Org files are read by more programs than people expect:

- GitHub, GitLab and Gitea render `.org` files on the web, the same way they render Markdown.
- Pandoc converts Org to and from Markdown, Word, HTML, EPUB and many other formats.
- On Android, Orgzly Revived reads and edits Org files, with saved searches and reminders. It's on F-Droid.
- On iOS, beorg does the same, with a to-do list and an agenda.
- In Neovim, nvim-orgmode brings a good part of Org to Vim users.

Put your `~/org` folder in whatever sync you already trust, and the same files follow you from the desktop to the phone.

Cheat sheet

Structure:

```
TAB: fold or unfold the heading
S-TAB: fold or unfold the whole file
M-RET: new heading or list item
C-RET: new heading after the current section
M-left / M-right: promote or demote
M-S-left / M-S-right: promote or demote with children
M-up / M-down: move the section
C-x n s / C-x n w: narrow to the section / show everything again
C-c C-j: jump to a heading
```

Tasks and dates:

```
C-c C-t: change the TODO state
S-left / S-right: cycle the state (or move a date)
S-up / S-down: change the priority (or the date part under the cursor)
C-c C-q: set tags
C-c C-s: schedule
C-c C-d: deadline
C-c .: insert a date
C-c !: insert an inactive date
C-c C-c: tick a checkbox, update a cookie, recalculate, run code (it does
the right thing for where you are)
C-u C-c C-c: turn a list item into a checkbox
```

Agenda, capture and links:

```
C-c a: open the agenda menu
C-c c: capture
C-c C-w: refile
C-c C-x C-a: archive
C-c l: store a link
C-c C-l: insert or edit a link
C-c C-o: open a link
```

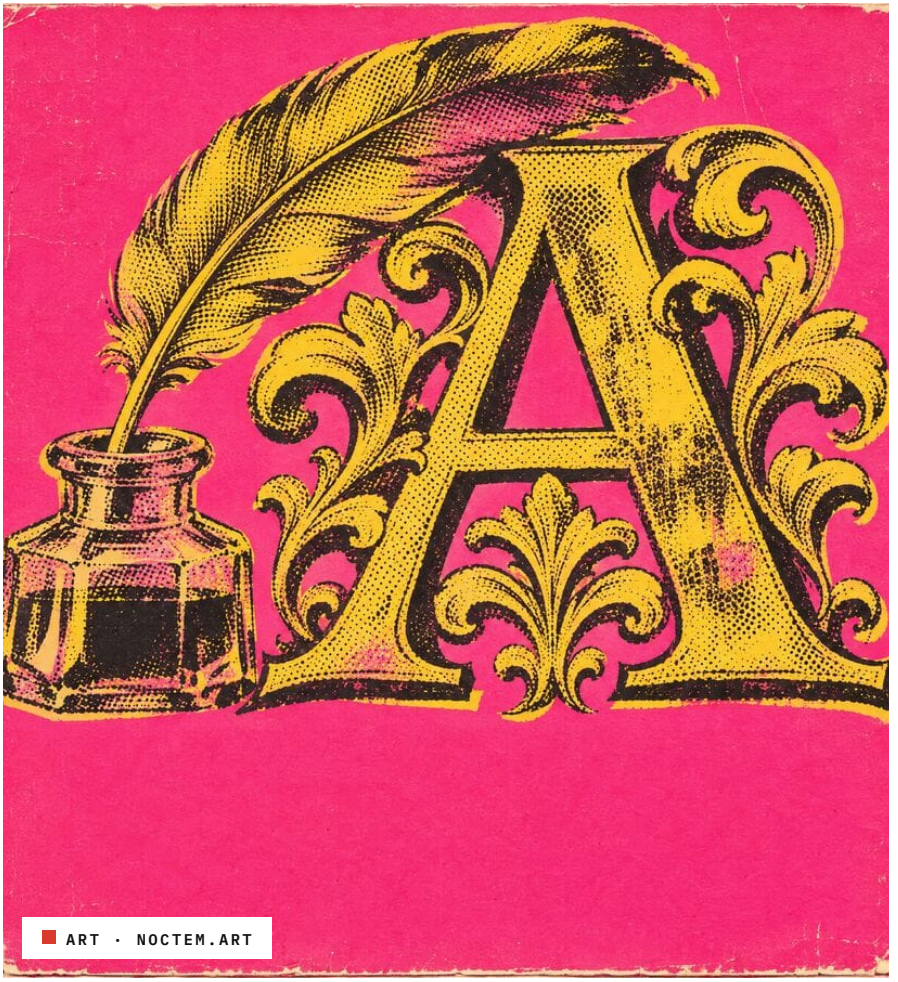
Clock, code and export:

```
C-c C-x C-i: clock in
C-c C-x C-o: clock out
C-c C-,: insert a block
C-c ': edit a code block in its own buffer
C-c C-v t: tangle code blocks into files
C-c C-e: export menu
```

Where to go from here

The Org manual is inside Emacs: press `C-h i`, then look for Org Mode in the list. It's long, well written and full of examples, and it's also online at orgmode.org/manual. Worg, the community wiki on the same site, has tutorials, configs and ideas from people who have been using Org for years. For a guided path, modeorg.org has 27 lessons and a 30-day plan.

And when you forget a key, remember the one key that never changes in Org: `C-c C-c` almost always does what you meant, wherever the cursor is. ■

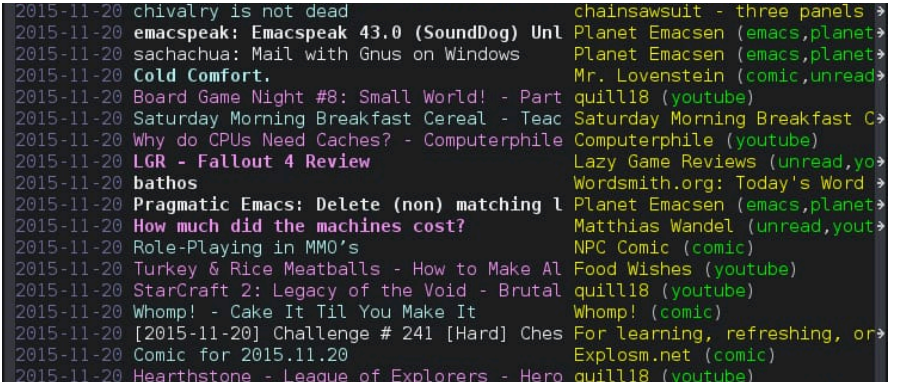


ART · NOCTEM.ART

03 EMACS · TUTORIAL · TOOLS 8 NOVEMBER 2026

Emacs beyond the basics, part I: feeds, gopher, the fediverse and mail

Emacs as a small operating system: read RSS with elfeed, browse gopher and Gemini with Elpher, follow the fediverse with mastodon.el, chat on IRC and Matrix, read the web in EWW and send mail. Keys, configs and a starter block.



The first Emacs guide on this blog repeated an old joke: Emacs is a great operating system lacking a decent text editor. This post takes the joke seriously. Emacs can read your feeds, browse gopher and Gemini, talk to the fediverse, sit in IRC rooms, browse the web and handle your email, all with the same keys you already learned for editing text.

You don't have to move your whole life into it. The fun is picking one thing, trying it for a week and seeing if it sticks. Each section below works on its own.

Getting packages

Some of what follows is built into Emacs, and some comes as packages. Emacs already knows two package archives out of the box, GNU ELPA and NonGNU ELPA. The config from the first guide also adds MELPA, the

biggest community archive. With those three, every package in this post is one `use-package` block away.

Feeds with elfeed

If you follow blogs, news sites, YouTube channels or forums through RSS, elfeed turns Emacs into a fast, keyboard-driven feed reader. It's on NonGNU ELPA, so Emacs can install it without any extra setup.

```elisp
(use-package elfeed
:ensure t
:bind ("C-c w" . elfeed)
:custom
(elfeed-feeds
'(("https://planet.emacslife.com/atom.xml" emacs)
("https://example.com/feed.xml" blogs friends)
("https://www.youtube.com/feeds/videos.xml?channel_id=CHANNEL_ID" video))))
```

Each feed is a list: the address first, then any tags you want to give it. Tags are how you'll slice your reading later. If you already have your subscriptions in another reader, export them as OPML and bring them in with `M-x elfeed-load-opml`.

Press `C-c w` to open elfeed, then `G` to fetch every feed. The main screen is a list of entries, newest first:

```
RET: read the entry
b: open the entry in your web browser
n / p: next / previous entry
r: mark as read
u: mark as unread
+ / -: add or remove a tag
s: filter the list as you type
c: clear the filter
G: fetch new entries
q: quit
```

Inside an entry, `n` and `p` jump to the next and previous one, `b` opens it in the browser and `q` goes back to the list.

The filter is the clever part. The default shows unread entries from the last six months, and you can write your own with a small language:

```
+unread: only unread entries
+emacs: only entries tagged emacs
-video: hide entries tagged video
@2-weeks-ago: only entries from the last two weeks
=example.com: only entries from feeds that match this text
```



Mix them as you like. `@1-week-ago +unread -video` shows what you haven't read this week, minus the videos. Press `s`, type the filter, and the list updates while you type.

Emacs also ships a feed reader of its own, Newsticker (`M-x newsticker-treeview`), if you'd rather not install anything. Elfeed is faster and nicer to filter, but it's good to know the built-in one is there.

Gopher and Gemini with Elpher

Gopher is the protocol that almost became the web, back in the early 90s. Gemini is its modern cousin, a small protocol made for plain text pages without tracking or scripts. Both are alive and full of personal sites, and Elpher, from NonGNU ELPA, browses both.

```elisp
(use-package elpher
:ensure t)
```

`M-x elpher` opens a start page with your bookmarks. To visit somewhere, press `g` and type an address:

```
gopher://sdf.org
gemini://geminiprotocol.net
```

From there, it's a keyboard browser:

```
TAB: jump to the next link
RET: follow the link under the cursor
u: go back
g: go to an address
a: bookmark the link under the cursor
A: bookmark the page you're on
B: show your bookmarks
d: download the link under the cursor
.: view the raw source of the page
R: reload
```

If you read the SDF guide, this is the comfortable way to read the gopherholes of SDF members without leaving your editor.

The fediverse with mastodon.el

mastodon.el, from NonGNU ELPA, is a full client for Mastodon. It also works with other servers that speak the same API, like GoToSocial, Pleroma and Akkoma. Tell it your server and your username:

```elisp
(use-package mastodon
:ensure t
:custom
(mastodon-instance-url "https://your.server")
(mastodon-active-user "yourname"))
```

Run `M-x mastodon`. The first time, it opens your browser so you can authorize the app on your server, then asks you to paste back the code the server shows you. It keeps the login in your auth-source file (usually `~/.authinfo.gpg`), so you only do this once. After that, you land on your home timeline.

The keys are short and easy to remember:

```
H: home timeline
L: local timeline
F: federated timeline
N: notifications
n / p: next / previous post
t: write a new post
r: reply
b: boost
f: favourite
k: bookmark
T: open the whole thread
o: open the post in the browser
s: search
O: your profile
g: refresh
Q: close mastodon.el
```



Writing a post opens a small buffer. Type, then `C-c C-c` sends it. Since it's an Emacs buffer, everything you know works there, including the spell checker (`M-x flyspell-mode`).

Chat: IRC and Matrix

ERC is an IRC client that has been part of Emacs for a long time. Run `M-x erc-tls`, answer the questions about server, port and nickname, and you're connected. Inside, it's IRC as usual: `/join #channel`, `/msg someone hello`, `/part` to leave.

To join the same rooms every time, list them per server:

```elisp
(setq erc-autojoin-channels-alist
'(("libera.chat" "#emacs")))
```

Libera.Chat, where a lot of free software projects live, has an `#emacs` channel full of patient people. Many public Unix systems and pubnixes also run their own IRC servers, and ERC talks to all of them. If your server asks for a SASL login, ERC has a `sasl` module: add it to `erc-modules` and use your NickServ password.

For Matrix, there's Ement.el, from GNU ELPA. `M-x ement-connect` asks for your Matrix ID and password, and then lists your rooms.

The web with EWW

EWW is a web browser built into Emacs. It doesn't run JavaScript and it shows pages as text with images, which sounds limiting until you realize how many pages read better that way. Documentation, blogs, Wikipedia and most of the small web look great in it.

`M-x eww` asks for an address or a search term. Then:

```
l / r: back / forward
g: reload
R: readable mode, which strips everything but the main text
&: open this page in your normal browser
w: copy the address
b: bookmark this page
B: list bookmarks
d: download the link under the cursor
v: view the page source
G: go somewhere else
```

`R` is the one that makes EWW worth it. It throws away menus, sidebars and banners and leaves only the article. And `&` is the escape hatch: when a page really needs a full browser, one key sends it there.

Mail

Email is the deepest rabbit hole here, so this section is a map more than a manual. There are three common paths.

Gnus comes with Emacs. It started as a newsreader for Usenet and grew into a mail client. It's powerful and has its own way of thinking, which some people love and others don't.

mu4e and notmuch both work with mail stored on your disk. A separate program, usually `mbsync` from the isync project, downloads your mail from the server into a local folder. Then `mu` or `notmuch` indexes it, and Emacs gives you fast search, threads and tags on top. Both are installed with your system's package manager, and their manuals explain the setup step by step.

Sending mail, on the other hand, is simple and built in. Tell Emacs who you are and which server to use:

```elisp
(setq user-full-name "Your Name"
user-mail-address "you@example.com"
send-mail-function 'smtpmail-send-it
smtpmail-smtp-server "smtp.example.com"
smtpmail-smtp-service 587
smtpmail-stream-type 'starttls)
```

Put your password in a file called `~/.authinfo.gpg`, one line per server:

```
machine smtp.example.com login you@example.com password your-password port
587
```

The `.gpg` at the end makes Emacs encrypt the file with GnuPG when you save it, and decrypt it when it needs the password. Your password never sits in plain text in your config.

Now `C-x m` opens a new message. Fill in the To and Subject lines, write, and `C-c C-c` sends it.

Small things worth knowing

A few more built-in tools that make Emacs feel like a small desktop:

- `M-x calendar` shows a calendar. Move with the arrows, `.` jumps to today, and `M-x diary` keeps simple appointments in a text file.
- `M-x dictionary` looks up words in online dictionaries.
- `M-x proced` lists running processes, like `top`, and lets you kill them with a key.
- `M-x webjump` opens favorite sites by short names.
- `M-x eshell` is a shell that works the same on Linux and Windows.
- `M-x doctor` is a psychotherapist from the 1960s. It won't help, but it will listen.

If you're on Emacs 31

Two changes in Emacs 31 make this way of living a bit nicer. In a terminal, the mouse now works by default in compatible terminals, so `emacs -nw` over SSH lets you click links in elfeed or scroll a timeline. And there's a new theme called `newcomers-presets`, off by default, that turns on a set of friendlier settings for people new to Emacs. Enable it like any other theme, with `M-x load-theme`.

A starter block

Here are the packages from this post together, to paste after the config from the first guide:

ELISP

```
;;; Living inside Emacs

(use-package elfeed
:ensure t
:bind ("C-c w" . elfeed)
:custom
(elfeed-feeds
'(("https://planet.emacslife.com/atom.xml" emacs))))
```

```
(use-package elpher
:ensure t)

(use-package mastodon
:ensure t
:custom
(mastodon-instance-url "https://your.server")
(mastodon-active-user "yourname"))

(use-package ement
:ensure t)

(setq erc-autojoin-channels-alist
'(("libera.chat" "#emacs")))
```

Select it and press `TAB` to let Emacs indent it nicely. Then `M-x eval-buffer`, or just restart, and every tool in this post is a command away. ■

■ ■ ■

04 EMACS · ELISP · LISP 9 NOVEMBER 2026

Emacs beyond the basics, part II: your first steps in Emacs Lisp

A gentle first step into Emacs Lisp: read the syntax, write your own commands, bind keys, use hooks and advice, build a quick-notes command and learn to debug and read Emacs's own source. No programming background needed.

The first Emacs guide on this blog ended by pointing at a manual called "An Introduction to Programming in Emacs Lisp", which is how people go from using Emacs to bending it. This post is the friendly version of that step. No computer science background needed. If you can copy a line into your `init.el`, you can do everything here.

The goal is small and practical. By the end you'll read Lisp without panicking, write your own commands, give them keys, hook them into modes and know how to dig into any function Emacs has. That's most of what people ever do with Elisp.

Why bother

Every key you press in Emacs runs a function written in Emacs Lisp. Your config is Lisp. The packages you install are Lisp. Once you can read a little of it, three things change:

- Your `init.el` stops being a list of spells copied from the internet. You know what each line does, and you can fix it when it breaks.
- When something annoys you, you can write the fix yourself, usually in five or ten lines.
- The source of every command is one click away, so Emacs becomes its own textbook.

Where to try things

You don't need to create a file to experiment. Emacs gives you three playgrounds.

The `*scratch*` buffer opens with Emacs, in a mode made for this. Type an expression, put the cursor right after it and press `C-j`. The result gets printed on the next line.

`M-:` asks for an expression in the minibuffer, runs it and shows the result at the bottom of the screen. It works from any buffer, which makes it great for quick questions like "what's the value of this variable right now?".

`C-x C-e` runs the expression right before the cursor, in any Lisp file. This is the one you'll use the most in your `init.el`: change a line, put the cursor after it, `C-x C-e`, and the change is live without restarting.

And if you like a proper REPL, `M-x ielm` gives you one, with a prompt and history.

Reading Lisp

Lisp looks scary because of the parentheses, but the rule is simple. A list in parentheses is a call: the first thing is the function, the rest are its arguments.

```elisp
(+ 1 2) ; => 3
(* 2 (+ 3 4)) ; => 14
(concat "Hello, " "Emacs") ; => "Hello, Emacs"
(upcase "hello") ; => "HELLO"
```

The inner parentheses run first, so `(* 2 (+ 3 4))` becomes `(* 2 7)`, which becomes 14. That's the whole grammar. Everything else is vocabulary.

A semicolon starts a comment. Strings go in double quotes. `t` means true, and `nil` means false and also "empty list". Those two show up everywhere.

Sometimes you want a list as data, not as a call. That's what the quote is for:

```elisp
'(+ 1 2) ; => (+ 1 2), just a list with three things in it
(+ 1 2) ; => 3
(car '(apples pears plums)) ; => apples, the first item
(cdr '(apples pears plums)) ; => (pears plums), the rest
```

The same quote is why config files are full of things like `'modus-vivendi` or `#'display-line-numbers-mode`. The `'` says "this name, as it is". The `#'` is the same idea for names of functions, and it helps Emacs warn you when you misspell one.

A note about indentation: Lisp doesn't care about it. The blocks of code in this post are written flat, and Emacs will indent them properly for you. Paste the code, select it and press `TAB`, or use `C-M-\`.

Variables

`setq` sets a variable. `let` creates variables that only exist inside its parentheses:

```elisp
(setq my-name "Pablo")

(let ((x 10)
(y 32))
(+ x y)) ; => 42, and x and y vanish after this
```

`format` builds a string with placeholders, `%s` for anything and `%d` for whole numbers:

```elisp
(format "Hi, %s! You have %d buffers open." my-name (length (buffer-list)))
```

For settings that Emacs or a package offers as options (the ones you'd see in `M-x customize`), Emacs 29 added `setopt`. It works like `setq`, but it also runs any code the option needs when it changes, and it warns you when the value has the wrong type:

```elisp
(setopt fill-column 80)
```

A simple habit: `setopt` for options, `setq` for everything else.

Your first command

A function is made with `defun`. Give it a name, a list of arguments, a short description and a body. Add `(interactive)` and it becomes a command, something you can run with `M-x` or bind to a key.

```elisp
(defun my-insert-date ()
"Insert today's date at point, like 2026-11-06."
(interactive)
(insert (format-time-string "%Y-%m-%d")))
```

Put the cursor after the last parenthesis and press `C-x C-e`. Now `M-x my-insert-date` works. And because you wrote a description, `C-h f my-insert-date` shows it, just like any built-in command.

Two habits worth having from day one. First, start your function names with a prefix, like `my-`, so they never clash with something Emacs or a package already uses. Second, always write the description. Future you will read it.

Giving it a key

Emacs 29 added a clean way to bind keys:

```elisp
(keymap-global-set "C-c d" #'my-insert-date)
```

Now `C-c d` inserts the date anywhere. The choice of `C-c` and a letter is on purpose. Emacs keeps every `C-c` followed by a plain letter for you, the user. Neither Emacs nor well-behaved packages will ever take those keys, so they're the safe place for your own commands.

You can also bind a key only inside one mode, using that mode's keymap:

```elisp
(with-eval-after-load 'org
(keymap-set org-mode-map "C-c n" #'my-new-note))
```

`with-eval-after-load` waits until Org is actually loaded before touching its keymap. Without it, Emacs would complain that `org-mode-map` doesn't exist yet. You'll write `my-new-note` in a minute.

Asking the user for something

`interactive` can do more than mark a function as a command. Give it a code and Emacs will ask the user for the arguments. `"s"` asks for a string:

```elisp
(defun my-greet (name)
"Say hello to NAME."
(interactive "sYour name: ")
(message "Hello, %s!" name))
```

Run `M-x my-greet` and Emacs prompts "Your name:" in the minibuffer. Other codes ask for a number (`"n"`), a file (`"f"`), a buffer (`"b"`) and more. `C-h f interactive` lists them all.

Working with text

Inside a buffer, the cursor position is called the point, and a few functions do most of the work:

- `(point)`: where the cursor is, as a number.
- `(goto-char 1)`: move the cursor. `(point-min)` and `(point-max)` are the start and the end of the buffer.
- `(insert "text")`: write at the cursor.
- `(buffer-string)`: the whole buffer as a string.
- `(count-words (point-min) (point-max))`: count words, which is a built-in command too.

One more is worth learning early: `save-excursion`. It runs some code and then puts the cursor back where it was, so your command doesn't leave the user lost in the file.

```elisp
(defun my-add-todo-on-top ()
"Insert a TODO line at the top of the buffer, without moving the cursor."
(interactive)
(save-excursion
(goto-char (point-min))
(insert "TODO: review this file\n")))
```

Making decisions and repeating things

`if` takes a condition, what to do when it's true and what to do when it isn't. `when` and `unless` are shortcuts for when you only need one side. `cond` is a chain of options, checked from top to bottom:

```elisp
(let ((n 7))
(cond ((< n 5) "small")
((< n 10) "medium")
(t "big"))) ; => "medium"
```

The `t` at the end works as "otherwise", since `t` is always true.

`dolist` walks through a list. This one goes over every open buffer and tells you which files have unsaved changes:

```elisp
(defun my-list-unsaved ()
"Show which file buffers have unsaved changes."
(interactive)
(dolist (b (buffer-list))
(when (and (buffer-file-name b) (buffer-modified-p b))
(message "Unsaved: %s" (buffer-name b)))))
```

The messages pile up in the `*Messages*` buffer, which you can open with `C-h e`.

A real command: quick notes

Time to put it all together. This command asks for a title, turns it into a clean file name and opens a new Org note in your notes folder, already with a title and the date.

```elisp
(defvar my-notes-directory "~/notes"
"Where `my-new-note' keeps its files.")

(defun my-slugify (title)
"Turn TITLE into a lowercase, dash-separated slug."
(let* ((s (downcase title))
(s (replace-regexp-in-string "[^a-z0-9]+" "-" s)))
(string-trim s "-" "-")))

(defun my-new-note (title)
"Create a new Org note called TITLE in `my-notes-directory'."
(interactive "sNote title: ")
(let* ((date (format-time-string "%Y-%m-%d"))
(file (expand-file-name
(concat date "-" (my-slugify title) ".org")
my-notes-directory)))
(make-directory my-notes-directory t)
(find-file file)
(when (= (buffer-size) 0)
(insert "#+title: " title "\n"
"#+date: " date "\n\n"))))

(keymap-global-set "C-c n" #'my-new-note)
```

Press `C-c n`, type “Ideas for the weekend”, and you’re editing `2026-11-06-ideas-for-the-weekend.org`, with the title and the date at the top. If the file already exists, it just opens, without writing the header twice.

A few new pieces showed up here:

- `defvar` creates a variable with a description, the polite way to define a setting.
- `let*` is like `let`, except each variable can use the ones before it. Here `s` gets rewritten in two steps.
- `replace-regexp-in-string` swaps every run of characters that aren’t letters or numbers for a dash.

- `string-trim` removes leftover dashes from the edges.
- `make-directory` with `t` creates the folder, and doesn't complain if it already exists.

Read it slowly, change something, run it again. That loop is how you learn Elisp.

Hooks: run code when something happens

A hook is a list of functions that Emacs runs at a specific moment. Every mode has one, named after the mode with `-hook` at the end. This is how you say "every time I open a text file, do this":

```elisp
(add-hook 'text-mode-hook #'visual-line-mode)
(add-hook 'prog-mode-hook #'display-line-numbers-mode)
```

The first line wraps long lines nicely in text files. The second turns on line numbers in any programming mode, since all of them build on `prog-mode`.

When you need more than one function call, use a `lambda`, which is a function without a name:

```elisp
(add-hook 'prog-mode-hook
(lambda ()
(setq-local fill-column 100)
(electric-pair-local-mode 1)))
```

`setq-local` changes the variable only in that buffer, so your Python files can have one width and your notes another.

There are hooks for almost everything: `before-save-hook`, `after-save-hook`, `kill-emacs-hook`, `emacs-startup-hook`. `C-h v` and the word "hook" will show you the list.

Advice: changing a command you didn't write

Sometimes you want a built-in command to do one extra thing. Advice lets you attach your own function before, after or around it:

```elisp
(defun my-announce-save (&rest _)
"Tell me which buffer was just saved."
(message "Saved %s" (buffer-name)))

(advice-add 'save-buffer :after #'my-announce-save)
```

Now every save shows a little message. To undo it, `advice-remove` with the same arguments.

Advice is powerful, and that's exactly why it deserves care. Reach for a hook first. If there isn't one, then advice is fine, but keep each piece small

and give it a clear name, because advice is invisible when you're reading the original command's code.

Reading Emacs's own code

This is where the investment pays off. `C-h f` describes any function, and the description has a link to the file where it's defined. Click it and you're reading the real source. `C-h k` does the same for a key: press `C-h k` and then `C-x C-s`, follow the link, and you'll see how Emacs saves a file.

Three other help commands that pair well with Elisp:

- `C-h o`: describe any symbol, function or variable, whatever it is.
- `C-h v`: describe a variable and show its current value.
- `C-h i`: the manuals, including the Elisp reference, inside Emacs.

You won't understand everything you read at first. That's fine. Skim, look for the parts you recognize, and every week you'll recognize more.

When things break

Errors in Elisp are normal, and Emacs has good tools for them.

`M-x toggle-debug-on-error` makes Emacs open a backtrace whenever an error happens. It looks dense, but the top lines usually point straight at the function that failed.

For your own functions, there's Edebug. Put the cursor inside a `defun` and press `C-u C-M-x`. The next time the function runs, Emacs stops at the first step and lets you walk through it with `SPC`, one expression at a time, showing each value as it goes. `q` gets you out.

If your `init.el` breaks so badly that Emacs starts in a strange state, remember `emacs -Q`, which starts without any config. Open the file, fix it, restart. And `emacs --debug-init` starts with your config and opens the backtrace at the first error, which usually tells you the exact line.

Finally, `M-x checkdoc` reviews the descriptions in your file and points out small style problems. It's a good teacher.

Organizing your init.el

Once you have a few functions, keep them tidy. Two simple rules go a long way.

Group things by topic, with a comment on top of each group, and keep your own functions together in one section. Use `;;;` for section titles. Emacs even understands them: `M-x outline-minor-mode` lets you fold the file by those headings.

And let `use-package` do the bookkeeping for packages and built-in features alike. It's built into Emacs since version 29:

```elisp
(use-package org
:hook (org-mode . visual-line-mode)
:bind ("C-c a" . org-agenda)
:config
(setq org-log-done 'time))
```

`:hook` adds hooks, `:bind` sets keys and `:config` runs code after the package loads. It's the same Lisp you learned above, organized into labeled boxes.

Where to go next

Inside Emacs, `C-h i` opens the manuals. Look for "An Introduction to Programming in Emacs Lisp", written for people who have never programmed. It's long, gentle and full of small exercises. When you want the details, the "Emacs Lisp Reference Manual" sits right next to it. Both are also online at gnu.org/software/emacs/manual.

A good next exercise: pick something you do by hand at least twice a week, and turn it into a command. Inserting a template, opening a set of files, cleaning up a pasted text. Small commands that fit your own habits are where Elisp starts to feel personal.

This is part II of a series. Part I is about living inside Emacs: feeds, gopher, the fediverse, chat and mail. Part III turns Emacs into a full programming setup. ■



■ ART · NOCTEM.ART

05 EMACS · PROGRAMMING · TUTORIAL 11 NOVEMBER 2026

Emacs beyond the basics, part III: a real programming setup

Turn Emacs into a modern programming setup: projects, Eglot and language servers, tree-sitter, completion, Consult and Embark, compile and debug, Git without Magit, containers and a ready-to-paste config.

Emacs has been a programming editor since before most programming languages we use today existed. For a while, though, setting it up for modern work meant a pile of third-party packages and a lot of patience. That changed. Since Emacs 29, the core pieces of a modern setup come in the box: a client for language servers, tree-sitter parsing and `use-package`, next to the project management that was already there. Emacs 30 and 31 filled in a lot of the gaps.

This post turns a plain Emacs into a comfortable place to write code. It assumes you've read the first Emacs guide, or at least that `C-x C-f` and `M-x` don't scare you anymore. Part II helps if you want to understand every line of the config, but you can follow along without it.

Everything here works on Emacs 29 and newer, unless I point out a version.

Projects

Most of programming happens inside a project, usually a Git repository. Emacs has `project.el` built in, and it figures out the project on its own: open any file inside a Git repository and Emacs knows where the project starts.

Everything lives under `C-x p`:

```
C-x p p: switch to another project
C-x p f: find a file in this project
C-x p g: search the whole project for a regexp
C-x p r: search and replace across the project
C-x p b: switch to a buffer from this project
C-x p D: open Dired at the root of the project
C-x p d: pick a folder inside the project and open it in Dired
C-x p s: open a shell at the root of the project
C-x p e: open Eshell at the root of the project
C-x p c: compile the project
```

```
C-x p !: run a shell command at the root
C-x p k: close every buffer of this project
```

`C-x p p` remembers every project you've opened. Pick one and Emacs asks what you want to do there: find a file, search, open a shell, and so on. Switching between three projects becomes a matter of a few keys.

If a folder isn't a Git repository but you still want Emacs to treat it as a project, you can teach it other markers, like a `package.json` or a `requirements.txt`, through the `project-vc-extra-root-markers` option.

Language servers with Eglot

A language server is a program that understands one language deeply: where every function is defined, what a variable's type is, what's wrong with the line you just wrote. VS Code, Neovim and Emacs all talk to the same servers, using the same protocol. Emacs's client is called Eglot, and it's built in since version 29.

First, install the server for your language. Some common ones:

```
npm install -g pyright # Python
go install golang.org/x/tools/gopls@latest # Go
rustup component add rust-analyzer # Rust
npm install -g typescript-language-server typescript # JavaScript and
TypeScript
sudo apt install clangd # C and C++, or your system's package manager
```

Then open a file and run `M-x eglot`. Eglot already knows which server goes with which language, so most of the time there's nothing to configure. For Python alone it knows about a long list of servers, from pyright and basedpyright to pylsp and ruff, and it picks the first one it finds installed.

From then on, these keys do the heavy lifting:

```
M-.: go to the definition of what's under the cursor
M-,: come back
M-?: find every place that uses it
M-x eglot-rename: rename it everywhere
M-x eglot-code-actions: quick fixes the server suggests
M-x eglot-format-buffer: format the file
M-x flymake-show-buffer-diagnostics: list every error and warning
```

The errors appear underlined as you type, through Flymake, which is also built in. The documentation of the thing under the cursor shows up at the bottom of the screen, through ElDoc. Neither needs setup.

To start Eglot automatically, add it to the hooks of the modes you use:

```elisp
(use-package eglot
:hook ((python-ts-mode go-ts-mode rust-ts-mode) . eglot-ensure)
:bind (:map eglot-mode-map
("C-c e r" . eglot-rename)
("C-c e a" . eglot-code-actions)
("C-c e f" . eglot-format-buffer)))
```

The `:bind` part gives the three most useful commands short keys, under `C-c e`, only when Eglot is active. And if your server isn't on Eglot's list, you add it yourself:

```elisp
(with-eval-after-load 'eglot
(add-to-list 'eglot-server-programs
'(lua-mode . ("lua-language-server"))))
```

Tree-sitter: syntax that understands the code

Old Emacs modes highlighted code with regular expressions, which works most of the time and breaks in strange ways the rest. Tree-sitter parses the file into a real syntax tree, like a compiler would. The result is faster, more accurate highlighting and indentation, and smarter movement through the code.

Each language has a mode with `-ts-` in the name: `python-ts-mode`, `go-ts-mode`, `rust-ts-mode`, `typescript-ts-mode`, `tsx-ts-mode`, `js-ts-mode`, `json-ts-mode`, `bash-ts-mode`, `yaml-ts-mode` and many more. Each one needs a grammar, a small library for that language.

On Emacs 31, this got a lot easier. One line tells Emacs to prefer the tree-sitter modes:

```elisp
(setopt treesit-enabled-modes t)
```

When a grammar is missing, Emacs 31 asks if you want to install it, and does the work for you. That's the default behavior, controlled by the `treesit-auto-install-grammar` option. Building a grammar needs a C compiler on your system, so have `gcc` or `clang` around.

On Emacs 29 and 30, it takes two steps. Install the grammar with `M-x treesit-install-language-grammar`, then tell Emacs to use the new mode instead of the old one:

```elisp
(add-to-list 'major-mode-remap-alist '(python-mode . python-ts-mode))
```

People on those versions often use a package from MELPA called `treesit-auto` to automate this. On Emacs 31 you don't need it anymore.

Completion while you type

Completion in the minibuffer was covered in the first guide, with Vertico, Orderless and Marginalia. Completion inside the code is a separate thing, and there are two good paths.

The built-in one, since Emacs 30, is `completion-preview-mode`. It shows the most likely completion in gray right after the cursor, and `TAB` accepts it. It's quiet and light.

If you prefer the classic popup with a list of options, use Corfu, from GNU ELPA:

```elisp
(use-package corfu
:ensure t
:custom (corfu-auto t)
:init (global-corfu-mode))
```

With `corfu-auto`, the list shows up on its own while you type. Eglot feeds it, so you get the same completions as any other editor using that language server.

Finding things

Programming is mostly reading, and reading is mostly searching. Consult, from GNU ELPA, gives Emacs a set of search commands with live previews:

```elisp
(use-package consult
:ensure t
:bind (("C-x b" . consult-buffer)
("M-s l" . consult-line)
("M-s r" . consult-ripgrep)
("M-g i" . consult-imenu)))
```

- `consult-buffer` replaces the buffer switcher and also lists recent files and bookmarks.
- `consult-line` searches the lines of the current file and jumps as you move through the results.
- `consult-ripgrep` searches the whole project with ripgrep, so install `rg` first.
- `consult-imenu` lists every function, class and heading in the file. Without Consult, the built-in `M-x imenu` does the same in a simpler way.

Embark, also from GNU ELPA, is the companion. Press its key on anything, a file in a list, a symbol in the code, a search result, and it shows what you can do with it: open, rename, copy, delete, search for it. If you use Consult and Embark together, also install `embark-consult`, which connects the two.

```elisp
(use-package embark
:ensure t
:bind ("C-." . embark-act))
```

`C-.` works in graphical Emacs. In a terminal, pick another key.

One gem in Emacs 31: search results became editable. Run `M-x rgrep` or `M-x grep`, press `e` in the results buffer, and edit the lines right there. Your changes go straight into the files. `C-c C-c` finishes. Renaming a word across forty files becomes a search and a couple of keystrokes.

Compile, test and jump to the errors

`M-x compile` runs any command, `make`, `cargo build`, `go test ./...`, `npm test`, and shows the output in a buffer. Error messages become links: press `RET` on one and Emacs opens the file at that line. `C-x p c` does the same from the root of the project.

```
M-x compile: run a build or test command
M-x recompile: run the same command again
M-g n: jump to the next error
M-g p: jump to the previous error
```

`C-x` followed by the backquote key also jumps to the next error. Set `compilation-scroll-output` to `'first-error` and the output stops scrolling at the first problem, which is usually the one you want to read.

Running code interactively

Many languages have a REPL integration. Python is a good example of how they work:

```
C-c C-p: start a Python shell (run-python)
C-c C-c: send the whole file to it
C-c C-z: jump between the code and the shell
```

The same idea exists for Lisp, Ruby, Scheme, shell scripts and others. Write some code, send it, look at the result, adjust. It's a different rhythm from save and run, and a pleasant one.

Debugging

For C, C++ and other languages GDB understands, Emacs has a full interface built in. Run `M-x gdb`, then `M-x gdb-many-windows`, and you get the source, the variables, the stack, the breakpoints and the GDB console laid out together.

For everything else, there's Dape, from GNU ELPA. It speaks the Debug Adapter Protocol, the same one VS Code uses for its debugger, so it works with debugpy for Python, delve for Go, codelldb for Rust and C++, and the JavaScript debuggers. Install the adapter for your language, run `M-x dape` and choose a configuration.

Git, besides Magit

Magit is still the star, and the first guide covers it. But Emacs also has a built-in version control interface, VC, which works with Git, Mercurial and others, and has quick keys for small jobs:

```
C-x v v: do the next logical thing (add a new file, commit changes)
C-x v =: diff of the current file
C-x v D: diff of the whole repository
C-x v l: log of the current file
C-x v g: who wrote each line (blame)
```

When a merge goes wrong, open the file with conflicts and Emacs turns on `smerge-mode` by itself. Its commands live under `C-c ^`: `C-c ^ n` goes to the next conflict, `C-c ^ u` keeps your version (the upper one), `C-c ^ l` keeps the other one (the lower one) and `C-c ^ a` keeps both.

And for a quiet reminder of what you've changed, `diff-hl` marks changed lines in the margin:

```
(use-package diff-hl                                                ELISP
:ensure t
:init (global-diff-hl-mode))
```

Small helpers that add up

A few built-in modes that are worth turning on in code buffers:

- `electric-pair-mode` closes brackets and quotes for you.
- `subword-mode` makes `M-f` and `M-b` stop inside `camelCase` words.
- `hs-minor-mode` folds functions and blocks. Its keys live under `C-c @`, and `C-c @ C-h` hides the block you're in.
- `which-function-mode` shows the name of the current function in the mode line.
- `whitespace-mode` shows tabs and trailing spaces, for when the linter complains about something invisible.
- `display-line-numbers-mode`, with `display-line-numbers-type` set to `'relative` if you like counting jumps.
- `editorconfig-mode`, built in since Emacs 30, follows the `.editorconfig` file that many projects have, so your indentation matches the team's.

Turning them on for every programming mode is one hook each:

```elisp
(add-hook 'prog-mode-hook #'display-line-numbers-mode)
(add-hook 'prog-mode-hook #'hs-minor-mode)
(add-hook 'prog-mode-hook #'subword-mode)
(add-hook 'prog-mode-hook #'electric-pair-local-mode)
(editorconfig-mode 1) ; Emacs 30 and newer
```

For snippets, the classic package is YASnippet, from GNU ELPA. Emacs also ships simpler tools for the same job, like abbrevs: `M-x define-global-abbrev`, type a short word and its expansion, and it expands as you type once `abbrev-mode` is on.

Code inside containers

TRAMP, which the first guide used for SSH, also opens files inside containers since Emacs 29:

```
/docker:container-name:/app/main.py
/podman:container-name:/app/main.py
```

Dired, shells and `M-x compile` all work in those paths, so you can browse and build inside a container without leaving Emacs.

Putting it together

Here's the programming part of a config, in one block, ready to paste after the config from the first guide:

```elisp
;;; Programming

(use-package eglot
:hook ((python-ts-mode go-ts-mode rust-ts-mode) . eglot-ensure)
:bind (:map eglot-mode-map
("C-c e r" . eglot-rename)
("C-c e a" . eglot-code-actions)
("C-c e f" . eglot-format-buffer)))

(use-package corfu
:ensure t
:custom (corfu-auto t)
:init (global-corfu-mode))

(use-package consult
:ensure t
:bind (("C-x b" . consult-buffer)
("M-s l" . consult-line)
("M-s r" . consult-ripgrep)
("M-g i" . consult-imenu)))

(use-package embark
```

```
:ensure t
:bind ("C-." . embark-act))

(use-package embark-consult
:ensure t)

(use-package diff-hl
:ensure t
:init (global-diff-hl-mode))

(setq compilation-scroll-output 'first-error)

(add-hook 'prog-mode-hook #'display-line-numbers-mode)
(add-hook 'prog-mode-hook #'hs-minor-mode)
(add-hook 'prog-mode-hook #'subword-mode)
(add-hook 'prog-mode-hook #'electric-pair-local-mode)

;; Emacs 30 and newer
(editorconfig-mode 1)

;; Emacs 31: prefer tree-sitter modes and offer to install grammars
(setopt treesit-enabled-modes t)
```

On Emacs 29, remove the editorconfig line. On Emacs 29 and 30, remove the tree-sitter line at the end and use `major-mode-remap-alist` for tree-sitter, as shown above. As always, select the block and press `TAB` to let Emacs indent it nicely.

A sane way to adopt all this

Don't paste everything on a Monday morning before a deadline. Add one piece at a time, in this order: projects, then Eglot, then the search commands, then the rest. Live with each one for a few days. The pieces you actually use will stick, and the ones you don't can go.

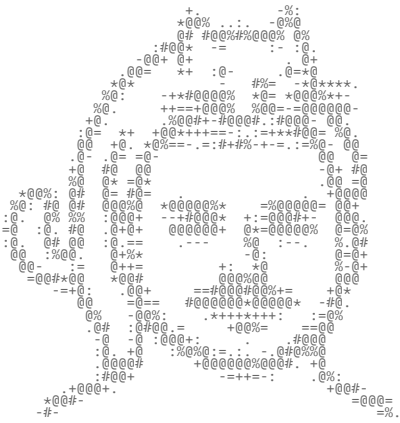
This is part III of a series. Part I is about living inside Emacs: feeds, gopher, the fediverse, chat and mail. Part II teaches enough Emacs Lisp to bend all of this to your taste. ■



~pablo

A zine made of what I wrote on pablomurad.com.

01



COLOPHON

ISSUE	N° 01, special edition: Emacs and Org mode, October – November 2026
PUBLISHED	18 November 2026, on pablomurad.com
TYPESET	18 November 2026, A5
ESSAYS	5, each one also on the website
PLATES	Pictures between the essays are courtesy of noctem.art.
AT HOME	Print pablo-01-booklet.pdf double-sided on A4, flip on the short edge, fold the stack in half and staple the spine.

pablomurad.com/zine/01

~PABLO

Nº 01

Essays from pablomurad.com, set on paper.

