Cursor AI: The Complete Beginner-to-Pro Guide

A practical, no-fluff guide to using Cursor as your everyday AI coding editor. Built for newcomers; useful for experts.

What is Cursor?

Cursor is a code editor supercharged with AI. It is designed to:

1 Understand your codebase and propose accurate edits across multiple files

1 Generate, refactor, and explain code with context from your repository

1 Review and improve code with human-readable diffs you can accept or reject

1 Speed up your workflow with inline completions, chat, and code-aware commands

You get the familiarity of a modern editor plus an AI pair programmer that sees your project.

Quick Start

1 Install the Cursor app and sign in.

2 Open a folder or clone a repository.

3 Let Cursor index your codebase so AI can reference your project.

4 Start a chat and describe a concrete task. Mention files or select code so Cursor has context.

5 Review proposed edits as diffs. Accept, tweak, or ask for alternatives.

Pro tip: Be specific. Point Cursor to files, selections, errors, or test failures. The more grounded the context, the better the result.

Interface Tour

1 **Editor**: Your main code view. Inline completions and AI edits appear here.

1 **Sidebars**: File explorer, search, Git, problems, outline, and extensions.

1 **Chat**: Code-aware conversation. Reference files, selections, errors, or tests.

1 **Composer/Actions**: Quick AI actions like Refactor, Explain, Add tests, Fix.

1 **Diff View**: AI edits appear as a reviewable diff. Apply all or per-file.

1 **Terminal**: Run commands, capture logs, and ask Cursor to fix issues from traces.

1 **Command Palette**: Discover everything. Search for commands and Cursor actions.

Core AI Features

1) Code-Aware Chat

1 Ask questions, generate code, or request edits.

1 Reference files, symbols, and selections so the model focuses on the right scope.

1 Get multi-file edits as a diff you can inspect and apply.

Common asks:

1 "Explain how `auth` middleware validates JWTs."

1 "Add input validation to `api/users.ts` and update tests."

1 "Refactor these functions to be pure and add unit tests."

2) Inline Completions

1 Type, and accept smart suggestions inline.

1 Cursor learns from your project style, libraries, and patterns.

1 Use completions for boilerplate, mapping types, data transforms, and repetitive code.

3) Selection-Based Actions

Highlight code and run focused actions:

1 Explain or document a function

1 Refactor, extract, or rename safely

1 Add types, improve performance, simplify logic

1 Generate unit tests and usage examples

4) Apply Edits as Diffs

1 Cursor proposes changes as reviewable diffs.

1 Inspect changes file-by-file. Accept, reject, or ask for alternatives.

1 For large changes, request smaller, incremental edits to stay in control.

5) Error- and Test-Driven Fixes

1 Paste an error trace or failing test output; ask Cursor to fix the root cause.

1 Cursor can point to likely files, propose diffs, and explain the reasoning.

Context and Codebase Awareness

1 **Repository Indexing**: Cursor indexes your code so the model can search by meaning, not just text.

1 **Selections and Open Files**: What you highlight and what is open is high-signal context.

1 **File and Symbol References**: Point to files, folders, or symbols to anchor the task.

1 **Limits**: Very large repos may require scoping. Work file-by-file or feature-by-feature.

Tips:

1 Quote relevant snippets or select code before asking.

1 Mention specific files and tests by path.

1 Keep conversations focused on one task at a time.

Smart Editor Actions (Examples)

1 **Explain**: Clarify a function, module, or architecture.

1 **Refactor**: Improve naming, extract functions, reduce complexity.

1 **Optimize**: Remove redundant work, reduce allocations, batch I/O.

1 **Type**: Add or tighten types; infer generics; remove `any`.

1 **Docs**: Generate docstrings and usage examples.

1 **Tests**: Create or expand unit/integration tests with realistic fixtures.

1 **Fix**: Resolve linter errors, runtime exceptions, or failing tests.

1 **Migrate**: Upgrade frameworks or libraries safely with codemods.

Git and PR Workflows

1 Stage only what you intend to commit; keep AI edits reviewable.

1 Ask Cursor to summarize diffs and suggest commit messages.

1 Use AI to review your changes for edge cases and performance.

1 For PRs, request a structured review: correctness, tests, readability, risks.

Check list before merging:

1 Are tests updated or added?

1 Are breaking changes documented?

1 Is public API typed and documented?

Working with the Terminal and Logs

1 Capture failing command output and ask Cursor to diagnose.

1 Provide enough surrounding context (inputs, env vars, versions).

1 Ask for a minimal fix first; then for a hardening pass and tests.

Prompting Playbook

Make requests concrete, scoped, and testable.

Good patterns:

1 "Given this selection, extract a pure function and return type."

1 "Implement this feature in `app/routes/user.tsx`; update affected tests."

1 "These tests fail; fix the underlying bug and keep behavior stable."

1 "Refactor to remove side effects; preserve API."

Include:

1 Goal and constraints

1 Target files and functions

1 Inputs/outputs and edge cases

1 Performance or security requirements

Settings to Review Early

1 Indexing options for large repos

1 Telemetry and cloud features per your org policy

1 Keybindings for chat, actions, and accept completion

1 Default language/formatter/linter integrations

Troubleshooting

1 **Model can’t find a symbol**: Reference the file or open it so it’s indexed and in context.

1 **Edits are too broad**: Ask for smaller, incremental diffs.

1 **Completions feel off**: Provide examples or accept a few suggestions to steer style.

1 **Chat lost the thread**: Start a fresh chat focused on a single task.

Security and Privacy

1 Never paste secrets. Use local environment variables and secret managers.

1 Sanitize production data before sharing logs.

1 Keep edits reviewable; run tests locally before committing.

Example Workflows

Implement a Feature

1. Describe the spec and affected files.
2. Ask Cursor to scaffold the implementation and tests.
3. Review the diff; request changes where needed.
4. Run tests; iterate on failures with Cursor.

Fix a Bug from a Trace

1. Paste the stack trace and reproduction steps.
2. Ask Cursor to identify the root cause and propose a minimal fix.
3. Request a follow-up hardening pass and tests for edge cases.

Refactor Safely

1. Select the code; state the desired outcome (e.g., extract, simplify, type).
2. Apply the diff and run tests.
3. Ask Cursor to scan for related call sites or dead code.

FAQ

1 **Is Cursor a replacement for tests?** No. Use it to write better tests faster.

1 **Can it change many files at once?** Yes, but keep changes scoped and review diffs.

1 **How do I get better results?** Provide concrete context, constraints, and examples.

1 **What if I disagree with an edit?** Reject it and explain what to keep/change.

Further Resources

1 Explore your editor’s command palette to find Cursor-specific commands

1 Read official docs and community tips for advanced workflows

1 Join community channels to learn real-world prompting patterns

Thanks for using Cursor. Treat it like a focused pair programmer: give it context, review its work, and iterate quickly.