START

The Claude Code Starter Guide

Free guide. No signup. 10 min read

Checked 2026-10-28 against Anthropic's Claude Code docs (code.claude.com/docs) and Claude Academy courses (academy.claude.com).

From @learningwithcole

This takes you from nothing installed to a setup you can use every day. Every command and claim here comes from Anthropic's Claude Code docs and their Claude Academy courses, checked on 2026-10-28. It's my own unofficial guide, built from their material. Claude Code changes fast, so if this page and the docs disagree, trust the docs.

1. What Claude Code is

Claude Code is an agentic coding tool that, in the docs' words, "reads your codebase, edits files, runs commands, and integrates with your development tools." You describe what you want in plain language. Claude plans the approach, writes the code across files, and checks that it works.

It runs in your terminal, VS Code, JetBrains IDEs, a desktop app, and the browser at claude.ai/code. This guide uses the terminal.

2. Install it and sign in

You need an account. The setup docs say (checked 2026-10-28): "Claude Code requires a Pro, Max, Team, Enterprise, or Console account. The free claude.ai plan does not include Claude Code access."

The native installer is the recommended method:

Terminal

```
# macOS, Linux, WSL
curl -fsSL https://claude.ai/install.sh | bash
```

powershell

```
# Windows PowerShell
irm https://claude.ai/install.ps1 | iex
```

Prefer a package manager? Use `brew install --cask claude-code` on macOS or `winget install Anthropic.ClaudeCode` on Windows. Those two don't auto-update. The native install updates itself in the background.

Open a new terminal window and check it worked:

Terminal

```
claude --version
```

You should see a version number. If your shell says `claude` isn't found, the install folder isn't on your PATH yet: see Fix your PATH in the docs. For a fuller check, run `claude doctor`. On native Windows, the docs recommend Git for Windows so Claude Code can use its Bash tool.

3. Your first session in a real repo

Terminal

```
cd /path/to/your/project
claude
```

The first time, Claude Code asks you to log in through your browser. Your credentials are then stored. To switch accounts later, type `/login`.

Start with questions. Claude reads your files as it needs them, so you don't have to paste code in.

Prompt

```
what does this project do?
```

Prompt

explain the folder structure

Then try a small change:

Prompt

add a hello world function to the main file

On current versions, new terminal sessions start in **auto mode** when your account and model support it. Claude works without routine permission prompts, and a separate classifier model checks risky actions before they run. If Claude does ask before a change, pick **Yes**. Section 5 covers the other modes.

Five things to know from day one:

- Type `/` to see the commands and skills you have. `/help` lists them too.
- Type `@` and a file path to put that file's contents in your message.
- Start a line with `!` to run a shell command yourself. Its output goes into the session and Claude responds to it.
- Press `Esc` to stop Claude mid-action. It keeps the work done so far, so you can redirect it.
- `claude -c` continues your most recent conversation in that folder. `claude -r` lets you pick an older one.

4. Give it a memory: CLAUDE.md and /init

Every session starts with a fresh context window. CLAUDE.md is a markdown file Claude reads at the start of every session, so it's where you write down what you'd otherwise re-explain.

Run this once in your project:

Text

```
/init
```

Claude analyzes your codebase and writes a starting CLAUDE.md with the build commands, test instructions and conventions it finds. If you already have one, `/init` suggests improvements instead of overwriting it.

Then trim it. Anthropic's best-practices page points out that Claude already finds your file layout and conventions by reading the code. Keep what it can't see: exact test and lint commands, misleading corners of the code, decisions that never made it into the repo, and anything you've corrected twice. Their example:

Markdown

```
# Commands
- Install: pnpm install
- Run one test file: pnpm vitest run path/to/file.test.ts
- Typecheck: pnpm tsc --noEmit

# Conventions
- Use the error helpers in src/lib/errors.ts, never throw raw strings
- src/legacy/ is still live but don't copy its patterns into new code

# Gotchas
- Integration tests need DATABASE_URL set, see .env.example
```

Rules of thumb from the memory docs:

- Aim for under 200 lines. Longer files use more context and get followed less reliably.
- Write instructions you could check: "Run `npm test` before committing", not "Test your changes".
- Personal preferences for one project go in `CLAUDE.local.md` (add it to `.gitignore`). Preferences for all your projects go in `~/.claude/CLAUDE.md`.

`/memory` opens your CLAUDE.md files. It also shows auto memory, the notes Claude keeps from your corrections, and lets you turn it off. One limit: Claude treats CLAUDE.md as context, not enforced configuration. A rule that must never bend belongs in a hook or a deny rule.

5. Permission modes and plan mode

A permission mode sets what Claude can do without asking you first. Press `Shift+Tab` to cycle through them. The status bar shows the active one.

Mode	What runs without asking
Manual (`default`)	Reads only
`acceptEdits`	Reads, file edits, and common filesystem commands like `mkdir`, `mv` and `cp`
`plan`	Reads, plus classifier-approved commands when auto mode is available. No source edits until you approve
`auto`	Everything, with background safety checks by a classifier
`bypassPermissions`	Everything. The docs say: isolated containers and VMs only

The docs also warn that auto mode "reduces permission prompts but does not guarantee safety." If you'd rather approve everything yourself, start with `claude --permission-mode default`, or make it your default in `~/.claude/settings.json`:

JSON

```
{
  "permissions": {
    "defaultMode": "default"
  }
}
```

Plan mode is the one to learn first. Claude reads files, runs shell commands to explore, and writes a plan, but it doesn't edit your source until you approve. Three ways in:

- Press `Shift+Tab` until the status bar shows `⏸ plan mode on`
- Put `/plan` in front of one prompt: `/plan fix the auth bug`
- Start in it: `claude --permission-mode plan`

When the plan is ready, pick **Yes, and use auto mode** (it reads **Yes, auto-accept edits** if auto mode isn't available), **Yes, manually approve edits**, or **No, keep planning**. Press `Ctrl+G` to open the plan in your text editor and change it before Claude starts.

Deeper guide: /plan

6. Undo with checkpoints

Claude Code saves a checkpoint of your code before each prompt you send. To go back, press `Esc` twice on an empty prompt, or run `/rewind`. Pick the message to go back to, then choose:

- **Restore code and conversation**: both go back to that point
- **Restore conversation**: rewind the chat, keep the current code
- **Restore code**: roll back the files, keep the chat
- **Summarize from here** or **Summarize up to here**: shrink part of the chat without touching files

Checkpoints track edits Claude makes with its own file tools. A file that only a Bash command changed isn't restored, and neither are edits you make outside Claude Code. The docs say they're no replacement for version control, so commit before anything big:

Prompt
commit my changes with a descriptive message

Deeper guide: /undo

7. Manage context

The context window holds everything Claude knows about your session: your instructions, the files it reads, and its replies. Claude Code compacts the conversation on its own near the limit, so running out is rare. You still pay for context you don't need, because every request sends the whole conversation.

Command	Use it when
`/clear`	You're switching to unrelated work. It starts a new conversation with empty context, and `/resume` brings the old one back
`/compact focus on the auth bug fix`	Same task, long chat. It summarizes the conversation and keeps what you name

Command	Use it when
`/context`	You want to see what's filling the window, as a colored grid
`/usage`	You want session cost, plan usage limits and activity stats
`/btw` + a question	You have a quick side question that shouldn't go into the conversation
`/rename` + a name	The work spans days. `claude --resume <name>` picks it up later

Anthropic's own advice for steering: watch the first minute, press `Esc` the moment Claude drifts, and after two failed corrections run `/clear` and write a better prompt instead of a third.

Deeper guide: /context

8. The daily loop: explore, plan, code, commit

This is the workflow Anthropic teaches in its Claude Code 101 course.

1. Explore. Switch to plan mode and describe the task. Claude reads the relevant files and works out where the change belongs.

Prompt

I need to add WebP conversion to our image upload pipeline. Work out where the change belongs, whether we need new dependencies, and how you'd approach it.

2. Plan. Review the plan, then approve it or ask for changes. This is the best place to correct course, because no code exists yet. Put your success criteria in the plan so Claude knows what "correct" means.

3. Code. Approve and let Claude work through the steps. Give it a check it can run, like a test file, a build or a linter. The best-practices page puts it simply: "Give Claude something it can run that comes back pass or fail." Ask for evidence too: the command it ran and its output.

Prompt

run the auth tests after, and keep going until they pass

4. Commit. Test the change yourself first. The course suggests a subagent code reviewer next, since it hasn't seen how the code was written. In the terminal, `/code-review` reviews your diff in a fresh subagent. Then ask Claude to write the commit message in your style.

Then `/clear` and start the next task.

Be specific. Instead of "fix the login bug", the docs suggest:

Prompt

users report login fails after the session times out. look at token refresh in src/auth/, write a failing test that reproduces it, then fix it

Not sure what you want yet? Have Claude interview you:

Prompt

I want to add rate limiting to the public API but haven't thought it through yet. Interview me until you could write the spec, then write it to SPEC.md.

9. Extend it: MCP, skills, hooks, plugins, subagents

You don't need all of this on day one. The docs suggest adding each piece when its trigger shows up:

When this happens	Add
Claude gets a convention or command wrong twice	A line in CLAUDE.md
You keep typing the same prompt to start a task	A skill
You keep copying data from a browser tab Claude can't see	An MCP server

When this happens	Add
A side task floods your conversation with output	A subagent
You want something to happen every time without asking	A hook
A second repo needs the same setup	A plugin

MCP servers connect Claude to outside tools like issue trackers, databases and design files. Add one from your shell:

Terminal

```
claude mcp add --transport http notion https://mcp.notion.com/mcp
```

Run `/mcp` in a session to see it and sign in. By default a server is private to you, in that project. `--scope project` writes it to a `.mcp.json` your team can share. The docs say to verify you trust each server before connecting it. Deeper guide: /mcp

Skills are saved instructions. A skill is a folder with a `SKILL.md` file, in `~/.claude/skills/<name>/` for all your projects or `.claude/skills/<name>/` for one repo. Type `/<name>` to run it, or Claude loads it on its own when your request matches its `description`. The docs' first-skill example, saved as `~/.claude/skills/summarize-changes/SKILL.md`:

Markdown

```
---
description: Summarizes uncommitted changes and flags anything risky. Use when the user
asks what changed, wants a commit message, or asks to review their diff.
---

## Current changes

!`git diff HEAD`

## Instructions

Summarize the changes above in two or three bullet points, then list any risks you
notice such as missing error handling, hardcoded values, or tests that need updating.
If the diff is empty, say there are no uncommitted changes.
```

The `!`git diff HEAD`` line runs first, so Claude sees your live diff. Test it with "What did I change?" or `/summarize-changes`. Design skills I'd install: /design

Hooks are commands Claude Code runs at set moments, every time, so Claude can't skip them. This one from the docs runs Prettier on every file Claude edits. It needs `jq`. Put it in `.claude/settings.json` in your project:

JSON

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.file_path' | xargs npx prettier --write"
          }
        ]
      }
    ]
  }
}
```

Run `/hooks` to check it loaded. Deeper guide: /hooks

Plugins bundle skills, hooks, subagents and MCP servers into one install. Claude Code adds Anthropic's official marketplace the first time you start an interactive terminal session. To install the docs' example plugin, which adds commands for committing, pushing and opening pull requests:

Text

```
/plugin install commit-commands@claude-plugins-official
```

In a session, that opens the plugin's details first, so you can see what it adds and pick a scope. A plugin can run hooks and MCP servers, so read that pane before you install. Deeper guide: /plugins

Subagents are helpers that work in their own context window and send back a summary. Claude already delegates to them on its own, for example broad codebase searches go to the built-in Explore agent. You can also ask directly:

Prompt

use a subagent to investigate how our auth system handles token refresh

A custom subagent is a markdown file in `.claude/agents/` with a name, a description and, optionally, a tool list. The best-practices page shows a reviewer limited to `tools: Read, Grep, Glob`, so it can read your code but can't edit it.

Subagents cost tokens: each one sends its own requests, which count toward the same usage limits as your main conversation. `Ctrl+B` sends a running command or agent to the background. Deeper guide: /background

10. Where to learn more

- **The docs:** code.claude.com/docs. Read Quickstart, Best practices and Common workflows first.
- **Claude Academy:** Anthropic's free self-paced courses at academy.claude.com. Start with Claude Code 101, then Claude Code in Action for longer, hands-off sessions.
- **Claude itself:** ask "what can Claude Code do?" in a session.
- **The repo:** github.com/anthropics/claude-code for issues and plugins.

Good to know

- Permission rules are enforced by Claude Code, not by the model. A deny rule blocks in every mode, including `bypassPermissions`. Manage rules with `/permissions` or in settings:

JSON

```
{
  "permissions": {
    "allow": ["Bash(npm run *)", "Bash(git commit *)"],
    "deny": ["Bash(git push *)"]
  }
}
```

- If you start a terminal session with `--dangerously-skip-permissions` or `--permission-mode bypassPermissions`, plan mode isn't enforced: Claude is told to plan, but an

edit it attempts runs without asking.

- When auto mode blocks something, it shows up under **Recently denied** in `/permissions`. Three blocks in a row, or 20 in total, pause auto mode and it goes back to asking you.
- `claude -p "query"` runs one query and exits, so it works in scripts and pipes.

Links this guide cites

code.claude.com/docs/en/troubleshoot-install#command-not-found-claude-after-installation

code.claude.com/docs/en/overview

code.claude.com/docs/en/quickstart

code.claude.com/docs/en/best-practices

code.claude.com/docs/en/common-workflows

academy.claude.com

academy.claude.com/courses/claude-code-101

academy.claude.com/courses/claude-code-in-action

github.com/anthropics/claude-code