

1. Introduction to Cursor AI Code Editor Setup and Features

Clip source: [Introduction to Cursor AI Code Editor Setup and Features](#)

[Home](#)

>
[Courses](#)

>
[Code Smarter with Cursor AI Editor](#)

Introduction to the Course

Explore how to install and navigate Cursor, the AI-enhanced code editor built on Visual Studio Code. Learn to use its natural language tools to write, debug, and improve code. Understand Cursor's chat modes and features that fit seamlessly into your existing workflow, preparing you for advanced AI-driven development tasks throughout the course.

Welcome to this course on Cursor, the AI-powered code editor built on top of Visual Studio Code.

This course is designed to help developers work more efficiently by integrating AI directly into their coding environment. From writing and refactoring code to managing projects and pushing to GitHub, we'll explore how Cursor enhances everyday development workflows.

Cursor includes multiple chat modes: Agent, Ask, and Manual. These give developers the flexibility to work with AI differently, whether they prefer hands-on edits, guided suggestions, or simple explanations.

The screenshot shows the Cursor AI-powered code editor interface. On the left, there's a file tree for a Django project named 'WORDLE_DJANGO'. The main area displays a file named 'wordle.html' with code related to a word game. A red box highlights a section of the code where the AI is suggesting changes. To the right of the code editor, there are two AI assistance panels. The top panel is titled 'Update view.py for Six Word Game' and says 'I'll help you update the views.py file to include a six-word game. First, let me check the current content of the views.py file.' The bottom panel is also titled 'Update view.py for Six Word Game' and says 'I'll help you update the views.py file to include a six-word game. I'll create a new view and modify the existing code to support both 5-letter and 6-letter word games. Here's how we'll update the file:' followed by a snippet of Python code. At the bottom of the interface, there are tabs for 'Problems', 'Output', 'Debug Console', 'Terminal', and 'Ports'. The status bar at the bottom right shows the command line prompt 'Asmat@Asmat-Batool-TCE-Lead wordle_django %'.

Cursor: AI-powered code editor

Who this course is for#

This course is designed for software developers comfortable writing code and using a code editor. It is especially helpful for those looking to enhance their workflow with AI assistance.

It's a great fit for:

- Frontend and full stack developers who want to write, refactor, and debug code using natural language tools more efficiently.
- VS Code users who want to try an AI-powered alternative without changing how they work.
- Python, JavaScript, or TypeScript developers building modern applications.
- Developers who are curious about AI and interested in integrating it into their daily workflow.
- Anyone who wants to reduce time spent on boilerplate, debugging, and repetitive tasks.

No prior experience with Cursor or other AI tools is required. Just bring your coding experience and a desire to work smarter.

What you'll learn#

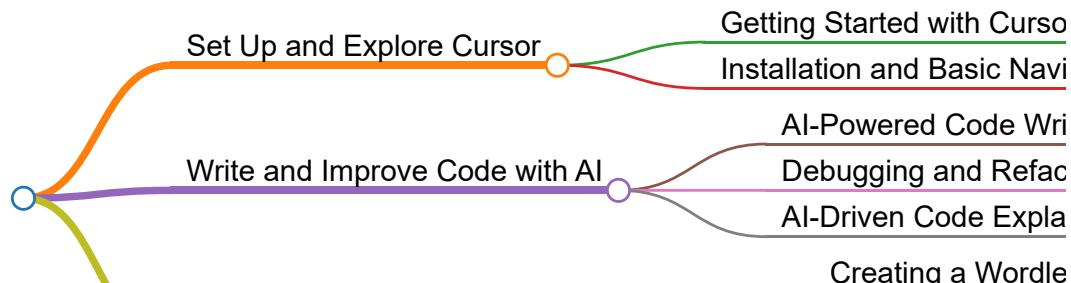
By the end of this course, you'll be able to:

- Set up Cursor and navigate its core interface.
- Use natural language prompts to write and edit code.
- Debug, refactor, and improve code using context-aware assistance.
- Explain and document existing code automatically.
- Build a complete Django-based Wordle game using AI-driven development.
- Push code to GitHub using integrated Git features within Cursor.
- Apply best practices to stay focused and productive while working with AI.

Course structure#

This course has three chapters, from setup and exploration to hands-on coding and real-world application.

- **Set Up and Explore Cursor:** We'll begin by installing Cursor, understanding, and getting familiar with its interface, settings, and core navigation.
- **Write and Improve Code with Curser AI Tools:** This chapter focuses on using Cursor's AI tools to generate, understand, and enhance code. We'll work with interactive chat, inline suggestions, and debugging support to improve code quality and productivity.
- **Build and Ship Projects with Cursor:** We'll apply what we've learned to build a real application and explore features that support larger workflows. This chapter shows how Cursor fits into a complete development cycle, from planning and testing to version control and productivity strategies.



Let's dive in!

2. Getting Started with Cursor AI Code Editor

Clip source: [Getting Started with Cursor AI Code Editor](#)

[Home](#)

>

[Courses](#)

› [Code Smarter with Cursor AI Editor](#)

Getting Started with Cursor

Explore how Cursor builds on Visual Studio Code with integrated AI tools to assist in writing, debugging, and refactoring code. Understand the core features that make Cursor a modern development environment and compare it to other AI coding assistants. This lesson helps you get familiar with Cursor's interface and learn why developers are increasingly adopting it for efficient coding workflows.

This lesson will explore how Cursor compares to traditional code editors and what makes it different from other AI-powered tools. We'll also walk through its key capabilities to understand how it fits into our workflow and why it's gaining adoption among developers.

By the end of this lesson, we'll be able to:

- Describe what Cursor is and how it builds on Visual Studio Code.
- Understand the role of AI models in Cursor's code assistance.
- Compare Cursor with other tools like GitHub Copilot and Windsurf.
- Identify the core features that make Cursor a modern development environment.
- Recognize when and why developers are choosing Cursor in real-world teams.

What is Cursor?#

Cursor is an AI-powered desktop code editor built on the open-source version of Visual Studio Code. It preserves the core experience of VS Code, including file navigation, the command palette, settings, and extension support. In addition, it adds integrated AI features to assist with everyday coding tasks.

These features include natural language code generation, context-aware completions, inline explanations, automated refactoring, and a built-in assistant that understands the project we are working on.



By combining the reliability of VS Code with intelligent AI tools, Cursor gives us a more productive and less repetitive way to write and maintain code.

Behind the scenes, Cursor connects to large language models (LLMs) to enhance its AI features, and we can select which models to use. We are not locked into a single provider. Whether GPT-4, Claude, Gemini, or a self-hosted model, Cursor supports flexible configuration based on our preferences for performance, privacy, latency, or cost.

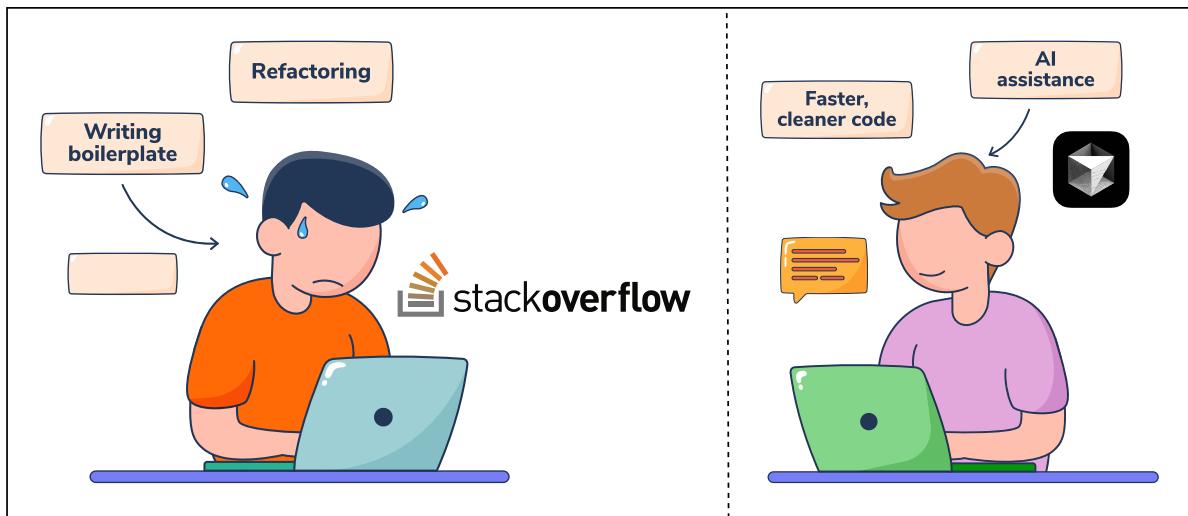
These models enable Cursor's intelligent features, such as code generation, explanation, refactoring, and in-context chat, all tailored to the project we're working on. We'll learn how to select and manage models in the next lesson, but it's helpful to know up front that Cursor gives us full control over the AI engine behind our workflow.



Security and privacy: Privacy Mode disables external API calls for code, ensuring nothing leaves your local environment. This supports strict enterprise data policies and local-only workflows.

Why should developers care#

As developers, we spend much of our time on tasks like writing boilerplate, debugging tricky errors, or restructuring messy functions. Traditional editors like Visual Studio Code help with syntax highlighting, basic autocompletion, and file navigation, but they leave most of the heavy lifting to us.



AI-assisted coding makes development faster and less frustrating compared to traditional methods. Cursor changes that. It uses AI to understand our code in context and assist with real development work. Instead of just editing text, we get a coding partner that helps us write, fix, and improve code more efficiently.

Here's what makes Cursor worth our attention:

- **Faster coding:** High-quality, context-aware suggestions that speed up development.
- **Fewer bugs:** AI-assisted debugging that helps catch issues earlier.
- **Cleaner code:** Automated refactoring that keeps projects maintainable.
- **Clarity:** Quick explanations for unfamiliar or complex logic.
- **More focus:** By automating repetitive tasks, we can concentrate on what matters.

For example, instead of manually rewriting a long function, we can ask Cursor to suggest a cleaner version, break it into smaller parts, or even generate docstrings to make it easier to understand.

To see the difference more clearly, here's how common tasks compare between a traditional editor and Cursor:

VS Code vs. Cursor

Task	In VS Code	In Cursor
Refactor a long function	Manually rewrite and test changes	Ask AI to restructure and improve logic
Debug an unknown error	Search Stack Overflow or try print/debug	Ask Cursor to explain and troubleshoot in place
Write boilerplate code	Use snippets or copy from examples	Describe in plain English and generate with AI
Add documentation	Manually write docstrings and comments	Use AI to generate context-aware summaries
Navigate codebase	Use file explorer or fuzzy search manually	Ask AI to locate functions or features across files
Manage version control	Use Git panel for commits and history	Use Git with AI assistance, including commit summaries and diff explanations
Write or update tests	Write tests manually or use snippets	Ask AI to generate or refactor test cases with full context awareness
Developer preference	VS Code widely adopted	In a 2025 report from Blott Studio, 83% of developers in trials preferred Cursor over other AI tools

ⓘ

A study by GitHub showed that developers using Copilot completed tasks 55.8% faster than those without AI assistance. While no equivalent study exists for Cursor, developers often report improved productivity and satisfaction when using it to plan, write, and debug code. Cursor's tight integration of chat, inline editing, and project-wide context is especially helpful in large or multi-file projects.

Instead of switching between tools, tabs, and mental modes, Cursor lets us handle these tasks directly in the editor with natural language and smart suggestions. It's a practical upgrade that reduces repetitive effort, speeds up development, and helps us stay focused on the work that matters.

How Cursor compares to other AI coding tools#

Before we explore Cursor's core features, it's helpful to see how it compares to other popular AI tools developers might already know, particularly GitHub Copilot and Windsurf. While all three offer AI assistance for coding, they differ in how they are integrated, the depth of features, and the flexibility they offer.

Feature / Focus	Cursor	GitHub Copilot	Windsurf
Editor integration	Cursor is a standalone desktop application built on a customized fork of open-source VS Code. It operates as a full-featured code editor with native AI capabilities.	GitHub Copilot is an extension that integrates into existing editors such as VS Code, JetBrains IDEs, and Neovim. It does not function as a standalone editor.	Windsurf offers both a VS Code extension and its own standalone editor, which is also based on a fork of VS Code. The editor includes deeper AI integration than the plugin.
AI capabilities	Cursor supports a wide range of AI-assisted features including in-editor chat, natural-language code edits, inline refactoring, function explanations, test generation, and a powerful agent mode that can autonomously perform complex coding tasks under developer supervision.	Copilot provides code autocomplete and an AI chat interface (Copilot Chat) that supports code explanations, debugging help, test writing, and terminal queries. Recent updates have introduced limited agent-like workflows in VS Code.	Windsurf combines code suggestions with an AI chat assistant capable of explaining, generating, and refactoring code. It also includes "Cascade," an agent-based system that can perform large, multi-step code modifications across files.

Cursor is a fully integrated AI code editor that helps us write, understand, and improve code across entire projects. It goes beyond simple code suggestions by offering tools for refactoring, explanation, and automation, all within a context-aware environment. Compared to tools like Copilot or Windsurf, Cursor provides more flexibility in model selection, deeper awareness of our codebase, and greater control over our AI work.

A quick overview of Cursor's core features#

Now that we understand Cursor and why it matters, let's explore its core features. This section provides a high-level preview of the tools and capabilities we'll explore in detail later. Each feature will be demonstrated through clear examples and annotated screenshots to help us understand how they fit into real development workflows.

AI-powered code completion#

Cursor offers intelligent, context-aware code suggestions. Unlike traditional editors that provide basic syntax hints, Cursor predicts entire lines or code blocks based on what we're working on. Writing a function, defining a route, or configuring a file helps us write faster and with fewer mistakes.

Natural language coding#

We can describe what we want in plain English, and Cursor turns it into working code. This is especially useful for generating boilerplate, utility functions, or prototyping new components quickly. It's like having a coding partner who understands our intent and programming style.

Context-aware chat#

Cursor includes an integrated chat assistant that understands the files we are working on. We can ask questions about the current code, troubleshoot issues, or request help with a specific task. Because it operates within the context of our project, its responses are more relevant and accurate than a generic chatbot.

Code explanation and inline help#

The Cursor helps us understand unfamiliar or complex code. We can highlight any portion of code and ask for an explanation in simple language. This is particularly helpful when reviewing someone else's code or returning to a project after a long break.

Debugging and refactoring suggestions#

Cursor goes beyond writing code. It actively helps us improve what we've already written. It can point out logic issues, recommend cleaner patterns, and suggest ways to simplify or refactor our code. These suggestions help keep our codebase readable and maintainable over time.

Automating tasks with the Cursor agent#

The **Cursor agent** allows us to automate common tasks like generating tests, adding type hints, or creating new file structures. Instead of performing these steps manually, we can ask the agent to handle them. This reduces repetitive work and helps us stay focused on problem-solving.

These features work together to create a smarter development environment. As we move through the course, we'll closely examine each one.

Who is using Cursor?#

Cursor is being adopted by engineering teams at leading technology companies, including OpenAI, Stripe, Spotify, Shopify, Instacart, and Ramp. These organizations use Cursor to speed

development, reduce manual effort, and improve project code quality. For example, Cursor grew from 150 to over 500 active users at Rippling within a few weeks. Their CTO shared that Cursor plays a central role in their day-to-day development workflow and has been especially helpful during large-scale code migrations. Cursor's growing adoption reflects how AI-assisted development is becoming a practical part of modern software engineering. It supports individual developers and teams by making common tasks like debugging, refactoring, and documentation faster and more manageable.

What's next?#

We've now seen what Cursor is and how it stands apart from traditional editors. Instead of being just a place to write code, Cursor actively supports us throughout the development process. It suggests completions, explains logic, fixes issues, and automates repetitive tasks. In the next lesson, we'll set up Cursor and examine the editor more thoroughly. We'll walk through the interface, learn how to navigate around a project, and get comfortable with the layout before we begin using its AI features.

Note: Cursor has built-in access to powerful AI models like GPT-4, GPT-4o, and Claude Sonnet. We can use its AI features immediately without setting up our API keys. The free plan gives us plenty of AI help for everyday coding tasks like writing, editing, and debugging. Some advanced features might run a little slower during busy times. We can upgrade to a paid plan if we need faster results or want to use AI more often. Cursor also gives us the option to connect our own AI accounts. For example, if we already have access to OpenAI or Anthropic or are running a custom model ourselves, we can link it to Cursor. This makes it easy to choose the best setup for our needs.

3. Installing and Navigating Cursor AI Code Editor

Clip source: [Installing and Navigating Cursor AI Code Editor](#)

[Home](#)

>

[Courses](#)

Installation and Basic Navigation

Explore the installation process for Cursor on macOS, Windows, and Linux, and learn to navigate its interface including the start screen, side panels, terminal, and AI assistant. Understand how to configure key settings such as AI models, privacy, and team collaboration to optimize your coding workflow with Cursor's AI tools.

In this lesson, we'll walk you through the setup process and explore Cursor's layout and configuration panels. By the end, we'll be ready to start using its AI features in real projects.

We'll learn how to:

- Install Cursor on macOS, Windows, or Linux.
- Launch the editor and access the start screen options.
- Navigate the interface using the top bar, side panels, and terminal.
- Explore key settings such as appearance, privacy, and chat behavior.
- Choose and configure AI models for personalized suggestions.
- Use version control and team collaboration tools inside Cursor.

Installing Cursor on your machine#

Installing Cursor is quick and straightforward.

Step 1: Download Cursor#

Visit the official Cursor website: <https://www.cursor.so>.

You'll find installation options for macOS, Windows, and Linux. Choose the version that matches your operating system and download the installer.

Minimum requirements:

- macOS 10.15 or later (Universal, x64, or Arm64)
- Windows 10 or 11 (x64 or Arm64)
- Linux distributions that support AppImage (e.g. Ubuntu 20.04+, Fedora 36+, Debian 11+, Arch)
- No additional Node.js or Python dependencies required

Step 2: Run the installer#

Once the download is complete:

- **macOS:** Open the `.dmg` file and drag the Cursor app into your Applications folder.
- **Windows:** Run the `.exe` installer and follow the prompts.
- **Linux:** Follow the CLI-based installation instructions provided on the website or GitHub page (usually via `.AppImage` or `.deb`).

Step 3: Launch Cursor#

After installation:

- Open the Cursor application from your system's application menu.
- The first time you open it, Cursor may ask for basic permissions and show a short onboarding screen.

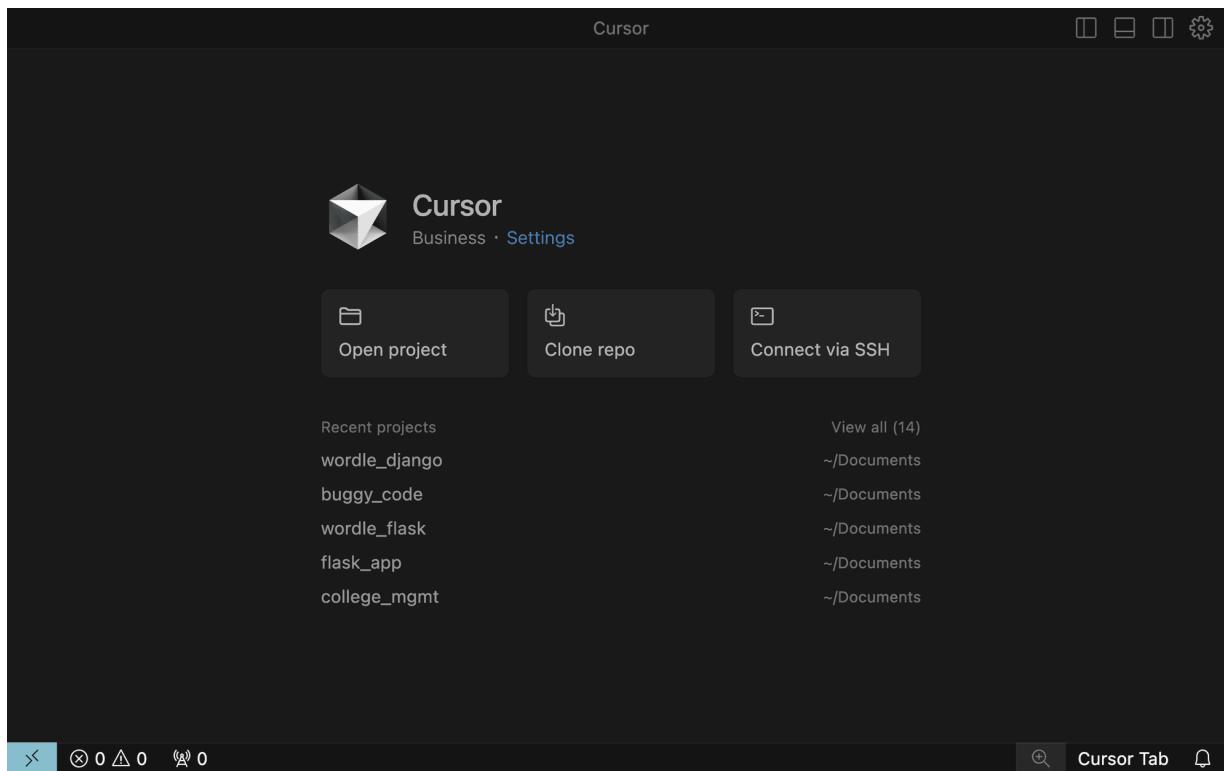


Note: Cursor is a modified version of VS Code, so it may prompt you to import some of your existing settings or extensions. You can skip this if you're starting fresh.

In the next section, we'll explore the layout of the Cursor and get comfortable navigating around it.

Exploring the Cursor interface#

Before diving into specific AI features, let's first get oriented with the main sections of the Cursor interface. The layout is intentionally clean and minimal, but it hides powerful functionality under a few key panels and controls. Here's a quick overview of what we'll see when we first open Cursor.



1 / 12

This is the first screen we see when we launch Cursor. From here, we can open a project, clone a repository, or connect to a remote environment. It also gives quick access to settings, terminal, AI tools, and notifications. We'll explore each of these options one by one in the next slides.



Cursor needs an internet connection to access its default cloud-based AI models. However, for fully offline usage, Cursor also supports connecting to self-hosted or proxy-hosted models, such as those run via Ollama or other local endpoints. If no model is connected, Cursor still works as a basic code editor, but AI features like chat, inline suggestions, and refactoring will be disabled. To get the full experience, either connect to the internet or set up a compatible local model.

The start screen#

When we launch Cursor, we land on the start screen. From here, we can:

- Open a project from our local machine.
- Clone a repository from a remote source.
- Connect to a project over SSH.
- Reopen recent projects we've worked on.

Cursor start screen

This screen gives us quick ways to jump back into work or start something new.

What is at the top-right corner?#

From left to right, the top-right corner contains:

- **Toggle primary side bar:** This lets us show or hide the main file explorer and navigation menu on the left.

- **Toggle panel:** This button is used to open or collapse the panel area at the bottom, which includes the terminal, debugger, and other tools.
- **Toggle AI pane:** This button opens or hides the Cursor AI assistant panel, where we interact with the AI directly.
- **Settings:** This opens the full Cursor settings interface, where we can configure everything from AI model preferences to keybindings, chat behavior, privacy options, and more.

Options available at the top-right of the Cursor start screen

What is at the bottom-right corner?#

On the far right, we have:

- **Cursor Tab:** This tab shows the AI status and lets us toggle between auto, manual, and snoozed modes or disable the assistant globally.
- **Notification center:** This icon displays system notifications such as AI task updates, model issues, or connectivity alerts.

Options available at the bottom-right of the Cursor start screen

What is at the bottom-left corner?#

This corner provides tools for remote development, debugging, and port management.

- **Remote window menu (X icon):** Clicking this opens a set of options to connect to SSH hosts, Codespaces, Dev Containers, or remote repositories.

- **Problems and warnings icon:** This shows the number of detected issues in our project and opens the “Problems” tab when clicked.
- **Debugger status icon:** This indicates whether a debugging session is active and links to the debug console.
- **Port forwarding manager:** This icon allows us to forward local ports and access running services through the browser or external tools.

Options available at the bottom-left of the Cursor start screen

Exploring Cursor settings: Where AI configuration comes to life#

The settings panel in Cursor is not just a place to tweak preferences, it's the command center that controls how Cursor integrates AI into our workflow. From choosing which models power our suggestions to deciding how the AI interacts with our code, every option here shapes the behavior and intelligence of the editor.

Let's go through the main categories in the settings view and understand what each allows us to control.

General settings#

The "General" tab is where we set up the basics of our Cursor environment. This includes account details, importing settings from Visual Studio Code, appearance preferences, and privacy controls. These settings help us get comfortable and align with our preferred work style.

Cursor general settings

Account#

At the top of the settings panel, we can see the email address currently logged in. This helps us confirm we are working under the correct account. From here, we can:

- Invite teammates to collaborate using the same workspace.
- Log out if we need to switch accounts or devices.



Working with Teams and Git: In addition to inviting teammates to collaborate in the same workspace, Cursor supports Git-based workflows for multi-

developer projects. You can create branches, commit changes, and sync code with remote repositories such as GitHub, all from within Cursor's built-in source control panel.

This allows developers to collaborate using familiar version control practices while still benefiting from Cursor's AI features. Later in the course, we'll walk through how to push code to GitHub directly from the editor.

Import settings from VS Code#

If we have used Visual Studio Code before, Cursor gives us an easy way to bring over our existing setup. The import option allows us to carry forward:

- Editor layout and visual preferences
- Custom keyboard shortcuts
- Extensions we previously used

This helps us get started without needing to rebuild our development environment from scratch.

Appearance#

Under "Appearance," we can access links to customize how Cursor looks and behaves. These options include:

- Editor font, spacing, and theme
- Settings like auto-save and word wrap
- Shortcut configuration for smoother navigation

These settings match what we expect from a modern code editor and help keep Cursor aligned with our style.

Privacy mode#

The "Privacy mode" is an important feature, especially when working with sensitive or private codebases. When enabled:

- Cursor does not store or use our code in any external systems.
- No context from our files is sent or logged outside our machine.

If we are part of a team, "Privacy mode" can also be enforced across all members to ensure consistent data protection.

Features settings#

This is one of the most important areas in Cursor, as it directly affects how the AI behaves inside the editor.

The "Features" tab controls how AI suggestions, chats, inline edits, and automation work inside Cursor. These settings allow us to decide how involved the AI assistant should be in our workflow and how much control we want over its actions.

Cursor Tab feature settings

Cursor Tab#

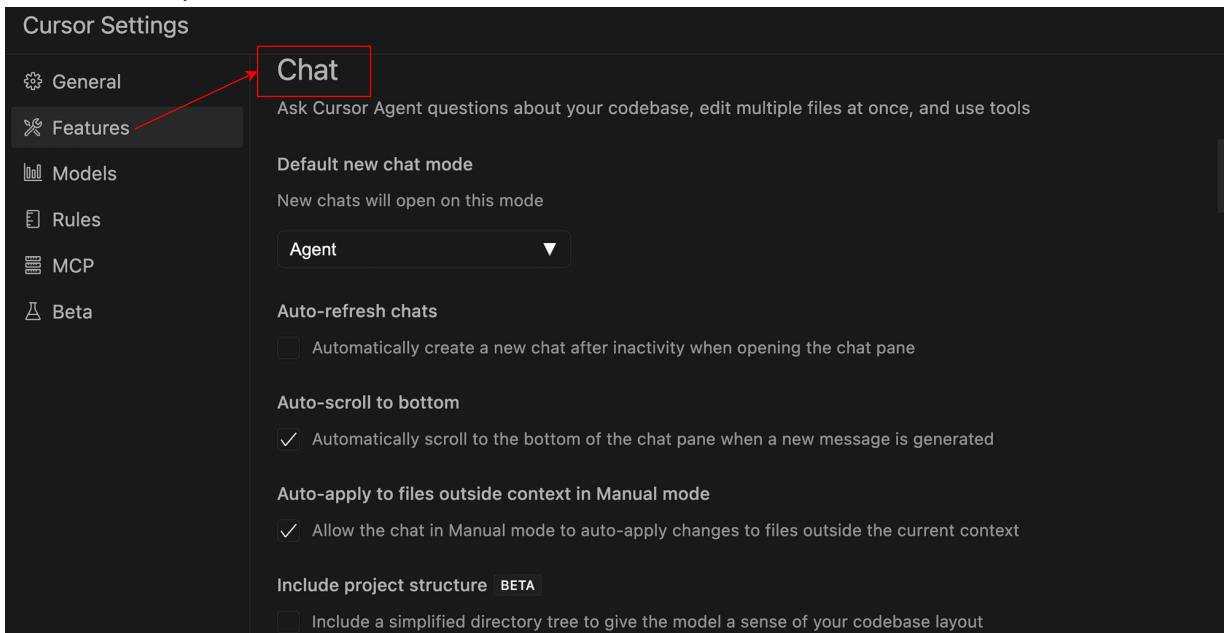
This feature defines how Cursor handles inline code suggestions, similar to tools like GitHub Copilot, but more customizable.

- **Enable multi-line suggestions:** This option allows Cursor to suggest several lines of code at once, rather than limiting output to a single line.
- **Enable suggestions in comments:** When this is turned on, Cursor can read the intent written in code comments and provide appropriate suggestions.

- **Allow partial accepts:** This setting allows us to accept part of a suggestion without inserting the full completion.
- **Enable auto-import (TypeScript only):** If we are working in TypeScript, Cursor can automatically insert import statements when required by a suggestion.

Chat settings#

The "Chat" feature settings let us configure how we interact with Cursor's AI through conversations, suggestions, and file updates. These options help tailor the experience to our workflow and preferences.



The screenshot shows the 'Cursor Settings' interface with a dark theme. On the left, a sidebar lists several sections: General, Features (which is selected and highlighted in grey), Models, Rules, MCP, and Beta. The main area is titled 'Chat' and contains the following configuration options:

- Default new chat mode:** Set to 'Agent' (selected from a dropdown menu).
- Auto-refresh chats:** An unchecked checkbox with the label 'Automatically create a new chat after inactivity when opening the chat pane'.
- Auto-scroll to bottom:** A checked checkbox with the label 'Automatically scroll to the bottom of the chat pane when a new message is generated'.
- Auto-apply to files outside context in Manual mode:** A checked checkbox with the label 'Allow the chat in Manual mode to auto-apply changes to files outside the current context'.
- Include project structure BETA:** An unchecked checkbox with the label 'Include a simplified directory tree to give the model a sense of your codebase layout'.

1 / 5

- **Default new chat mode:** This setting determines which mode is automatically selected when we start a new chat session. We can choose from the following options:
 - **Agent:** The AI will act as a proactive coding assistant, capable of performing actions across files.
 - **Ask:** The AI will behave like a conversational assistant, responding to queries without applying changes to the codebase.
 - **Manual:** The AI will suggest edits, but we must manually review and apply them.

- **Last used mode:** The Cursor will remember and use the same mode we last interacted with for any new chat.
- **Auto-refresh chats:** When enabled, Cursor will automatically start a new chat session if the chat pane is reopened after a period of inactivity. This helps keep conversations fresh and relevant.
- **Auto-scroll to bottom:** This option ensures that the chat pane always scrolls to the most recent message, so we don't miss new responses during an ongoing exchange with the AI.
- **Auto-apply to files outside context in Manual mode:** If we enable this setting, the Cursor will apply changes even to files that weren't explicitly opened or referenced in the chat context, as long as we're in Manual mode.
- **Include project structure (Beta):** Enabling this allows Cursor to use a simplified directory tree to better understand our project's layout and structure, which can improve the accuracy of AI-generated code suggestions.
- **Enable auto-run mode:** With this setting enabled, Cursor's agent can execute commands and apply changes without asking for confirmation. It's useful for speeding up workflows, though it should be used with caution.
- **Command allowlist:** We can add specific commands here that the agent is allowed to run automatically when auto-run mode is active. This gives us control over what it can execute without prompts.
- **Command denylist:** In contrast, this list allows us to explicitly block certain commands from ever being executed automatically by the agent.
- **Delete file protection:** If we enable this, the agent will not be allowed to delete files on its own. This adds a safety layer against accidental data loss.
- **MCP tools protection:** This setting prevents the agent from using tools defined in the MCP (Model Context Protocol) section unless we explicitly allow it.
- **Dot files protection:** When this is enabled, Cursor won't modify dotfiles like `.gitignore`. This is helpful for preserving configuration and system-level integrity.

- **Outside workspace protection:** This ensures that Cursor only modifies files inside the current workspace, preventing unintended edits to unrelated directories.
- **Dialog “Don’t ask again” preferences:** We can manage previously suppressed dialogs here. This is where we restore any confirmations we told Cursor not to show again.
- **Large context:** When turned on, Cursor will use extended context windows during chats. This can provide more detailed suggestions, but it may also consume more tokens or processing time.
- **Collapse input box pills in pane or editor:** This option reduces visual clutter by collapsing command pills in the chat or editor input fields.
- **Iterate on lints:** If we enable this, Cursor will automatically fix linter errors during agent-assisted coding sessions, saving us the effort of resolving them manually.
- **Hierarchical Cursor Ignore:** This lets `.cursorignore` rules apply to all subdirectories, mirroring how `.gitignore` works across nested folders. A restart is required after changing this setting.
- **Auto-accept diffs:** This automatically accepts all diffs in the composer once the changes are no longer part of the working tree. It’s helpful for keeping things clean after edits are finalized.
- **Custom modes (Beta):** We can create our own chat modes here, allowing tailored AI behavior for specific workflows. This setting is still in beta.
- **Play sound on finish (Beta):** If we enable this, Cursor will play a notification sound when the AI finishes generating a response, making it easier to know when it’s ready.
- **Auto Group Changes (Beta):** This feature allows Cursor to group related edits made during a chat session, helping us review them more clearly before committing.
- **Web Search Tool (Beta):** When enabled, this allows the AI to search the web for relevant information while in chat or agent mode. It expands Cursor’s capabilities beyond the local codebase.

Codebase Indexing#

The “Codebase Indexing” feature governs how Cursor processes and understands our entire codebase using embeddings and metadata. While the code remains local, indexing metadata is stored in the cloud to power code-aware AI responses.

Cursor Codebase Indexing feature settings

- **Index new folders by default:** When this setting is enabled, Cursor will automatically index any new folders that we open. If we disable it, we will need to manually index folders using the “Compute Index” button. It is also important to note that folders with more than 250,000 files will never be auto-indexed.
- **Ignore files:** This allows us to specify which files Cursor should exclude from indexing. Cursor respects our `.gitignore` file, and we can additionally configure exclusions manually using the “Configure ignored files” link.
- **Git graph file relationships:** When this is enabled, Cursor will analyze the git history to understand relationships between files. Metadata like commit messages, change frequency, and author information (with SHA obfuscation) are uploaded to improve contextual awareness. We can inspect what is stored using the “Show debug view” option.
- **Team level repository control:** If our team has set up repository-wide indexing rules, those rules will be applied automatically. These patterns define which files are included or excluded in the indexing process and can be explored using the “Show team patterns” link.

Docs#

The “Docs” feature allows us to manage custom documentation that we’ve added to Cursor. This documentation can be referenced in chats and edits to enrich context-aware suggestions.

Cursor Docs feature

- **Manage custom docs:** We can add personalized or project-specific documentation to guide the AI's responses. To create one, we type `@Add` in a chat or during an edit session to start building a new doc.

Editor#

The "Editor" feature controls how Cursor behaves in the code editing interface, particularly with tooltips, keyboard interactions, and visual diff presentation.

Cursor editor settings

- **Show chat/edit tooltip:** When this is enabled, Cursor displays a floating tooltip near any highlighted code. This helps us quickly trigger actions or suggestions from the AI.
- **Auto parse inline edit links:** If we enable this option, any edit link pasted into the Command-K input field is parsed automatically. This streamlines the process of applying suggested edits.

- **Auto select for Ctrl/⌘ + K:** When enabled, Cursor will automatically select relevant code regions when we use the Ctrl or ⌘ + K shortcut to invoke inline edits. This removes the need to highlight code manually.
- **Use themed diff backgrounds:** This option adds color-based themes to diff views, making it easier for us to visually distinguish changes made during editing.
- **Use character-level diffs:** If we turn on this option, Cursor will show changes at the character level in the diff view. This gives us precise insight into even the smallest code modifications.

Terminal#

The “Terminal” feature provides us with useful enhancements and AI-assisted feedback while working in the integrated terminal inside Cursor.

Cursor terminal settings

- **Terminal hint:** This setting displays helpful text hints at the bottom of the terminal window. It ensures we have constant guidance on what actions are available or recommended.
- **Show terminal hover hint:** When enabled, Cursor shows interactive hints, such as “Add to chat,” when we hover over parts of the terminal. This allows us to quickly pass terminal output into a chat prompt.

- **Use preview box for terminal ⌘K:** If this is turned on, Cursor will show a preview box when using ⌘K in the terminal. If it is disabled, responses are streamed directly into the shell without preview, providing a more seamless command execution flow.

Models settings#

The “Models” section in Cursor settings lets us choose which large language models (LLMs) to use and configure how Cursor accesses them. Some models come pre-integrated with Cursor, while others require manual setup.

AI model settings in Cursor

- **Model selection list:** We can enable support for specific models such as Claude (3.5-sonnet, 3.7-sonnet, 3.7-sonnet-max), Gemini (2.5-pro-exp, 2.5-pro-max), GPT (GPT-4, GPT-4o), and several others. Each checkbox determines whether that model is available for use across Cursor's chat and automation features. Cursor also supports reasoning-optimized models like OpenAI o3, Claude 3.7, and Grok 3. These are designed for deeper analysis, multi-step reasoning, and long-context understanding, useful when working with large or complex codebases.
- **The “Add model” button:** If we want to use a model not listed, we can add it manually using the “Add model” option. This is especially useful for connecting to newly released models, private or enterprise deployments.

The table below lists some of the most popular models supported in Cursor, along with their core strengths and typical use cases. This can help us choose the right model based on our development needs or preferences.

Popular AI Models in Cursor and When to Use Them

Model	Provider	Strengths	Best For
GPT-4 / GPT-4o	OpenAI	Strong reasoning, balanced performance	General development, code writing, multi-turn conversations
Claude 3.7 Sonnet	Anthropic	Long context, thoughtful explanations	Document-heavy projects, code summarization, debugging
Gemini 2.5 Pro	Google	Tool-aware, precise, good with structure	Refactoring, navigation, planning code changes
Grok 3	xAI	Reasoning-focused, exploratory	Architecture planning, complex prompts, creative coding
DeepSeek	DeepSeek	Open and research-driven, multilingual	Prototyping, testing ideas, cost-sensitive experimentation

API keys#

As we've seen in the model list above, Cursor provides built-in access to many popular state-of-the-art models, allowing us to use them without adding our own API keys for standard usage. However, if we want to connect to a custom model or use our own provider account for cost control, access to private deployments, or support for newer models, we can enter the necessary API credentials in the Model Settings.

Note: Connecting to GPT-4, Claude or other LLMs via API may incur usage fees based on the provider's pricing.

API key settings in Cursor

- **OpenAI API Key:** This allows us to use GPT models directly through OpenAI's public API. We also have the option to override the base URL, which is useful for routing requests through a proxy or internal gateway.
- **Anthropic API Key:** By adding this key, we can use any of the Claude models. Once enabled, the key automatically applies to all models starting with "Claude—."

- **Google API Key:** With a Google AI Studio API key, we can activate access to models like Gemini and integrate them into our workflows.
- **Azure API Key:** If we are using OpenAI models via Microsoft Azure, we can enter the necessary base URL, deployment name, and key. The Cursor will then send requests through the Azure-managed endpoints.
- **AWS Bedrock:** This section allows us to connect Claude models through AWS. We can either input access and secret keys directly or configure IAM roles for a more secure, enterprise-friendly setup. We also define the AWS region and test model identifier.

By selecting the right models and linking them with the appropriate API keys, we gain full control over the AI capabilities Cursor can use, balancing performance, accuracy, and cost depending on our preferences.

Rules#

The “Rules” section in Cursor settings helps us shape how Cursor’s AI behaves by providing additional context and preferences. These rules guide the models in adapting to our personal style and project-specific standards, ultimately making interactions more efficient and aligned with our needs.

Cursor rules settings

- **User Rules:** We can define global preferences that apply across all AI interactions in chats, code composers, and Command-K executions. For example, we might instruct the model to always reply in a specific language, use a certain coding style, or avoid verbose explanations. These rules act like lightweight behavioral instructions that persist across sessions.

- **Project Rules:** We can specify conventions unique to our current project so the model better understands and respects its structure. These rules are tied to the project itself and are synced with the codebase. Cursor allows us to add new project rules manually or fetch them automatically when an agent is active.
- **Include `.cursorrules` file:** This toggle lets us control whether the `.cursorrules` file is included in our AI requests. When it is enabled, the file provides additional structure to the model's context. If we turn it off, Cursor will exclude the file from its prompt context, which can be useful in tightly scoped interactions.

MCP#

MCP stands for Model Context Protocol, which allows us to extend the capabilities of Cursor by integrating external tools into the agent system. This is especially useful when we want to offer our own tools or services to the Cursor agent beyond what is built-in.

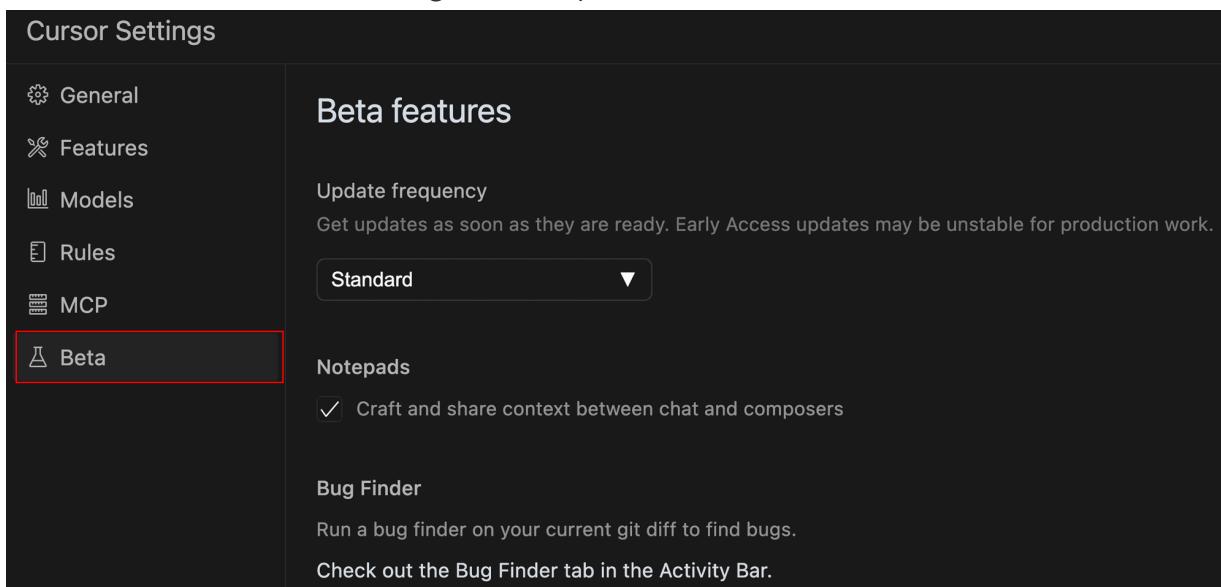
MCP settings in Cursor

- **Add new global MCP server:** We can register a global MCP server using the provided button. This connects Cursor to an endpoint that exposes new functionalities, such as internal APIs or custom automations, making them available to the agent.

- **Project-specific configuration:** Instead of using a global server, we can configure MCP per project by defining the endpoint in the `.cursor/mcp.json` file within the project root. This lets us tailor tool access and behavior on a per-project basis.
- **Documentation reference:** Cursor provides a direct link to the MCP documentation so we can understand the full specification and implementation details if we are building or hosting our own tools.

Beta#

The “Beta” tab in the Cursor settings allows us to experiment with upcoming features that may still be in development or testing. While these options can unlock powerful capabilities, we should be cautious when enabling them for production workflows.



The screenshot shows the 'Cursor Settings' interface with the 'Beta' tab selected. The left sidebar lists several categories: General, Features, Models, Rules, MCP, and Beta. The 'Beta' category is highlighted with a red border. The main content area is titled 'Beta features'. It contains two sections: 'Update frequency' and 'Notepads'. Under 'Update frequency', it says 'Get updates as soon as they are ready. Early Access updates may be unstable for production work.' with a dropdown menu set to 'Standard'. Under 'Notepads', there is a checked checkbox for 'Craft and share context between chat and composers'. Below that is the 'Bug Finder' section, which includes a description and a note to 'Check out the Bug Finder tab in the Activity Bar.'

1 / 2

- **Update frequency:** We can choose how often Cursor delivers updates to our environment. Setting it to “Early Access” lets us receive the newest features as soon as they are available, while “Standard” keeps our environment more stable with fewer update disruptions.
- **Notepads:** We can enable this to allow Cursor to automatically share context between chats and composers. This helps the AI maintain continuity across different parts of our workspace.

- **Bug Finder:** We can activate a bug detection tool that inspects our current Git diff for potential issues. If enabled, a dedicated Bug Finder tab will appear in the Activity Bar, making it easier for us to catch and resolve errors early in development.

Conclusion#

In this lesson, we set up the Cursor on our local machine and became familiar with its interface. We started by walking through the installation steps for macOS, Windows, and Linux, making sure we could open the editor and access its main controls. Once inside, we explored how to open projects, connect to remote environments, and use the key navigation points like the side bar, terminal panel, AI pane, and settings. We also examined the layout of the start screen and learned what each icon and button helps us do. This gave us a clear picture of how to move between files, run tasks, and access AI tools efficiently.

We then took a closer look at the settings panel. This is where we configure how Cursor behaves and how it integrates with different AI models. We reviewed the settings for general preferences, AI features, model keys, rule definitions, external tool integration, and experimental options. Understanding these options helps us make Cursor work the way we want, both in how it looks and how it responds to our prompts.

Now, we are ready to begin building with Cursor!

4. Write and Run Flask Apps with Cursor AI Code Editor

Clip source: [Write and Run Flask Apps with Cursor AI Code Editor](#)

[Home](#)

>

[Courses](#)

>

[Code Smarter with Cursor AI Editor](#)

AI-Powered Code Writing

Explore how to use Cursor's AI-powered code editor to create, run, and improve a Flask "Hello, World!" app from natural language prompts. Understand setup steps, environment management, debugging issues, and incremental feature additions. This lesson shows how Cursor assists coding workflows with chat modes, model selection, and context attachment to streamline Python web app development.

In this lesson, we'll build a simple "Hello, World!" Flask application using natural language prompts and explore how Cursor supports real development workflows. The focus is on how the AI assists us at each step, from generating files to debugging and running the code.

We'll learn how to:

- Prompt Cursor to generate functional code using plain English.
- Scaffold and structure a working Python Flask app.
- Handle setup steps like creating virtual environments and installing dependencies.
- Run code, respond to errors, and adapt when things do not go as expected.
- Add new features incrementally by continuing the conversation.
- Use chat settings like Agent mode, model selection, and context attachment for better results.

The screenshot shows the Cursor AI-powered code editor interface. On the left, there is a file tree for a project named 'FLASK_APP' containing files like 'venv', 'app.py', 'pyenv.cfg', 'README.md', and 'requirements.txt'. The main workspace shows a code editor with the following content in 'app.py':

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == '__main__':
    app.run(debug=True)
```

To the right of the code editor is a large text area titled 'Basic Flask Hello World App' with the instruction: 'Please write code to create a basic hello world python flask app.' Below this, a message says: 'I'll help you create a basic Flask "Hello World" application. I'll create the necessary files including a requirements.txt for dependencies and a README.md for documentation.' A sub-section titled 'First, let me create the main application file:' shows the generated code. Further down, it says: 'Now, I'll create a requirements.txt file to specify the Flask dependency:' with the content: 'Flask==3.0.2'. At the bottom, it says: 'Finally, let me create a README.md with instructions on how to run the application:'.

We may run into missing packages, permission issues, or environment-related errors. We'll work through them step by step, just like we would in any real-world development process. Along the way, we'll see how Cursor helps streamline tasks, reduce manual effort, and support our decision-making without taking control away.

Setting up project folder (Important first step)

Before entering your first prompt in Cursor, it's important to start with a clean setup. A common mistake is starting a chat in a blank Cursor window, which can lead to problems like lost chat history or permission errors when the AI tries to write files.

To avoid this, follow this simple best practice:

- **For a new project:** Create a new folder on your system (e.g., `MyFirstCursorApp`). Then open this folder in Cursor using "Open Folder", and only then start our AI chat.
- **For an existing project:** Always open the relevant project folder first before you begin chatting or generating files.

By opening a folder first, we ensure that:

- Our chat history is tied to the specific project and saved with it.
- Cursor has access to create or modify files as needed.
- The AI understands your project structure from the start.

This small step ensures a smoother and reliable development experience.

Generating the Flask app with a prompt#

To get started, we open a new chat with Cursor and enter a natural language prompt:

Prompt: Please write code to create a basic hello world python flask app.

Cursor responds by scaffolding a basic Flask application for us. It automatically generates:

- `app.py` : The main application file that sets up a simple route returning "Hello, World!".
- `requirements.txt` : Specifies the Flask dependency (`Flask==3.0.2`).
- `README.md` : Provides setup instructions, including virtual environment creation and server startup steps.

This saves us time and gets us straight into a working structure.



Note: The generated output may differ slightly depending on the model selected and the specificity of your prompt. Feel free to refine your instructions or iterate with follow-up prompts to align the result with your expectations.

Cursor also walks us through how to:

1. Create and activate a virtual environment.
2. Install dependencies.
3. Run the app with `python app.py`.
4. Visit `http://localhost:5000` to test the output.

The application is configured to run in debug mode, which is helpful during development.

Below, we walk through the full response generated by Cursor for this prompt, step by step:

I'll help you create a basic Flask "Hello World" application. I'll create the necessary files including a `requirements.txt` for dependencies and a `README.md` for documentation. First, let me create the main application file:



Python 3.10.4

File C []

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == '__main__':
    app.run(debug=True)
```

File

...

app.py

Now, I'll create a `requirements.txt` file to specify the Flask dependency:



Python 3.10.4

File C []

```
Flask==3.0.2
```

File

...

requirements.txt

Finally, let me create a `README.md` with instructions on how to run the application:



Python 3.10.4



```
# Flask Hello World App
A simple Flask application that displays "Hello, World!" on the homepage.

## Setup
1. Create a virtual environment (recommended):
```bash
python -m venv venv
source venv/bin/activate # On Windows, use: venv\Scripts\activate
```

2. Install dependencies:
```bash
pip install -r requirements.txt
```

## Running the Application
Run the application using:
```bash
python app.py
```

The application will be available at `http://localhost:5000`
```



...

README.md

I've created a basic Flask application with the following structure:

app.py : The main application file with a simple route that returns "Hello, World!"
requirements.txt : Lists Flask as a dependency
README.md : Contains instructions for setting up and running the application

To run the application:

1. First, create a virtual environment and activate it



2. Install the requirements using `pip install -r requirements.txt`
3. Run the application using `python app.py`
4. Visit `http://localhost:5000` in your web browser to see the "Hello, World!" message

The application is set up with debug mode enabled, which is helpful during development as it will automatically reload when you make changes to the code.

Would you like me to help you set up the virtual environment and run the application?

The AI agent creates the code files, provides setup instructions, and proactively offers to help run the application once the initial setup is complete.

ⓘ In Agent mode, Cursor can suggest follow-up actions or offer to help with the next step based on what it has just completed. This makes the interaction feel more guided and collaborative.

Next, we ask the Cursor to run the application.

Running the app#

With the files generated, we moved on to running the application. So we ask Cursor:

Prompt: Please run the application

Cursor suggested the typical next steps: create a virtual environment, activate it, install dependencies, and then run the app. But, as often happens in real development, things didn't go exactly as planned.

ⓘ Some commands may require elevated privileges or system configuration outside Cursor's control.

Let's go through the response step by step:

I'll help you run the Flask application. First, let's create and activate a virtual environment, then install the requirements, and finally run the application.

```
☒ python -m venv venv
Asmat@Asmat-Batool-TCE-Lead flask_app % python -m venv venv
zsh: command not found: python
Asmat@Asmat-Batool-TCE-Lead flask_app %
```

I notice that the `python` command is not found. On macOS, we should try using `python3` instead:

```
| Asmat@Asmat-Batool-TCE-Lead flask_app % python3 -m venv venv
| zsh: killed      python3 -m venv venv
| Asmat@Asmat-Batool-TCE-Lead flask_app %
```

I see we're having some issues with the virtual environment creation. Let's try to run the application directly with python3:

```
☒ python3 -m pip install -r requirements.txt  
  
Asmat@Asmat-Batool-TCE-Lead flask_app % python3 -m pip install -r requirements.txt  
zsh: killed      python3 -m pip install -r requirements.txt  
Asmat@Asmat-Batool-TCE-Lead flask_app %
```

Cursor started by running the following command:

Ace Editor

1

```
python -m venv venv
```



HTML Content

This failed with a message saying `python` was not found. Cursor immediately adapted and recommended using `python3` instead. We tried:

Ace Editor

1

```
python3 -m venv venv
```



HTML Content

But this too failed. Cursor then suggested skipping virtual environment setup for now and trying:

Ace Editor

1

```
python3 -m pip install -r requirements.txt
```

XX
XX
XX
XX
XX

XX
XX

HTML Content

Again, the command was killed before it could be completed. Recognizing that the issue likely wasn't with the virtual environment itself but with the Python installation, Cursor suggested a fallback plan.

I apologize for the issues we're encountering. It seems there might be some system-level problems with Python execution. Let me suggest a few things you can try:

1. First, verify your Python installation by running:

```
$ bash
which python3
python3 --version
```

Run

2. If Python is properly installed, you might want to try installing Flask globally first:

```
$ bash
python3 -m pip install Flask
```

Run

3. Then try running the application:

```
$ bash
python3 app.py
```

Run

Could you please try these commands and let me know what happens? This will help me better understand what might be causing the issues and provide more specific guidance.

Also, if you're using any specific Python version manager (like pyenv) or have a preferred way of running Python applications on your system, please let me know, and I can adjust the instructions accordingly.



Note that we can run all the suggested commands directly within the Cursor chat window using its built-in terminal support. We can choose to run commands manually or enable auto-run mode to let Cursor execute them automatically.

As we followed Cursor's suggestions, we kept encountering errors like `zsh: killed`, even when running basic commands such as checking the Python version. After a bit of investigation, we identified the actual issue: the version of Python installed on the system was outdated. Specifically, Python 3.6 was pre-installed on macOS Sonoma, but it's not compatible.

Once we updated Python to a more recent, supported version, the issues were resolved. We were able to confirm the Python version successfully,

create a virtual environment without any problems, and move forward with the setup.

i

While we performed the upgrade manually in a separate terminal, Cursor includes a fully functional built-in terminal, so we could have run the Python upgrade from within Cursor as well.

With the virtual environment successfully created, we asked Cursor for the next step:

Prompt: I have created the virtual environment, what is the next step?

Cursor responded with clear guidance:

I see that you've created the virtual environment. The next step is to activate it and install the required dependencies. Since you're on macOS, let's activate the virtual environment:

```
└─ source venv/bin/activate
```

Great! Now that the virtual environment is activated (you can see the `(venv)` prefix in your terminal), let's install the required Flask package:

```
pip install -r requirements.txt

(venv) Asmat@Asmat-Batool-TCE-Lead flask_app % pip install -r requirements.txt
Collecting Flask==3.0.2 (from -r requirements.txt (line 1))
  Downloading flask-3.0.2-py3-none-any.whl.metadata (3.6 kB)
Collecting Werkzeug>=3.0.0 (from Flask==3.0.2->-r requirements.txt (line 1))
  Using cached werkzeug-3.1.3-py3-none-any.whl.metadata (3.7 kB)
Collecting Jinja2>=3.1.2 (from Flask==3.0.2->-r requirements.txt (line 1))
  Using cached jinja2-3.1.6-py3-none-any.whl.metadata (2.9 kB)
Collecting itsdangerous>=2.1.2 (from Flask==3.0.2->-r requirements.txt (line 1))
  Using cached itsdangerous-2.2.0-py3-none-any.whl.metadata (1.9 kB)
Collecting click>=8.1.3 (from Flask==3.0.2->-r requirements.txt (line 1))
  Using cached click-8.1.8-py3-none-any.whl.metadata (2.3 kB)
Collecting blinker>=1.6.2 (from Flask==3.0.2->-r requirements.txt (line 1))
  Using cached blinker-1.9.0-py3-none-any.whl.metadata (1.6 kB)
Collecting MarkupSafe>=2.0 (from Jinja2>=3.1.2->Flask==3.0.2->-r requirements.txt (line 1))
  Downloading MarkupSafe-3.0.2-cp313-cp313-macosx_10_13_universal2.whl.metadata (4.0 kB)
  Downloading flask-3.0.2-py3-none-any.whl (101 kB)
```

On macOS, Cursor suggested the standard activation command:

Ace Editor

1

```
source venv/bin/activate
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

HTML Content

After running this, we could see the `(venv)` prefix appear in the terminal prompt, confirming that the environment was active.

Next, it asked us to install the dependencies listed in the `requirements.txt` file:

Ace Editor

1

```
pip install -r requirements.txt
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

HTML Content

This successfully installed Flask and related packages.

With Flask installed, we moved on to the final step:

Ace Editor

1

```
python app.py
```



The server started up on `http://127.0.0.1:5000`, and visiting that URL in the browser showed the expected “Hello, World!” message.



Hello World! Python Flask app is running in the browser

This confirmed that the Flask app was running correctly, just as we had asked for in our original prompt. Cursor guided us from idea to execution with helpful explanations, fallback suggestions when things didn't work, and an AI that felt like a real coding partner.

Make an incremental feature request#

To explore how Cursor handles incremental changes, we tried a simple follow-up:

Prompt: Now add a new route that returns the current server time.

Cursor updated `app.py` by adding a new route using Python's `datetime` module, as we can see in the screenshot below:

```

app.py
from flask import Flask
from datetime import datetime

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

@app.route('/time')
def current_time():
    return f'Current server time is: {datetime.now().strftime("%Y-%m-%d %H:%M:%S")}'


if __name__ == '__main__':
    app.run(debug=True)

```

The screenshot shows a code editor interface with multiple tabs: `app.py`, `requirements.txt`, and `README.md`. The `app.py` tab contains the provided Python code for a Flask application. A sidebar on the right displays AI-generated code completion and documentation for the `current_time()` route, including a preview of the output and instructions to accept or reject the changes.

Since the Flask app was already running in debug mode, the server automatically reloaded the updated file. We opened the new route (`http://127.0.0.1:5000/time`) in the browser, and saw the following output:

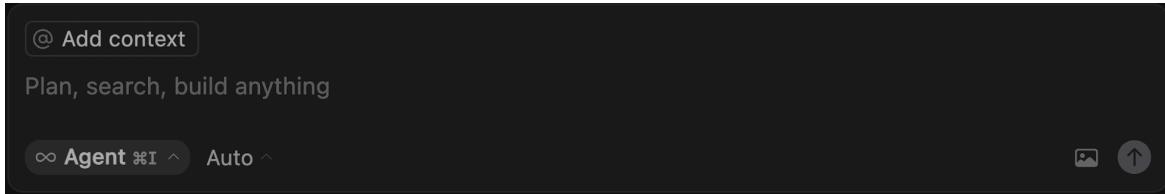


This follow-up demonstrates how Cursor supports incremental coding. We didn't need to start over or re-explain the context. Cursor understood our intent, edited the code in the right place, and let us move forward smoothly, much like pair programming with a helpful assistant.

Note: By default, Cursor sends code and context to external large language models (LLMs) such as GPT-4 or Claude to generate responses. This means parts of your code may leave your machine when using AI features. For sensitive or enterprise-level projects, you can manage this behavior in the settings panel. Cursor supports “Privacy mode,” which disables sending context externally, and it also allows connecting to self-hosted or OpenAI-compatible local models for fully offline or controlled usage.

Chat features in action#

As we interacted with Cursor's AI to build and run the Flask app, we also explored how flexible and customizable the chat interface is. Before sending a message, we can adjust a few settings that shape how the assistant will behave.



Chat input bar

Chat mode selection#

At the bottom left of the chat input bar, we can choose the chat mode that fits our intent:

- **Agent:** This mode is best for planning, building, and running code with AI assistance. The AI can take actions such as editing files, running commands, and using tools like search and diff across the project.
- **Ask:** This is ideal for getting explanations or asking questions about the codebase. The AI responds with guidance or suggestions but does not change the code.
- **Manual:** It gives full control to us; the AI won't trigger actions or add context unless we explicitly do so.

These modes let us shift between assistant-driven, conversational, and manual workflows as needed.

ⓘ

It's easy to confuse Agent mode with a typical AI chatbot. In Agent mode, Cursor is not just replying with suggestions. It can take real actions across your project, such as creating or editing files, running terminal commands, and using tools like search or diff. You can think of it as a coding co-pilot that actively participates in the workflow. It's still important to review changes before applying them, Agent mode is powerful, but not infallible.

By contrast, Ask mode functions more like a conversational assistant. It provides explanations, suggestions, and guidance, but does not make any changes to your codebase.

Model selection#

Next to the chat mode, there's an option to either let Cursor auto-select the best model for the task or pick one manually. Cursor defaults to auto-select, which chooses a model based on speed, quality, and available context length. But we can switch to any installed model like GPT-4o, Claude, or Gemini.

This flexibility is useful when we want more control over cost, latency, or model behavior.

Adding context to a prompt#

The "@Add context" option lets us attach relevant information to a prompt so the AI can generate more accurate responses. We can include files, folders, terminal output, past chats, or even recent Git changes from the current workspace.

One especially useful feature is the ability to reference **Git diffs**. For example, we can ask:

- "What changed in my last commit?"
- "Summarize the recent changes in this branch"
- "Explain this diff"

Cursor can automatically extract the latest changes and use them as input. This is incredibly helpful for reviewing our own work, documenting changes, or debugging code we just modified.

⊗

We have not yet learned to push our code to GitHub. We will do it later in the course.

With this context option, we stay in control of what's included. Before sending a message, we can choose exactly what context to attach, ensuring the assistant sees only what's relevant.

Attaching an image#

At the bottom-right of the chat input bar (see the image above for reference), we have the option to attach an image directly to a prompt. This is helpful when asking about screenshots of error logs, UI layouts, or visual bugs.

By using these features together, we can guide the assistant effectively and create a workflow that feels both collaborative and adaptable to different needs.

Conclusion#

In this lesson, we used Cursor's chat interface to go from a plain language prompt to a fully running Flask application. Along the way, we explored how the assistant can generate files, guide us through environment setup, troubleshoot common issues, and help us run the project from within the editor.

We also learned how Cursor adapts when things don't go as planned, offering practical alternatives and helping us stay focused on the task. This hands-on example showed that Cursor supports real development workflows by combining code generation, context awareness, and useful automation.

5. Debugging and Refactoring Code Efficiently with Cursor AI

Clip source: [Debugging and Refactoring Code Efficiently with Cursor AI](#)

[Home](#)

>

[Courses](#)

>

[Code Smarter with Cursor AI Editor](#)

Debugging and Refactoring with Cursor

Explore how to use Cursor AI to debug runtime errors and refactor existing code for improved readability and maintainability. This lesson guides you through fixing bugs with precise prompts, attaching context like terminal output or screenshots, and applying standard refactoring techniques. You will also master Cursor's keyword and semantic search features to navigate and understand code effectively within a familiar editing environment.

In this lesson, we'll explore how Cursor helps us debug and refactor existing code. We're intentionally starting with a simple example to concentrate on learning how to use Cursor's features effectively. The objective is to build confidence in the workflow and tools before we advance to a more complex project later.

We'll use Cursor to:

- Identify and fix runtime errors or faulty logic.
- Detect bad practices like redundant code or inconsistent naming.

- Break down large functions into smaller, cleaner parts.
- Apply standard refactoring techniques to improve structure and maintainability.
- Locate relevant logic using both keyword and semantic search.

Debugging and refactoring are part of the natural development process, and Cursor is designed to support both. By selecting code, submitting prompts, and reviewing suggestions, we can improve code quality while staying focused on the broader task.

Starting with a buggy script#

Let's say we're handed a simple script that is meant to calculate the average of even numbers in a list. At first glance, it looks functional, but when we try to run it, we encounter a runtime error.

Here's the original code:

 Python 3.10.4

File C []
Ace Editor

```
1
2
3
4
5
6
7
8
9
10
11
12
13

def get_even_average(numbers):
    total = 0
    count = 0
    for num in numbers:
```

```
if num % 2 == 0:  
    total = total + num  
    count = count + 1  
  
else:  
    continue  
  
if count == 0:  
    return 0  
  
return total / count
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
□  
...  
buggy.py
```

We run the script using the integrated terminal in Cursor. As expected, it throws an error, `TypeError: can only concatenate str (not "float") to str`:

The screenshot shows a code editor interface with a dark theme. The left sidebar has tabs for 'OPEN EDIT...', 'BUGGY_CODE', 'OUTLINE', and 'TIMELINE'. The main area is titled 'buggy.py' and contains the following Python code:

```
buggy.py x
buggy.py
1 def get_even_average(numbers):
2     total = 0
3     count = 0
4     for num in numbers:
5         if num % 2 == 0:
6             total = total + num
7             count = count + 1
8         else:
9             continue
10        if count == 0:
11            return 0
12        return total / count
13
14 def main():
15     data = [1, 2, 3, 4, 5, 6, 7, 8]
16     average = get_even_average(data)
17     print("The average is: " + average)
18
19 main()
```

Below the code, the terminal tab is active, showing the output of running the script:

```
Problems Output Debug Console Terminal Ports
Asmat@Asmat-Batool-TCE-Lead buggy_code % python3 buggy.py
Traceback (most recent call last):
  File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 19, in <module>
    main()
    ^~~~~^
  File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 17, in main
    print("The average is: " + average)
    ^~~~~^
TypeError: can only concatenate str (not "float") to str
Asmat@Asmat-Batool-TCE-Lead buggy_code %
```

This error occurs because the code tries to concatenate a string with a float using the `+` operator, which is not allowed in Python. Next, we'll use Cursor to fix this issue and then continue improving the structure of the code through refactoring.

Fixing the runtime error with Cursor#

To begin debugging, we highlight the part of the code that triggered the error. In this case, it's the `print` statement inside the `main()` function:

Ace Editor

```
1  
print("The average is: " + average)  
  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

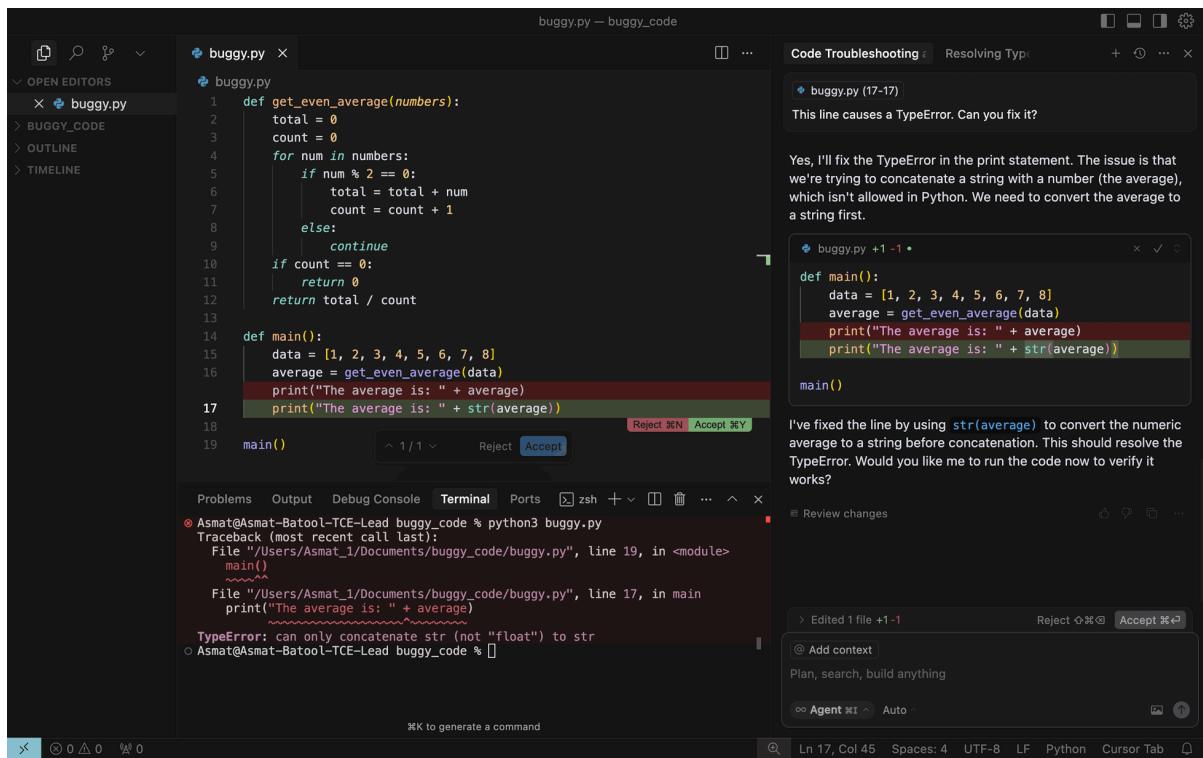
```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

HTML Content

With this line selected, we open Cursor's chat and type a prompt like:

Prompt: This line causes a `TypeError`. Can you fix it?

Cursor reviews the selected code and uses the prompt to generate a suggested fix. Here's the response:



The screenshot shows a VS Code interface. On the left, the 'buggy.py' file is open, with line 17 highlighted. The code contains a bug where it tries to concatenate a float with a string. The 'Code Troubleshooting' panel on the right has a message: "This line causes a `TypeError`. Can you fix it?". It suggests changing `print("The average is: " + average)` to `print("The average is: " + str(average))`. The 'Terminal' at the bottom shows the command `python3 buggy.py` being run, and the output shows the error: `TypeError: can only concatenate str (not "float") to str`. After accepting the fix, the terminal shows the correct output: `The average is: 4.5`.

We accept the suggestion and run the updated code. This time, the script executes correctly and prints the expected result:

The screenshot shows a Python code editor interface with two main panes. The left pane displays the file `buggy.py` containing the following code:

```
buggy.py
1 def get_even_average(numbers):
2     total = 0
3     count = 0
4     for num in numbers:
5         if num % 2 == 0:
6             total = total + num
7             count = count + 1
8         else:
9             continue
10    if count == 0:
11        return 0
12    return total / count
13
14 def main():
15     data = [1, 2, 3, 4, 5, 6, 7, 8]
16     average = get_even_average(data)
17     print("The average is: " + str(average))
18
19 main()
```

The right pane shows a "Code Troubleshooting" panel with the message: "This line causes a TypeError. Can you fix it?". Below it, a code editor shows the fix:

```
buggy.py +1 -1 ✓
def main():
    data = [1, 2, 3, 4, 5, 6, 7, 8]
    average = get_even_average(data)
    print("The average is: " + average)
    print("The average is: " + str(average))
main()
```

A terminal window at the bottom shows the execution of the code and the resulting error:

```
Asmat@Asmat-Batool-TCE-Lead buggy_code % python3 buggy.py
Traceback (most recent call last):
  File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 19, in <module>
    main()
           ^
  File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 17, in main
    print("The average is: " + average)
           ^
TypeError: can only concatenate str (not 'float') to str
● Asmat@Asmat-Batool-TCE-Lead buggy_code % python3 buggy.py
The average is: 5.0
○ Asmat@Asmat-Batool-TCE-Lead buggy_code %
```

This confirms the issue was with string formatting, as expected.

Attaching terminal output or screenshots to improve debugging#

In the above example, we highlighted the specific line that caused the error and asked Cursor to help fix it. This direct approach is often quick and effective for resolving isolated issues. In addition to highlighting code, Cursor also allows us to provide broader context from the terminal output or runtime logs. This becomes especially helpful when dealing with errors that involve stack traces, execution flow, or environment-specific problems. Let's see how we can attach terminal output or screenshots to guide the Cursor more accurately during debugging.

Example: Using terminal output for debugging#

Suppose we run the following script, as seen in the screenshot below, and encounter a runtime error:

```
buggy.py
1  def get_even_average(numbers):
2      even_numbers = [num for num in numbers if num % 2 == 0]
3      return sum(even_numbers) / len(even_numbers) if even_numbers else 0
4
5  def main():
6      data = "12345678"
7      average = get_even_average(data)
8      print("The average is: " + str(average))
9
10 main()

Problems Output Debug Console Terminal Ports
Asmat@Asmat-Batool-TCE-Lead buggy_code % python3 buggy.py
Traceback (most recent call last):
File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 10, in <module>
    main()
    ~~~~~~
File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 7, in main
    average = get_even_average(data)
File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 2, in get_even_average
    even_numbers = [num for num in numbers if num % 2 == 0]
    ~~~~~~

TypeError: not all arguments converted during string formatting
Asmat@Asmat-Batool-TCE-Lead buggy_code %
```

To inspect the error, we open the Cursor's chat, click `@Add context`, and attach the terminal output as shown below:

```
buggy.py
1  def get_even_average(numbers):
2      even_numbers = [num for num in numbers if num % 2 == 0]
3      return sum(even_numbers) / len(even_numbers) if even_numbers else 0
4
5  def main():
6      data = "12345678"
7      average = get_even_average(data)
8      print("The average is: " + str(average))
9
10 main()

Problems Output Debug Console Terminal Ports Terminal_bug
Asmat@Asmat-Batool-TCE-Lead buggy_code % python3 buggy.py
Traceback (most recent call last):
File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 10, in <module>
    main()
    ~~~~~~
File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 7, in main
    average = get_even_average(data)
File "/Users/Asmat_1/Documents/buggy_code/buggy.py", line 2, in get_even_average
    even_numbers = [num for num in numbers if num % 2 == 0]
    ~~~~~~

TypeError: not all arguments converted during string formatting
Asmat@Asmat-Batool-TCE-Lead buggy_code %
```

1 / 2

Then, we enter a prompt like:

Prompt: Please help me resolve this error.

Cursor reads the terminal output and replies with a clear explanation and suggested fix, as shown in the screenshot below:

The screenshot shows a Python file named `buggy.py` in the editor. On line 6, there is a code snippet: `data = [int(x) for x in "12345678"] # Convert string to list of integers`. A red arrow points to the modulo operator `%` in the string `"12345678"`, which is highlighted in yellow. A tooltip box titled "Error Resolution Assistance" appears, containing a "Prompt" section with the text "Please help me resolve this error." and a "Solution" section with the text "I'll help you resolve this error. The error message indicates there's an issue with the modulo operation (%) in your code, specifically suggesting that you're trying to perform string formatting instead of a modulo operation. Let me check the contents of your `buggy.py` file to see what's going on." Below the tooltip, a note says "I can see the issue now. The problem is that you're passing a string "12345678" to the `get_even_average` function, but the function expects a list of numbers. When you try to use the modulo operator (%) on a string, Python interprets it as a string formatting operation, which is why you're getting the error." It also suggests fixing the code by converting the string into a list of integers.

In this case, it correctly points out that the error is due to applying a modulo operation on a string instead of integers. The assistant suggests converting the string to a list of numbers before using it in the function.

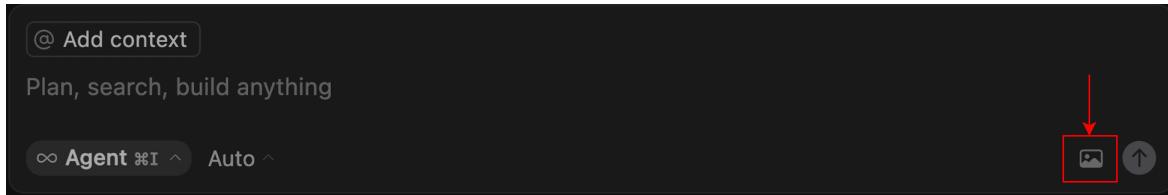
After applying the fix by accepting the suggested changes, we run the script again and confirm that the output is correct:

The screenshot shows the same `buggy.py` file after the fix. The problematic line is now: `data = [int(x) for x in "12345678"] # Convert string to list of integers`. The terminal output shows the command `python3 buggy.py` and the output `The average is: 5.0`. The "Error Resolution Assistance" panel is still visible, showing the resolved code and the fix applied.

Example: Attaching a screenshot for visual debugging#

Another helpful option in Cursor is the ability to attach a screenshot directly in the chat. This is especially useful when we want to debug issues related to UI layout, unexpected behavior in the browser, or visual bugs that may not be obvious in the code or terminal output.

For example, if a web application renders incorrectly or behaves in an unexpected way, we can take a screenshot of the browser and upload it using the image attachment icon in the chat input bar. Cursor will analyze the image, identify the potential issue, and suggest relevant fixes.



This feature expands our debugging options. We can use it to address a wide range of problems, including layout issues in the frontend, rendering mismatches, or state inconsistencies, all within the same collaborative AI workflow.

Refactoring with Cursor#

Now that the bug is fixed, let's step back and review the structure for improvements. The `get_even_average()` function works, but there are several opportunities to clean it up:

- The `else: continue` is unnecessary and can be removed.
- The way `total` and `count` are incremented could be simplified.
- The overall logic could be expressed more clearly with built-in Python features.

We start by selecting the entire `get_even_average()` function and entering a prompt like:

Prompt: Can you refactor this function to make it more concise and readable?

Cursor responds with an updated version of the function:

The screenshot shows the Refactoring Assistant interface. On the left, the original code in `buggy.py` is displayed:

```

1 def get_even_average(numbers):
2     total = 0
3     count = 0
4     for num in numbers:
5         if num % 2 == 0:
6             total = total + num
7             count = count + 1
8         else:
9             continue
10    if count == 0:
11        return 0
12    return total / count
13 even_numbers = [num for num in numbers if num % 2 == 0]
14 average = get_even_average(data)
15 print("The average is: " + average)
16
17 def main():
18     data = [1, 2, 3, 4, 5, 6, 7, 8]
19     average = get_even_average(data)
20     print("The average is: " + average)
21
22 main()

```

A tooltip from the Refactoring Assistant suggests refactoring the function to be more concise and readable. The proposed changes are shown in the right-hand pane:

```

1 total = total + num
2 count = count + 1
3 else:
4     continue
5 if count == 0:
6     return 0
7 return total / count
8 even_numbers = [num for num in numbers if num % 2 == 0]
9 return sum(even_numbers) / len(even_numbers) if even_numbers else 0
10
11 def main():
12     data = [1, 2, 3, 4, 5, 6, 7, 8]
13
14     average = get_even_average(data)
15     print("The average is: " + average)
16
17
18
19
20
21
22

```

The tooltip lists six key improvements made by the refactoring:

- Used list comprehension `[num for num in numbers if num % 2 == 0]` to filter even numbers in a single line
- Used Python's built-in `sum()` function instead of manually adding numbers
- Used `len()` to get the count of even numbers
- Used a conditional expression (ternary operator) to handle the case when there are no even numbers
- Removed unnecessary `continue` statement since it was at the end of the loop
- Eliminated the need for separate `total` and `count` variables

The new version is more Pythonic and achieves the same functionality in just two lines of code. It's easier to read and understand at a glance what the function does: it filters even numbers, then returns their average (or 0 if there are no even numbers).

Would you like me to make any additional improvements to the function?

The new version uses a list comprehension to filter even numbers, and a one-liner to calculate the average. It is shorter, easier to read, and removes unnecessary control flow.

We review the change and accept it. Cursor updates the file automatically.

Now our final code looks like this:

Ace Editor

```

1
2
3
4
5
6
7
8
9
10
def get_even_average(numbers):
    even_numbers = [num for num in numbers if num % 2 == 0]
    return sum(even_numbers) / len(even_numbers) if even_numbers else 0

```

```
def main():
    data = [1, 2, 3, 4, 5, 6, 7, 8]
    average = get_even_average(data)
    print("The average is: " + str(average))
main()

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

HTML Content

We run the code again to verify everything still works, and confirm that the output is unchanged:

```

buggy.py
1 def get_even_average(numbers):
2     even_numbers = [num for num in numbers if num % 2 == 0]
3     return sum(even_numbers) / len(even_numbers) if even_numbers else 0
4
5 def main():
6     data = [1, 2, 3, 4, 5, 6, 7, 8]
7     average = get_even_average(data)
8     print("The average is: " + str(average))
9
10 main()

```

Refactoring for Conciseness and Readability

buggy.py (1-12)

Can you refactor this function to make it more concise and readable?

I'll help you refactor the `get_even_average` function to make it more concise and readable. I'll use Python's built-in functions and list comprehension to simplify the code.

Let me edit the file:

```

total = total + num
count = count + 1
else:
    continue
if count == 0:
    return 0
return total / count
even_numbers = [num for num in numbers if num % 2 == 0]
return sum(even_numbers) / len(even_numbers) if even_numbers else 0

```

def main():
 data = [1, 2, 3, 4, 5, 6, 7, 8]

I've refactored the function to be more concise and readable. Here are the key improvements:

1. Used list comprehension `[num for num in numbers if num % 2 == 0]` to filter even numbers in a single line
2. Used Python's built-in `sum()` function instead of manually adding numbers
3. Used `len()` to get the count of even numbers
4. Used a conditional expression (ternary operator) to handle the case when there are no even numbers
5. Removed unnecessary `continue` statement since it was at the end of the loop
6. Eliminated the need for separate `total` and `count` variables

The new version is more Pythonic and achieves the same functionality in just two lines of code. It's easier to read and understand at a glance what the function does: it filters even numbers, then returns their average (or 0 if there are no even numbers).

Would you like me to make any additional improvements to the function?

Asmat@Asmat-Batool-TCE-Lead buggy_code % python3 buggy.py

The average is: 5.0

Asmat@Asmat-Batool-TCE-Lead buggy_code %

This example shows how Cursor can assist not only with fixing errors but also with improving the structure and quality of our code. It helps us move from working code to clean code with minimal friction.



You can ask Cursor to refactor an entire file by opening the file and prompting without selecting any specific code.

In addition to improving code readability, we can also ask Cursor to optimize for performance. For example, we can prompt:

Prompt: Can you refactor this function to reduce time complexity?

Or

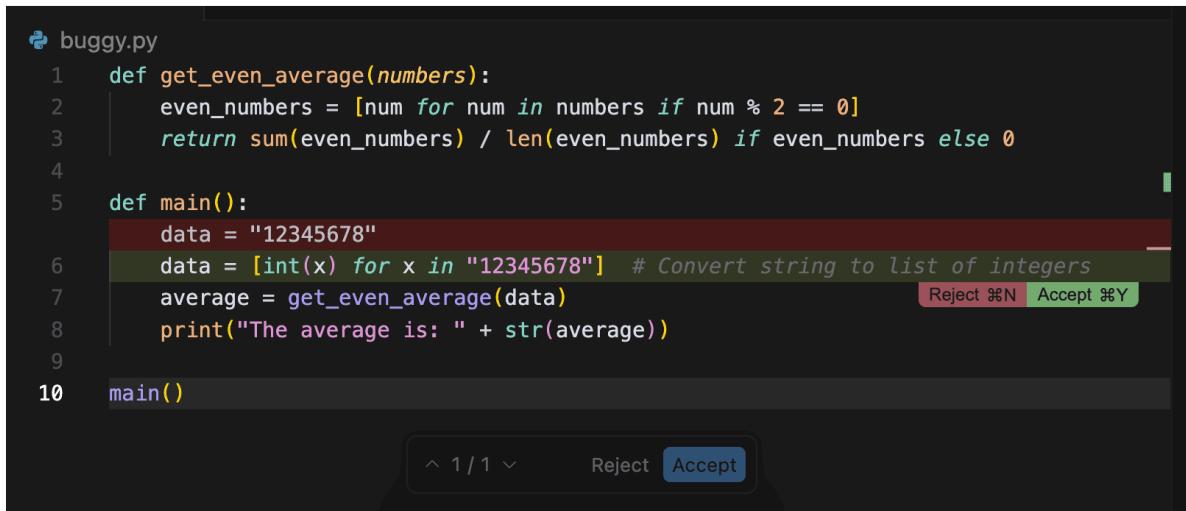
Prompt: Can you avoid redundant computations in this logic?

Cursor's refactoring capabilities are not limited to formatting or cleanup. It can also help improve the underlying logic, reduce computational overhead, and suggest more efficient patterns, especially useful in performance-critical code.

Reviewing and accepting changes with Diffs#

When Cursor proposes a fix or a refactor, it does not immediately modify our code without review. Instead, it opens an inline diff view directly in the editor. This lets us preview what changes are being made, why they were suggested, and where they will be applied.

As seen in the screenshots below, when we asked Cursor to fix the bug in our script, it generated a modified version of the code and showed it as a visual diff:



```
buggy.py
1 def get_even_average(numbers):
2     even_numbers = [num for num in numbers if num % 2 == 0]
3     return sum(even_numbers) / len(even_numbers) if even_numbers else 0
4
5 def main():
6     data = "12345678"
7     data = [int(x) for x in "12345678"] # Convert string to list of integers
8     average = get_even_average(data)
9     print("The average is: " + str(average))
10    main()

    ^ 1 / 1 ^ Reject Accept
```

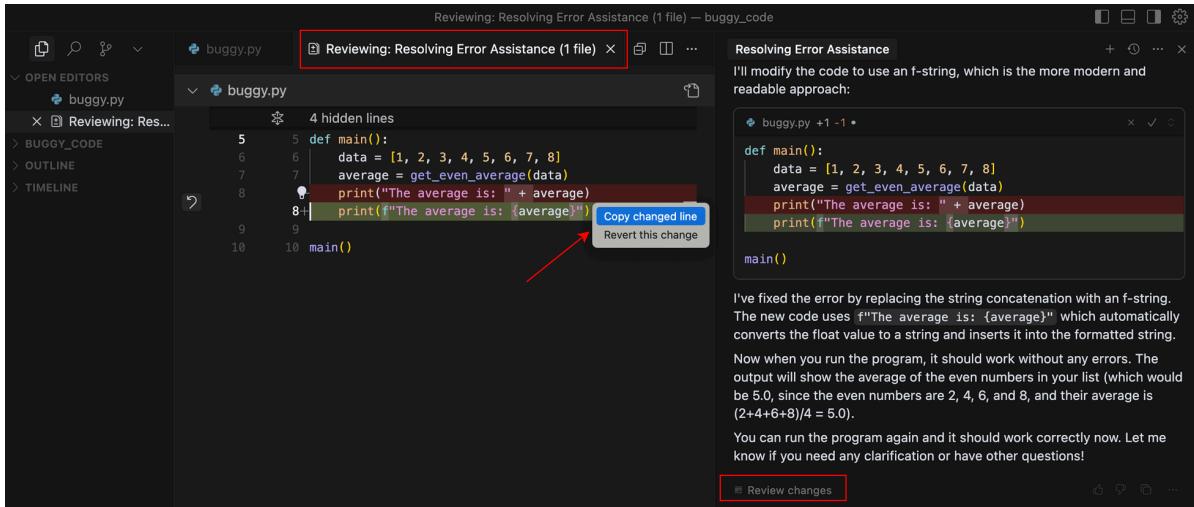
In this example, Cursor changed the value of `data` from a string to a list comprehension that converts each character to an integer. The deleted line is highlighted in red, and the added line is highlighted in green. This makes it easy to see what was changed. We can accept or reject each of the suggestions one by one using the green (accept) and red (reject) buttons.

At the bottom of the diff panel, we have options to accept or reject all suggestions at once:

- Accept all of the suggested changes by clicking the green “Accept” file button.
- Reject all of the suggested changes by clicking the “Reject” file if we prefer not to apply them.

These buttons provide a controlled editing experience, ensuring we are always in charge of what gets written into our files.

Below the AI response in the chat panel, we'll notice a review changes option. Clicking it opens a diff view where we can review all proposed edits across one or more files and decide whether to accept or reject them, copy the suggestion, or revert an accepted change to the previous state.



As we debug or refactor code, we often need to locate specific functions, logic, or past changes. Cursor offers powerful search capabilities to help with this, both through traditional keyword search and more advanced AI-powered semantic search.

Searching in Cursor: Semantic vs. keyword search#

When working with large codebases, it is important to quickly locate relevant logic or definitions. Cursor supports two types of search features that can help us find what we are looking for more effectively.

Keyword search#

Like Visual Studio Code, Cursor includes a standard keyword search panel. We can access this using the search icon in the left sidebar. This feature is useful when we know the exact text or function name we want to locate.

For example, in the screenshot below, we searched for `current_time` and the search panel returned the matching function in `app.py`.

```

app.py:
10 @app.route('/time')
11 def current_time():
12     return f'Current server time is: {datetime.now().strftime("%Y-%m-%d %H:%M:%S")}'

```

This type of search performs a literal match, helping us quickly jump to known strings in the codebase.

Semantic search in AI chat#

Cursor also provides a more advanced way to search the code using natural language through its built-in AI chat. This is a semantic search feature, which means it does not rely on exact keywords. Instead, it understands the intent behind our query and finds the relevant logic accordingly.

For example, in the screenshot below, we used Cursor's chat to ask, "Where is the server time logic implemented?" by entering the question as a natural language prompt. Cursor located the `current_time()` function in `app.py`, explained its purpose, and broke down the logic step by step.

Where is the server time logic implemented?

Let me search through the codebase to find where the server time logic is implemented.

I found the server time implementation in `app.py`. Let me read the relevant section to get more context.

The server time logic is implemented in `app.py` through a Flask route handler. Specifically:

1. The time functionality is exposed through the `/time` endpoint
2. The implementation uses Python's built-in `datetime` module
3. The time is formatted using `strftime()` to display in the format "YYYY-MM-DD HH:MM:SS"

The relevant code is:

```

from flask import Flask
from datetime import datetime
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
@app.route('/time')
def current_time():
    return f'Current server time is: {datetime.now().strftime("%Y-%m-%d %H:%M:%S")}'
if __name__ == '__main__':
    app.run(debug=True)

```

Add context

Plan, search, build anything

Agent Auto

By combining keyword and semantic search, we can navigate and understand our code more effectively. When we know the exact term, we can use the search panel. When we are trying to understand what a piece of code does or where a feature is implemented, the AI assistant can help us reason through it using conversational prompts.

This flexibility is especially helpful when working on unfamiliar codebases, debugging complex interactions, or reviewing team contributions.

Conclusion#

Debugging and refactoring are part of everyday software development. In this lesson, we used Cursor to handle both. With a few simple prompts and code selections, we fixed a minor runtime error and refactored a function to make it cleaner and easier to read. The example was intentionally simple, but it showed how Cursor can support real development tasks in a practical and efficient way.



Note: Cursor does not automatically validate logic correctness, enforce code style, or optimize performance unless we explicitly ask it to. It also does not perform static analysis or detect architectural issues on its own. Think of Cursor as a smart assistant that can support and accelerate our work, but we remain responsible for final review, testing, and adherence to project-specific standards.

6. Using Cursor AI to Explain and Document Code Efficiently

Clip source: [Using Cursor AI to Explain and Document Code Efficiently](#).

[Home](#)

>

[Courses](#)

>

[Code Smarter with Cursor AI Editor](#)

AI-Driven Code Explanation and Documentation

Explore how to leverage Cursor's AI features to understand unfamiliar code, generate clear documentation including docstrings and comments, summarize files, and produce project-level documentation. This lesson helps you improve code clarity and maintainability directly in your coding environment without switching tools.

In this lesson, we'll explore how Cursor can help us understand and document code. Whether we're reviewing someone else's work, returning to an old project, or trying to get up to speed quickly, Cursor's AI features can save time and reduce confusion.

We'll learn how to:

- Use Cursor to explain the selected code in plain language.
- Understand complex logic by asking focused questions.
- Generate comments and docstrings using the assistant.
- Produce project-level documentation and summaries with minimal manual effort.

Instead of writing everything from scratch, we'll focus on how Cursor can assist us in understanding what code does and how to document it clearly for ourselves and others.

Starting with a code snippet that needs explanation#

Let's say we come across the following piece of code in a project we didn't write ourselves. Before making any changes, we want to understand what it does.

Ace Editor

```
1  
2  
3  
4  
5  
6
```

```
7
8
9
10
11
12
class RateLimiter:
    def __init__(self, max_calls, window_seconds):
        self.max_calls = max_calls
        self.window_seconds = window_seconds
        self.call_times = []
    def allow_request(self, current_time):
        self.call_times = [t for t in self.call_times if current_time -
t < self.window_seconds]
        if len(self.call_times) < self.max_calls:
            self.call_times.append(current_time)
            return True
        return False
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```



There are no comments or docstrings. The method names give us a hint, but the actual logic needs a closer look. Instead of reading through it line by line

and trying to piece it together manually, we'll use Cursor to help explain what this class does and how each part works.

Using Cursor to understand unfamiliar code#

We start by selecting the `allow_request` method and opening the Cursor chat. Since there's no existing documentation or comments, we ask:

Prompt: Can you explain what this function does?

Cursor reads the code context and responds with a plain-language explanation. Here's an example of what the assistant returns:

The screenshot shows the Cursor AI interface. On the left, a code editor displays a Python file named `unknown.py` with the following code:1 class RateLimiter:
2 def __init__(self, max_calls, window_seconds):
3 self.max_calls = max_calls
4 self.window_seconds = window_seconds
5 self.call_times = []
6
7 def allow_request(self, current_time):
8 self.call_times = [t for t in self.call_times if current_time - t < self.window_seconds]
9 if len(self.call_times) < self.max_calls:
10 self.call_times.append(current_time)
11
12 return True
13
14 return FalseOn the right, a "Understanding the allow_request Function" panel provides an explanation:

I'll explain what this `allow_request` function does. It appears to be implementing a rate limiting mechanism, commonly known as a "sliding window" rate limiter. Let me break it down:

1. The function takes a `current_time` parameter and is part of a class (as indicated by the `self` parameter).
2. The function maintains a list called `call_times` that stores timestamps of previous requests.
3. Here's what the function does step by step:
 - a. First, it filters the `call_times` list to only keep timestamps that are within the last `window_seconds` period:

```
self.call_times = [t for t in self.call_times if current_time - t < self.window_seconds]
```

This effectively removes any old timestamps that are outside the current time window.
 - b. Then it checks if the number of requests in the current window is less than the maximum allowed calls (`self.max_calls`):

```
if len(self.call_times) < self.max_calls:
```
 - c. If there's room for more requests:
 - It adds the current timestamp to the list
 - Returns `True` to indicate the request is allowed

```
self.call_times.append(current_time)
```
 - d. If the maximum number of calls has been reached, it returns `False` to indicate the request should be rejected.

This is a common pattern used in rate limiting to ensure that a service or API doesn't get overwhelmed with too many requests in a given time period. For example, if `max_calls` is set to 100 and `window_seconds` is set to 60, this would allow a maximum of 100 requests per minute. The "sliding window" aspect means that the time window moves forward with each new request, rather than being fixed to specific time blocks. This provides a more smooth and fair rate limiting mechanism compared to fixed time windows.

This response helps us quickly understand the logic without stepping through the code manually. If we want to go deeper, we can ask follow-up questions like:

- What happens if the window is set to 0?
- How could this scale with high request volume?

Cursor stays in context and can build on earlier questions to help us explore different parts of the code's behavior.



Cursor's AI chat has project-level awareness when opened within a workspace, even if we do not select code or attach files as context. This allows us to ask broad questions like "What does this project do?", "Summarize the main components" or "Create project documentation," and receive responses based on the full codebase.

This project-level understanding depends on Codebase Indexing being enabled and fully completed. If indexing is not finished or disabled, Cursor's responses will be limited to currently open files or manually added context. When we want to focus on specific parts of the code, it is helpful to provide targeted input by selecting the relevant section or using the `@Add context` option. This ensures that Cursor emphasizes the part we care about in its response.

Generating docstrings and comments with Cursor#

Once we understand what the code does, we can ask Cursor to help document it. We start by selecting the entire `RateLimiter` class and entering a prompt like:

Prompt: Add docstrings and comments to this class and its methods.

Cursor processes the code and suggests clear, concise documentation. Here's an example of what it generates:

unknown.py

```
1 class RateLimiter:
2     """
3         A rate limiter implementation that restricts the number of requests allowed within a time window.
4     """
5     This class implements a sliding window rate limiting mechanism that tracks request timestamps
6     and enforces limits based on the maximum number of calls allowed within a specified time window.
7
8     Attributes:
9         max_calls (int): Maximum number of calls allowed within the time window
10        window_seconds (int): Time window in seconds for rate limiting
11        call_times (list): List of timestamps for recent calls within the window
12
13    def __init__(self, max_calls, window_seconds):
14        """
15            Initialize the rate limiter with specified limits.
16
17            Args:
18                max_calls (int): Maximum number of calls allowed within the time window
19                window_seconds (int): Time window in seconds for rate limiting
20
21        """
22        self.max_calls = max_calls
23        self.window_seconds = window_seconds
24        self.call_times = []
25
26    def allow_request(self, current_time):
27        """
28            Check if a new request should be allowed based on the rate limiting rules.
29
30            This method implements a sliding window approach:
31            1. Removes timestamps older than the window period
32            2. Checks if adding a new request would exceed the maximum calls
33            3. If allowed, adds the current timestamp to the tracking list
34
35            Args:
36                current_time (float): Current timestamp in seconds
37
38            Returns:
39                bool: True if the request is allowed, False if it should be rate limited
40
41            # Remove timestamps that are outside the current window
42            self.call_times = [t for t in self.call_times if current_time - t < self.window_seconds]
43
44            # Check if value under the maximum calls limit
45            if len(self.call_times) < self.max_calls:
46                self.call_times.append(current_time)
47
48            return True
49
50        return False
```

Add Docstrings and Comments to Class

unknown.py (1-12)

Add docstrings and comments to this class and its methods.

I'll help you add comprehensive docstrings and comments to the `RateLimiter` class. I'll edit the file to include clear documentation that explains the purpose, parameters, and behavior of the class and its methods.

unknown.py 36+

```
A rate limiter implementation that restricts the number of requests allowed within a time window.

This class implements a sliding window rate limiting mechanism that tracks request timestamps and enforces limits based on the maximum number of calls allowed within a specified time window.

Attributes:
max_calls (int): Maximum number of calls allowed within the time window
window_seconds (int): Time window in seconds for rate limiting
call_times (list): List of timestamps for recent calls within the window
```

I've added comprehensive documentation to the `RateLimiter` class including:

1. A detailed class docstring that explains:
 - The overall purpose of the class
 - The implementation approach (sliding window)
 - All class attributes and their meanings
2. Method docstrings for both `__init__` and `allow_request` that include:
 - Description of what the method does
 - Detailed explanation of the implementation steps (for `allow_request`)
 - Args section with parameter descriptions
 - Returns section (for `allow_request`)
3. Inline comments that explain the key operations within the `allow_request` method

The documentation follows Python's standard docstring format and provides clear information about how to use the class and what to expect from its methods. The comments make the code more maintainable and easier to understand for other developers.

Is there anything specific about the documentation that you'd like me to clarify or expand upon?

Review changes

Edited 1 file +36

Reject ➔ Auto ➔ Accept ➔

Add context

Plan, search, build anything

Agent ➔ Auto ➔

We review the AI-generated suggestions and, when necessary, refine the wording via prompting to align with our project's tone and target audience.

i

Cursor does not insert these suggested edits immediately. Instead, it shows a diff view where we can preview the changes across one or more files and

choose to accept or reject them. This gives us full control over what is added to our codebase.

Below is the updated code, now including docstrings and comments added using Cursor.



Python 3.10.4

File Code []
Ace Editor

```
1
2
3
4
5
6
7
8
9
10
11
12
13
class RateLimiter:
    """
    A rate limiter that implements a sliding window rate limiting mechanism.
    """

    This class allows a maximum number of requests within a specified time
    window.

    It uses a sliding window approach, which is more accurate than fixed window
    rate limiting as it only considers requests within the specified time window.

    Attributes:
        max_calls (int): Maximum number of requests allowed in the time window
        window_seconds (int): Size of the time window in seconds
        call_times (list): List of timestamps when requests were made
    """
```

Attributes:

```
max_calls (int): Maximum number of requests allowed in the time window
window_seconds (int): Size of the time window in seconds
call_times (list): List of timestamps when requests were made
```

"""

XX
□
...

The core advantage of using Cursor over a standalone LLM or chat assistant is not just the AI's ability to generate explanations or docstrings. It's that everything happens in context, inside the editor. There's no copy-pasting, no jumping between tools, and no manual effort to insert responses back into files.

That smooth, in-place workflow is exactly what makes Cursor feel like an integrated development partner rather than just an external helper.

Generating summaries and project-level documentation#

In addition to commenting on specific code, Cursor can help us document broader structures. For example, if we want to generate a summary of what a file or module does, we can select the entire file and prompt:

Prompt: Summarize this file in one or two sentences.

Cursor reviews the full context and returns a short explanation of what the module is for and what its main components do. This is useful when preparing for code reviews, handoffs, or onboarding new team members.



Note that only the portion of the file within the model's context window (e.g., ~128K tokens for GPT-4) is actually processed. For very large files, learners

should break them into smaller chunks.

We can take it further by asking Cursor to:

- Write a usage example for the class or function.
- Generate a README section describing how to use the module.
- Add docstring templates across multiple files.
- Create a complete project documentation for sharing with others.

Here's an example prompt:

Prompt: Create a usage example for this `RateLimiter` class.

Cursor responded by creating a new file with two structured example functions, clearly annotated with comments and docstrings. This helps demonstrate not just how the class works, but also how to use it effectively in different scenarios.



This kind of autonomous action, such as creating new files or writing directly into the editor, is available in Agent mode. In Manual or Ask modes, Cursor will only suggest code in the chat panel and leave it to us to decide whether and where to insert it.

Here is the result shown directly in the editor:

The screenshot shows a code editor with two files open: `rate_limiter_example.py` and `rate_limiter_example.py (new)`. The `rate_limiter_example.py` file contains Python code demonstrating the use of a `RateLimiter` class. The `rate_limiter_example.py (new)` file is a new file created by Cursor, containing a usage example for the `RateLimiter` class. The right pane of the editor displays the `Understanding Rate Limiting Function` interface, which includes a message from Cursor suggesting a usage example for the `RateLimiter` class, and a note that it will create a new file with example code that shows different use cases.

```
from unknown import RateLimiter
import time

def example_basic_usage():
    """
    Basic example showing how to use the RateLimiter class.
    This example demonstrates a rate limit of 3 requests per 5 seconds.
    """
    # Create a rate limiter that allows 3 requests per 5 seconds
    limiter = RateLimiter(max_calls=3, window_seconds=5)

    # Simulate some requests
    current_time = time.time()

    # First request - should be allowed
    print("Request 1:", limiter.allow_request(current_time)) # True

    # Second request - should be allowed
    print("Request 2:", limiter.allow_request(current_time + 1)) # True

    # Third request - should be allowed
    print("Request 3:", limiter.allow_request(current_time + 2)) # True

    # Fourth request - should be rejected (exceeds max_calls)
    print("Request 4:", limiter.allow_request(current_time + 3)) # False

    # Wait for window to expire and try again
    print("Request 5 (after window):", limiter.allow_request(current_time))

def example_api_rate_limiting():
    """
    Example showing how to use RateLimiter in an API context.
    This simulates an API endpoint with rate limiting.
    """
    # Create a rate limiter for API calls (100 requests per minute)
    api_limiter = RateLimiter(max_calls=100, window_seconds=60)

    def make_an_request():
        ...


# rate_limiter_example.py (new) +60 -1 *
from unknown import RateLimiter
import time

def example_basic_usage():
    """
    Basic example showing how to use the RateLimiter class.
    This example demonstrates a rate limit of 3 requests per 5 seconds.
    """
    # Create a rate limiter that allows 3 requests per 5 seconds
    limiter = RateLimiter(max_calls=3, window_seconds=5)

    # Simulate some requests
    current_time = time.time()

    # First request - should be allowed
    print("Request 1:", limiter.allow_request(current_time)) # True

    # Second request - should be allowed
    print("Request 2:", limiter.allow_request(current_time + 1)) # True

    # Third request - should be allowed
    print("Request 3:", limiter.allow_request(current_time + 2)) # True

    # Fourth request - should be rejected (exceeds max_calls)
    print("Request 4:", limiter.allow_request(current_time + 3)) # False

    # Wait for window to expire and try again
    print("Request 5 (after window):", limiter.allow_request(current_time))
```

This helps make the code more approachable for others who may use it or modify it later.

Conclusion#

In this lesson, we explored how to use Cursor to explain unfamiliar code and generate helpful documentation. We saw how to:

- Use prompts to get plain-language explanations of code behavior.
- Generate docstrings and comments directly in the editor.
- Ask for summaries and usage examples across functions and files.

Instead of copying code into an external AI tool and manually editing results, we were able to do everything in place with Cursor. This seamless workflow makes it much easier to keep our projects well-documented and easy to understand, both for ourselves and for others.



- While Cursor can explain code, generate docstrings, and summarize files, it does not automatically validate logic correctness or performance unless explicitly prompted. It also does not run linters or enforce code style. We should review all generated content and validate it with tools like `flake8`, `mypy`, or our project's own style guide.
- AI models have a maximum context window, which defines how much input they can consider at once. If the context is too large, parts of it may be silently ignored, leading to incomplete or less accurate responses. To get the best results when summarizing or documenting large files or multiple files, consider breaking them into smaller sections and prompting Cursor to work on each part individually.

7. Build a Django Wordle Game with Cursor AI Editor

Clip source: [Build a Django Wordle Game with Cursor AI Editor](#)

[Home](#)



[Courses](#)



[Code Smarter with Cursor AI Editor](#)

Creating a Wordle Game with Cursor

Explore how to build a complete Wordle-style game in Django using Cursor's AI-assisted editor. Learn to plan, scaffold, debug, and push your project to GitHub while collaborating with AI to enhance your coding process.

In this lesson, we'll build a fully functional Wordle-style game using Django and Cursor's AI-assisted workflow. This project will help us apply everything we've learned in a real-world scenario.

We'll learn how to:

- Use a high-level prompt to plan and scaffold a complete Django project.
- Implement multi-file logic for a browser-based Wordle game.
- Collaborate with the AI agent to handle iterations and refine features.
- Debug tricky behavior using screenshots and contextual prompts.
- Use session-aware logic to maintain game state across guesses.
- Push our finished code to GitHub directly from Cursor using the Git integration.

Preview what we're building#

The Wordle game challenges players to guess a hidden five-letter word in six attempts. After each guess, the app returns feedback for each letter:



Wordle game interface

- **Green:** Correct letter in the correct position
- **Yellow:** Correct letter, wrong position
- **Gray:** Letter not in the word

Below, we can see the full project structure and code, all created and built using Cursor using natural language prompts. Click the “Run” button to launch the app and try it out.

If you’re using Safari and encounter an error when clicking the “New Game” button, it may be due to strict privacy settings that block essential cookies.

To resolve this:

1. Open Safari > Settings > Privacy.
2. Uncheck the following options:

Prevent cross-site tracking

Block all cookies

3. Refresh the page and try again.

These settings ensure the game can store a secure CSRF token, which is required to start a new session.

```
/  
  v  
  game  
    >  
    __pycache__  
      __init__.cpython-313.pyc  
      admin.cpython-313.pyc  
      apps.cpython-313.pyc  
      models.cpython-313.pyc  
      urls.cpython-313.pyc  
      views.cpython-313.pyc  
    >  
    migrations  
      >  
      __pycache__  
        __init__.cpython-313.pyc  
        __init__.py  
      >  
      templates  
    >  
    game  
      wordle.html  
      __init__.py  
      admin.py  
      apps.py
```

```
python
models.py
python
tests.py
python
urls.py
python
views.py
>
wordle_django
>
__pycache__
  __init__.cpython-313.pyc
  settings.cpython-313.pyc
  urls.cpython-313.pyc
  wsgi.cpython-313.pyc
  .DS_Store
python
__init__.py
python
asgi.py
python
settings.py
python
urls.py
python
wsgi.py
  .DS_Store
  README.md
```

db.sqlite3



manage.py



requirements.txt

Ace Editor

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21
```

```
import random  
import json  
from django.shortcuts import render, redirect  
from django.http import JsonResponse  
from django.views.decorators.csrf import csrf_exempt  
from collections import Counter  
# A simple list of 5-letter words (expand as needed)  
WORD_LIST = [
```

```

'CRANE', 'PLANT', 'BRAVE', 'SHINE', 'GHOST', 'WORDS', 'APPLE', 'MOUSE', 'TRAIN',
'SHEET',

'CLEAN', 'BRAIN', 'SUGAR', 'WATER', 'EARTH', 'HEART', 'SWEET', 'NIGHT', 'LIGHT',
'SOUND'
]

MAX_GUESSES = 6

def get_solution(request):
    solution = request.session.get('solution')
    if not solution:
        solution = random.choice(WORD_LIST)
        request.session['solution'] = solution

```

XX
 XXX
 XXX
 XXX
 XXX
 XXX
 XXX
 XXX
 XXX



Your app can be found at: <https://ed-6007281325441024.educative.run>

Complete Wordle game code

Our version of the game includes:

- A Django backend for routing, logic, and session handling.
- A basic HTML template for displaying the game interface and feedback.
- A clean, multi-file project layout.

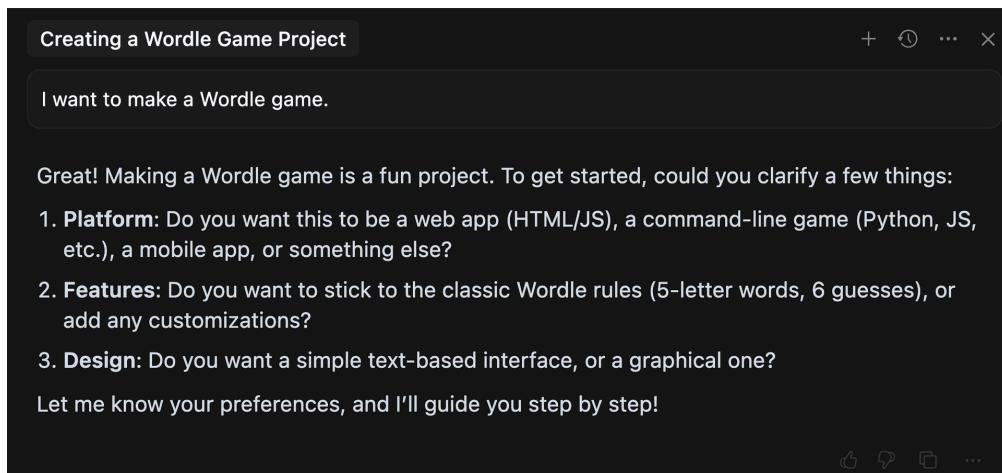
- Git integration to track and publish changes.

Planning and building the Wordle project#

To build our Wordle game, we will work with Cursor's Agent mode, which acts like an intelligent coding partner. We will begin by typing a single high-level prompt into Cursor's chat with the Agent mode selected:

Prompt: I want to make a Wordle game.

Cursor responds with a few follow-up questions to help clarify our intent and gather more details before suggesting any code.



Follow-up questions

These follow-up questions help narrow the scope and align with our goal: a simple, browser-based Wordle clone using Django.



Note: While high-level prompts like the above can lead to helpful follow-ups, the quality of the response depends on the project context, model selected, and clarity of our input. If the follow-up seems vague or off-target, it helps to guide the assistant with more specific information.

Sharing a complete plan before coding#

Once we answer the follow-up questions, the agent shares a full project plan.

Creating a Wordle Game Project

Absolutely! Here's a clear plan for your Django Wordle game, tailored to your requirements:

Tech Stack

- **Backend:** Django (Python)
- **Frontend:** HTML/CSS/JavaScript (Django templates)
- **Game Rules:** 5-letter words, 6 guesses, random word from a list

Features

- User enters guesses via a web form
- App gives feedback (correct letter & position, correct letter wrong position, not in word)
- Game state persists per user session

Steps

1. Create Django Project & App

- Set up a new Django project and a game app.

2. Word List

- Add a static list of 5-letter words (as a file or Python list).

3. Game Logic

- Implement logic to:
 - Select a random word at the start of a session
 - Accept user guesses
 - Provide feedback for each guess (color coding: green/yellow/gray)

4. Session Management

- Use Django sessions to store the target word, guesses, and feedback per user.

5. Frontend

- Build a simple, interactive game board using Django templates and CSS for feedback colors.
- Use a form for user input and display previous guesses with feedback.

Next Step

Would you like me to:

- Generate the commands to create the Django project and app?
- Or do you already have a Django project set up and want to start with the app/game logic?

Let me know your starting point!

Project plan

This allows us to review the plan before any files are created. If something didn't look right, we could adjust the scope or stack. Once the plan looks

good, we can proceed with creating the project.

Scaffolding and writing code with agent#

With our approval, the agent moves forward to:

- Scaffold a complete Django project.
- Create all core files, including `views.py`, `urls.py`, and `settings.py`.
- Set up template and static folders
- Write boilerplate code for routing and views.
- Generate the initial HTML structure for the game.

ⓘ

Before making changes to the codebase, Cursor shows a diff preview for each file involved. This allows us to review the proposed edits, accept or reject them, and stay fully in control of what gets applied.

Ace Editor

```
1
2
3
4
5
6
7
8
9
10
11
12
13
wordle_django/
└── game/
    ├── views.py
    ├── urls.py
    └── templates/game/wordle.html
        └── ...
└── wordle_django/
    ├── settings.py
    └── urls.py
```

```
|   └ ...  
|   └ manage.py  
|   └ requirements.txt  
└ README.md
```

XX
XX
XX
XX
XX
XX
XX
XX
XX

HTML Content

Project structure

This isn't a one-shot response. The agent builds out each part of the app step by step while maintaining context across files. As new functionality is added, relevant files are updated together. For example, when the backend view is added, routing is handled automatically in `urls.py`. This kind of coordinated, multi-file editing is powered by Agent mode, which lets Cursor apply context-aware changes across the project. It uses the codebase context available in the chat session to identify where updates are needed.



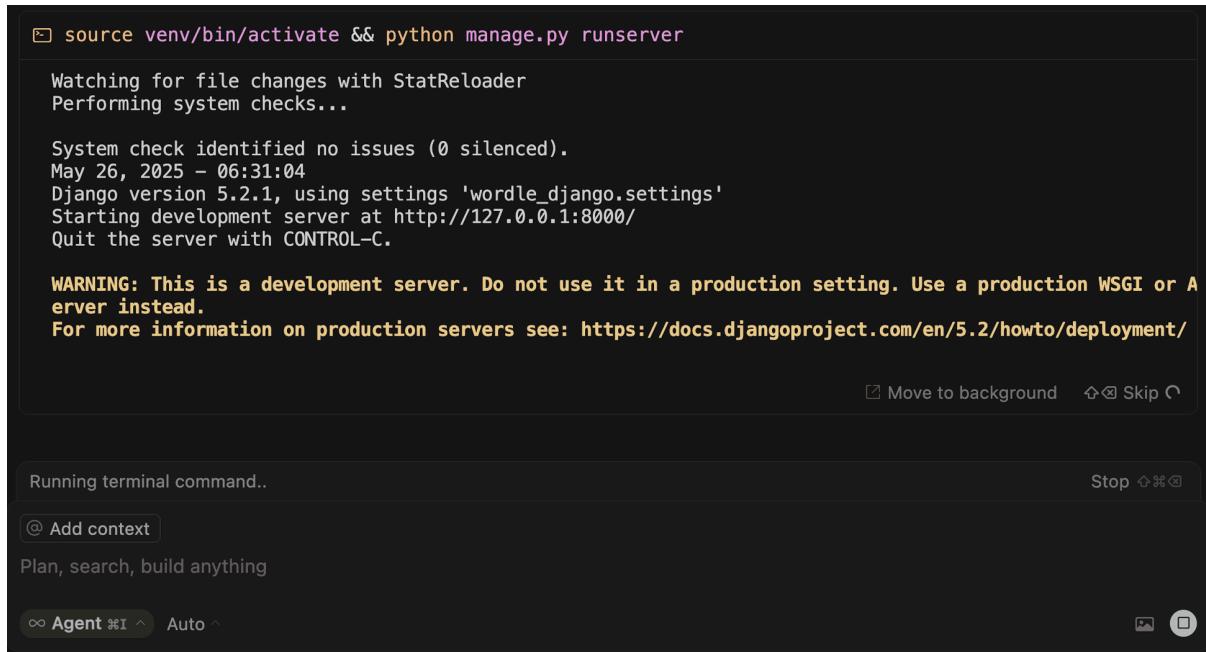
Note: Cursor maintains a full understanding of the codebase when working inside a project. However, since it's powered by AI, it may occasionally lose track of specific details or dependencies. If that happens, we can guide it by manually selecting files or using the `@Add context` option to refocus the conversation.

Running the project#

When the core functionality is in place, we ask the agent to run the app.

Prompt: Run the app.

The agent installs all required dependencies and launches the Django development server directly from the integrated terminal. The agent gets our approval before running each command. We can also change this setting to auto-run.



A screenshot of the Cursor AI interface showing a terminal window. The terminal output is as follows:

```
source venv/bin/activate && python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).
May 26, 2025 - 06:31:04
Django version 5.2.1, using settings 'wordle_django.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

WARNING: This is a development server. Do not use it in a production setting. Use a production WSGI or A
server instead.
For more information on production servers see: https://docs.djangoproject.com/en/5.2/howto/deployment/
```

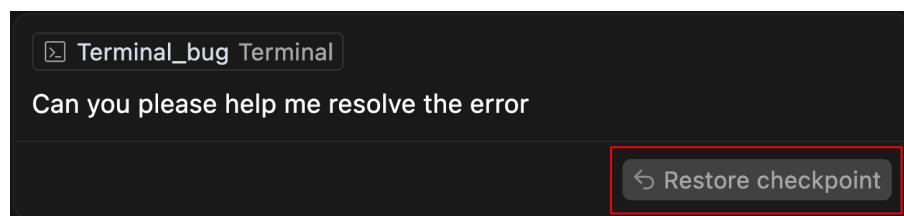
The terminal has a "Move to background" button and a "Skip" button at the bottom right. Below the terminal, there's a text input field with placeholder text "Plan, search, build anything" and a "Stop" button. At the bottom left, there's an "Agent" section with "Auto" and a dropdown arrow. On the far right, there are three small icons.

Running the app

If a system-level issue appears, such as permission errors or a missing runtime, the agent diagnoses the problem and provides step-by-step guidance to fix it. In our case, we faced one such system-level environment error early on, and the agent walked us through resolving it before retrying the run.



If Cursor begins to go off track, we can stop the agent and reframe our prompt to guide it more effectively. We can also use the “Restore checkpoint” option to return the chat to a previous state.



A screenshot of the Cursor AI interface showing a message in the chat:

Terminal_bug Terminal
Can you please help me resolve the error

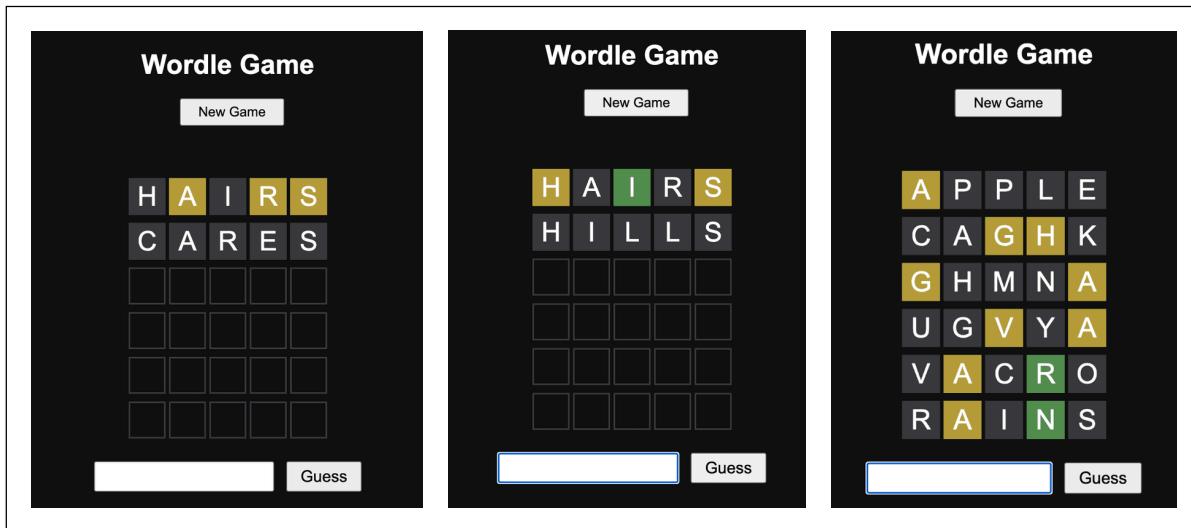
At the bottom right, there is a button labeled "Restore checkpoint" with a red border around it.

Debugging unexpected behavior#

We have leveraged two of Cursor’s core capabilities to debug our Wordle app:

- **Error handling with screenshots:** If we run into a runtime error or unexpected behavior, we can upload a screenshot directly into the chat. The Cursor agent will analyze the image, interpret the issue, and suggest relevant fixes based on what it sees.
- **Context-aware code navigation:** We can ask the agent to locate where a particular function, behavior, or piece of logic is implemented. Instead of manually searching through the codebase, Cursor returns precise references to relevant files, code snippets, and even surrounding logic, helping us stay focused and save time.

While testing the Wordle game, we noticed a few behaviors that suggested something wasn't quite right. For example, a letter marked yellow in one guess later turned gray. Since yellow indicates that the letter exists somewhere in the solution, it should not disappear in subsequent guesses. Similarly, a letter marked green (correct letter and position) should not become gray when guessed again in the same position.



Unexpected outputs

We ask the Cursor:

Prompt: I have attached the screenshots of the game. There is something wrong, please see what could be the possible issues.

Cursor understood the problem just by analyzing the game image. Here's how it interprets the issues based on the screenshot:

 Image

I have attached the screenshot of the game. There is something wrong, plesae see what could be the possible issues.

Thank you for the screenshot. Let's analyze the issue in detail:

What the Screenshot Shows

First guess: HAIRS

- H: yellow
- A: yellow
- I: yellow
- R: yellow
- S: yellow

Second guess: CARES

- C: gray
- A: gray
- R: gray
- E: gray
- S: gray

What's Wrong?

- In the first guess, all the letters (H, A, I, R, S) are marked yellow, which means they are present in the solution but in the wrong position.
- In the second guess, some of the same letters (A, R, S) are now marked gray, which means they are not in the solution at all. This is inconsistent with the previous feedback.
- In Wordle, if a letter is marked yellow in one guess, it should not be marked gray in another guess unless the number of times it appears in the guess exceeds the number of times it appears in the solution.

Interpreting the screenshot of the wrong output/error

It then list the possible issues:

Possible Issues

- The coloring logic for marking letters as yellow or gray is likely incorrect.
- The game may not be properly tracking the number of occurrences of each letter in the solution versus the guess.
- There may be a bug in how the game updates the state for repeated letters or for letters that have already been marked in previous guesses.

After providing some more context to the Cursor that this issue happens when the code is run in a different environment/platform other than the Cursor. The same code works fine in the Cursor. We also shared what problem we suspect from our understanding that the solution word might be changing after every guess. Through a few iterations, we worked together to isolate the root cause and received precise, file-level recommendations to resolve it.

We provided additional context to Cursor, explaining that the issue occurred only when running the code in an environment outside of Cursor. Within Cursor, the same code worked as expected. We also shared our suspicion

that the solution word might be changing after each guess, which suggested a problem with session persistence.

With this insight, the agent quickly refined its analysis. Through a few focused iterations, we collaborated to identify the root cause and received clear, file-specific recommendations to resolve the issue.



Takeaway: Cursor works best as a collaborative development partner.

Whether we're planning a project, refining requirements, or debugging issues, clear and focused input helps the agent respond more effectively. The more context and precision we provide, the better the results. By guiding Cursor with intent, we streamline development, reduce friction, and build with greater clarity and efficiency.

Cursor Integration with developer tools: Pushing the code to GitHub#

Cursor offers seamless Git integration, enabling developers to manage version control and synchronize code with GitHub directly from the editor. After building and testing the Wordle application, we can publish our work to GitHub using Cursor's built-in Git integration.

- **Initialize Repository:** This option sets up a local Git repository in our current project folder, enabling version control features such as tracking changes, staging files, and committing updates directly within Cursor. It also gives us a reliable fallback. If a refactor or multi-step AI session results in unwanted changes, we can review the commit history and roll back to a previous state safely.
- **Publish to GitHub:** This option allows us to create a new repository on GitHub and link it to our local project. Once published, we can push commits, pull updates, and manage your GitHub repository without leaving the Cursor environment.



Note: You must have Git installed and configured on your system for version control to work. If it is not already installed, Cursor will prompt you to install it or help guide the setup process when you attempt to initialize a repository.

The screenshot shows the VS Code interface. On the left, the Source Control panel displays a message: "The folder currently open doesn't have a Git repository. You can initialize a repository which will enable source control features powered by Git." It includes two buttons: "Initialize Repository" and "Publish to GitHub". The main editor area contains a Python file named "views.py" with the following code:

```
views.py
1 import random
2 import json
3 from django.shortcuts import render, redirect
4 from django.http import JsonResponse
5 from django.views.decorators.csrf import csrf_exempt
6 from collections import Counter
7
8 # A simple list of 5-letter words (expand as needed)
9 WORD_LIST = [
10     'CRANE', 'PLANT', 'BRAVE', 'SHINE', 'GHOST', 'WORDS', 'APPLE', 'MOUSE', 'TRAIN',
11     'CLEAN', 'BRAIN', 'SUGAR', 'WATER', 'EARTH', 'HEART', 'SWEET', 'NIGHT', 'LIGHT'
12 ]
13
14 MAX_GUESSES = 6
15
16 def get_solution(request):
17     if 'solution' not in request.session:
18         request.session['solution'] = random.choice(WORD_LIST)
19     return request.session['solution']
20
21 def wordle(request):
22     # Ensure a solution is set in the session
23     get_solution(request)
24     # Reset guesses for a new game
25     request.session['guesses'] = []
26     return render(request, 'game/wordle.html')
27
28 def new_game(request):
29     request.session.flush() # Clear the session (removes solution and guesses)
```

The Git panel in Cursor shows options to initialize a Git repository and publish the project to GitHub, making it easy to enable version control and share code directly from the editor

Conclusion#

In this lesson, we brought together everything we've learned so far by building a complete Django-based Wordle game with Cursor. This project helped us move beyond simple examples and explore Cursor's full potential in a real development workflow. We planned, scaffolded, debugged, refined, and published an entire application.

Throughout the process, we learned how Cursor's agent can help us reason through ideas, generate and edit multi-file codebases, handle errors, and push projects to GitHub, all within a single interface. More importantly, we reinforced the value of working collaboratively with the AI. The clearer our direction and feedback, the more precise and helpful the results became.



Note: A potential improvement to this Wordle game would be to disable the input field after the player either guesses the correct word or exhausts all attempts. While we haven't implemented that here, it serves as a great reminder that every real-world project has room for refinement. Our primary focus in this lesson was to showcase how Cursor helps us plan, scaffold, debug, and iterate on a multi-file application. If we want to enhance the user

experience further, we can simply prompt Cursor to add this behavior and update the relevant code files. It's a great opportunity to continue building with AI support.

8. Productivity Best Practices for Efficient Coding with Cursor AI

Clip source: [Productivity Best Practices for Efficient Coding with Cursor AI](#)

[Home](#)

>

[Courses](#)

>

[Code Smarter with Cursor AI Editor](#)

Productivity Best Practices

Explore effective productivity best practices for using Cursor AI in software development. Understand how to manage context, utilize different chat modes, integrate Git, and review code changes to enhance efficiency and maintain clarity. This lesson helps you adopt habits that streamline your workflow and make AI assistance more effective during coding projects.

So far, we've explored Cursor's capabilities in debugging, writing, refactoring, and building full projects. In this lesson, we'll focus on how to use Cursor more intentionally, not just with its features but through a set of productive habits that elevate your experience.

From managing project context to integrating Git, small choices in how we interact with Cursor can make a big difference in efficiency and clarity. Let's walk through the practices that help developers stay in flow and make the most of what Cursor has to offer.



Use the right chat mode for the task#

Cursor offers three chat modes, Agent, Ask, and Manual,**each designed for different workflows:**

- **Agent mode** is ideal when you want Cursor to take initiative in planning, editing, or running code. It is great for interactive tasks like scaffolding a new project, debugging issues, or transforming large code sections.
- **Ask mode** is suited for getting answers or explanations without triggering automatic edits. It behaves more like a read-only advisor.
- **Manual mode** gives you complete control. Cursor won't include tools or context unless you explicitly provide it.

By switching modes based on intent, you can avoid surprises and stay aligned with your goal.

Add context before asking#

Cursor is powerful when it understands the full picture. Before sending prompts that depend on external files or logic, use the "@ Add Context" feature to attach relevant files, folders, recent changes, or terminal output.

For example, if you're asking for help refactoring a class, attach the entire file so the agent sees the full structure. If you're debugging an issue, include logs or test output.

Well-scoped context leads to better responses and reduces back-and-forth.

Start with a plan#

Instead of diving directly into implementation, consider starting a conversation by asking the agent to outline a plan. This is especially useful when:

- Adding a new feature
- Reorganizing part of the codebase
- Setting up a new project

For example:

"I want to add user login functionality. What files should be created or modified?"

Cursor will usually propose a step-by-step approach. You can confirm or adjust this plan before proceeding, which helps prevent rework and aligns expectations.

Review edits before accepting#

Whenever the Cursor agent proposes changes to your code, you'll see a diff view that lets you inspect, accept, or reject the changes. Use this to your advantage:

- Accept only what you understand and agree with.
- Reject or rephrase when the suggestion isn't quite right.
- Review changes across all affected files before committing.

This helps maintain clarity in your codebase and encourages a more thoughtful development process.

Use Git from the start#

Cursor integrates with Git and GitHub out of the box. Once you open a folder, you can initialize a Git repository using the Source Control tab, then publish it directly to GitHub.

Once connected, you can:

- Track code changes file by file.
- Commit and push updates without leaving the editor.
- Use Git history as context during AI conversations.

Setting this up early allows you to work confidently, knowing changes are versioned and recoverable.

Rename chat tabs for clarity#

Cursor lets you work across multiple chats, each tracking its own context and history.

Renaming tabs based on the task (e.g., "Fix navbar bug", "Auth refactor") helps you:

- Stay organized across multiple threads
- Resume past work easily
- Avoid losing valuable responses in generic "New Chat" tabs

This small habit leads to a cleaner, more focused workspace.

Keep prompts clear and targeted#

Rather than writing long, ambiguous prompts, break your requests down:

- State clearly what you want.
- Include relevant code or context.
- Ask one thing at a time when possible.

For example:

"Can you convert this function to async?"

"Where is the color-coding logic for the Wordle game?"

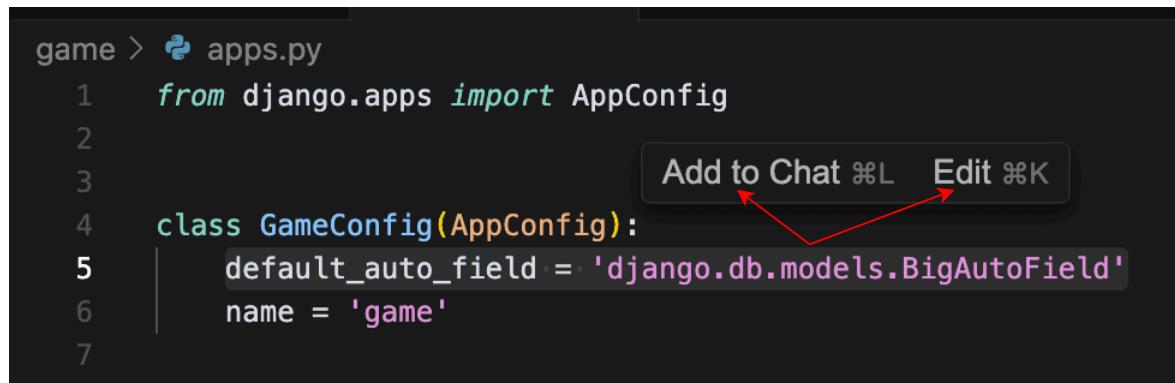
"Explain why this test is failing."

Specific prompts yield better and faster answers.

Combine inline tools with chat#

Cursor's power isn't limited to the chat sidebar. You can right-click on code to access inline tools like:

- Add to chat
- Edit with AI



A screenshot of a code editor showing a Python file named `apps.py`. The code defines a class `GameConfig` that inherits from `AppConfig`. The class has two attributes: `default_auto_field` set to `'django.db.models.BigAutoField'` and `name` set to `'game'`. A context menu is open over the code, with two options highlighted: `Add to Chat` and `Edit`. Red arrows point from the text "These are especially useful for quick fixes or localized changes." to both of these options.

```
game > 📁 apps.py
1   from django.apps import AppConfig
2
3   class GameConfig(AppConfig):
4       default_auto_field = 'django.db.models.BigAutoField'
5       name = 'game'
6
7
```

These are especially useful for quick fixes or localized changes. They help you stay focused without jumping in and out of conversations.



Inline tools are great for isolated logic, but for changes that affect multiple files or rely on project structure, use the chat with full context.

Use the terminal and output without leaving the editor#

Cursor includes a built-in terminal, letting you:

- Run scripts
- View logs and outputs
- Install dependencies
- Manage environments

The terminal launches in your project root directory, matching the open workspace folder. It behaves like any standard shell, which means environment variables, virtual environments, and path settings work just as they do in a standalone terminal. This consistency makes it easier to debug, experiment, and stay focused without switching tools.

Reflect and iterate with chat history#

Cursor saves your previous chats, and each one keeps track of the files and context you were working with. Use past threads to:

- Revisit decisions
- Copy useful responses
- Continue unfinished tasks

You can even branch off from older conversations by duplicating tabs. This is a simple way to explore alternatives without losing progress.

Conclusion#

Cursor is more than an AI-enhanced editor; it's a thinking partner that integrates into your building process. But like any tool, it shines when used with care.

By applying practices like setting context, reviewing code changes, planning tasks, and leveraging built-in tools, you'll get better results in less time. Cursor works best when paired with your judgment, clarity, and direction.

Make the tool work for you, and your development process becomes faster, cleaner, and more focused.

9. Final Thoughts on Building Projects with Cursor AI Editor

Clip source: [Final Thoughts on Building Projects with Cursor AI Editor](#)

[Home](#)

>

[Courses](#)

>

[Code Smarter with Cursor AI Editor](#)

Final Thoughts

Explore the completion of the Cursor AI Editor course, reviewing how to effectively write, debug, and manage code with AI assistance. Learn essential workflows including using chat modes, multi-file edits, and Git integration. Understand best practices for collaboration and productivity to confidently apply Cursor in your development projects.

Congratulations on completing the course!



Together, we've explored how Cursor can transform how we write, debug, and manage code. This journey has shown that Cursor is more than just an editor. It's a capable development companion that fits naturally into how we build real projects.

From writing simple scripts to developing a Django-based Wordle game, we've learned how Cursor works and how to work with it effectively.

What have we learned?#

- We began by understanding Cursor, which is built as a fork of Visual Studio Code and retains full compatibility with VS Code extensions and settings. We then learned how to install and navigate it.
- We used natural language prompts to generate files, scaffold functionality, and bring ideas to life with less manual effort. Depending on the task, we can use different chat modes, Agent, Ask, or Manual, to control how the AI responds. Each mode's behavior, such as whether changes are applied automatically or commands are run on our behalf, can be adjusted directly from the AI Chat Pane or in the settings under the "Features" section.
- We identified and resolved bugs, cleaned up code, and used AI to suggest more readable and maintainable solutions. In Agent mode, we accepted or rejected edits through diff previews, ensuring we reviewed multi-file changes before applying them.
- We explored how Cursor helps us understand unfamiliar code and generate useful documentation for collaboration and clarity. We also used "@Add context" and file selections to help Cursor focus on the right parts of the codebase when working across larger files or projects.
- We combined all these skills by planning and building a Wordle-style web application. Using Cursor, we worked across multiple files, resolved real issues, and pushed the code to GitHub.
- We wrapped up with productivity habits that help us stay focused and get more done using Cursor's built-in tools and smart workflows.

Key takeaways#

- **Collaboration is key:** The more clearly we communicate with Cursor, the better its responses will be. Providing context and asking focused questions leads to more accurate and useful suggestions.
- **Cursor is a complete development environment:** With code editing, AI assistance, terminal, Git integration, and chat history all in one place, we can reduce context switching and stay in the zone.
- **Treating Cursor like a teammate works best:** Cursor shines when we bring our judgment and reasoning to the table. It helps us move faster, but we guide the direction. While it is a powerful tool, we should always review AI-generated changes using linters, tests, or manual inspection before merging them into production code.

What's next?#

Now that we've built a solid foundation, we can:

- Apply the Cursor in our personal and professional projects.
- Try more advanced workflows like test generation, deployment automation, or CI integration.
- Share feedback, join the Cursor community, and keep learning from other developers.
- Keep experimenting with new ways to use AI in our everyday development.

Thanks for learning with us. This course wasn't just about features but about building a mindset for working with AI as part of our daily workflow. We now have the tools and techniques to write better code, stay productive, and solve problems confidently.