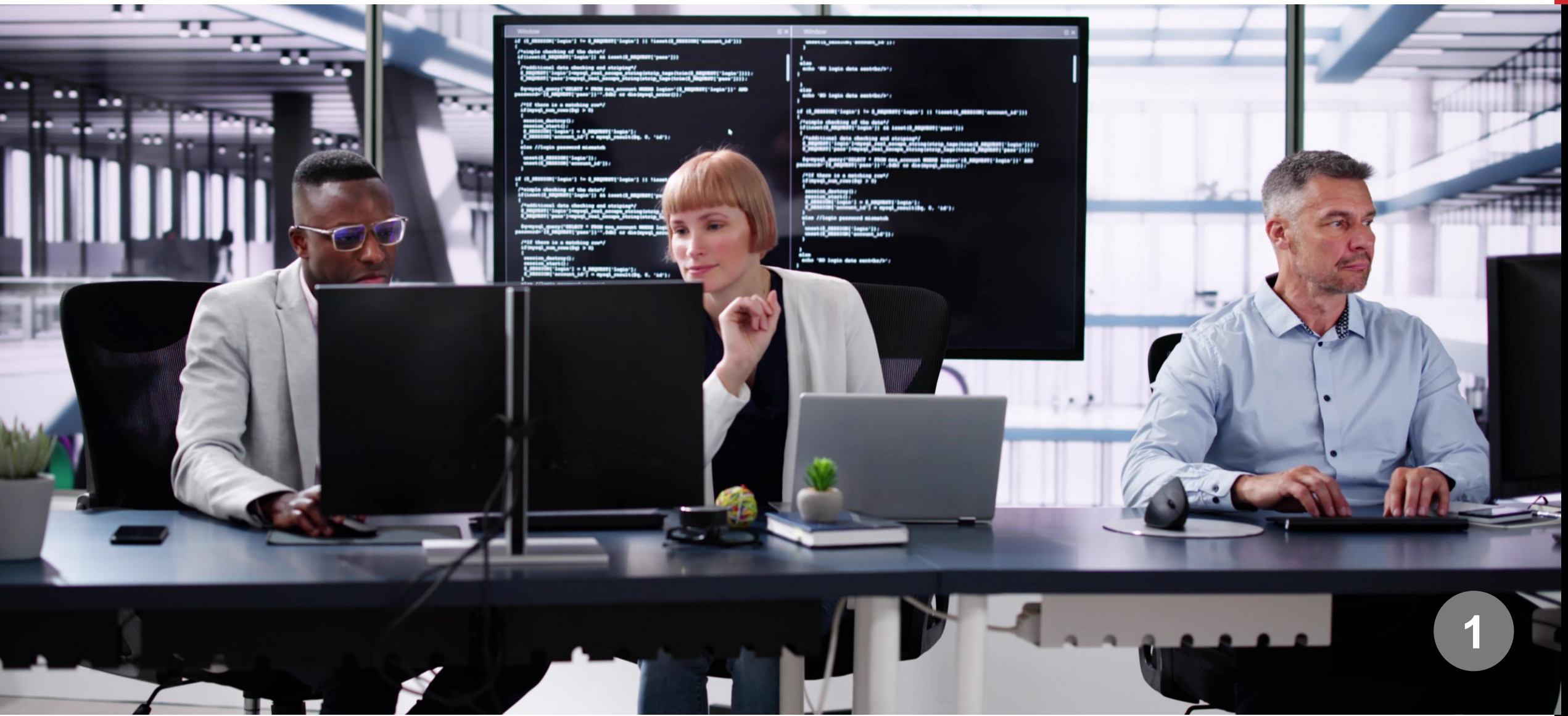
GITHUB COPILOT

FOR DEVELOPERS



TEKsystems Global Services

WORKFORCE DEVELOPMENT

PARTICIPANT GUIDE

Content Usage Parameters

Content refers to material including instructor guides, student guides, lab guides, lab or hands-on activities, computer programs, etc. designed for use in a training program

1

Content is subject to copyright protection

2

Content may only be leveraged by students enrolled in the training program

3

Students agree not to reproduce, make derivative works of, distribute, publicly perform and publicly display content in any form or medium outside of the training program

4

Content is intended as reference material only to supplement the instructor-led training

LOGISTICS

Class Hours:

- Instructor will set class start and end times.
- There will be regular breaks in class.

Telecommunication:

- Turn off or set electronic devices to silent (not vibrate)
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

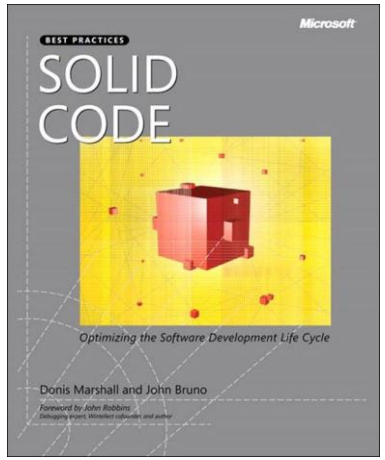
Miscellaneous:

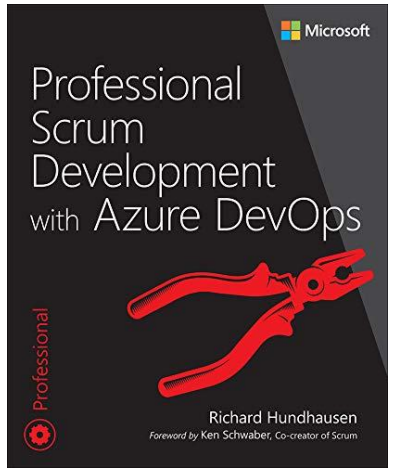
- Courseware
- Bathroom
- Fire drills

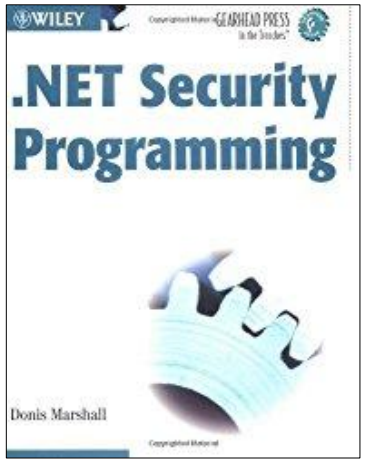
DONIS MARSHALL

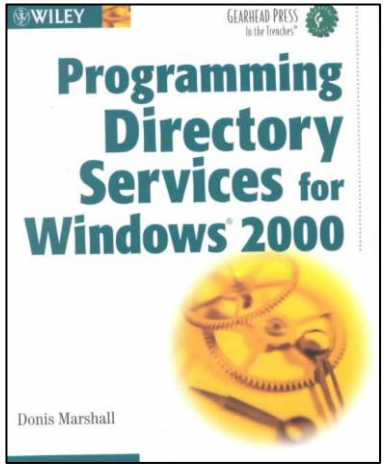
Security Professional
Microsoft MVP
Microsoft Certified Author

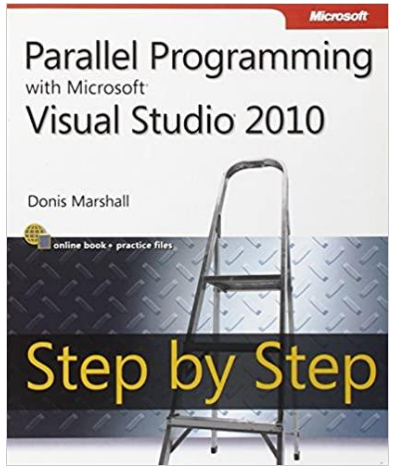
dmarshall@innovationinsoftware.com

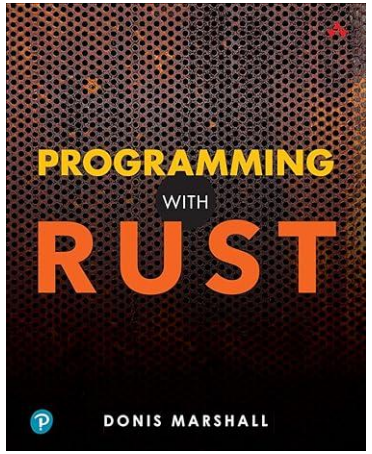












INTRODUCE YOURSELF

Time to introduce yourself:

- Name
- What is your role in the organization
- Indicate Github Copilot and Vscode experience

COPILOT

In technology, the term copilot refers to a digital assistant that augments a user's capabilities by providing real-time guidance, automation, or suggestions.

It doesn't replace human judgment but instead supplements it, offering context-aware insights and actionable recommendations.

Just as a copilot in aviation supports the pilot with navigation and decision-making, a technology copilot helps professionals carry out complex tasks more efficiently.

GITHUB COPILOT

GitHub Copilot

GitHub Copilot is an AI-powered coding assistant that integrates into development environments such as Visual Studio Code. It provides real-time code suggestions, entire functions, or even file templates as developers type.

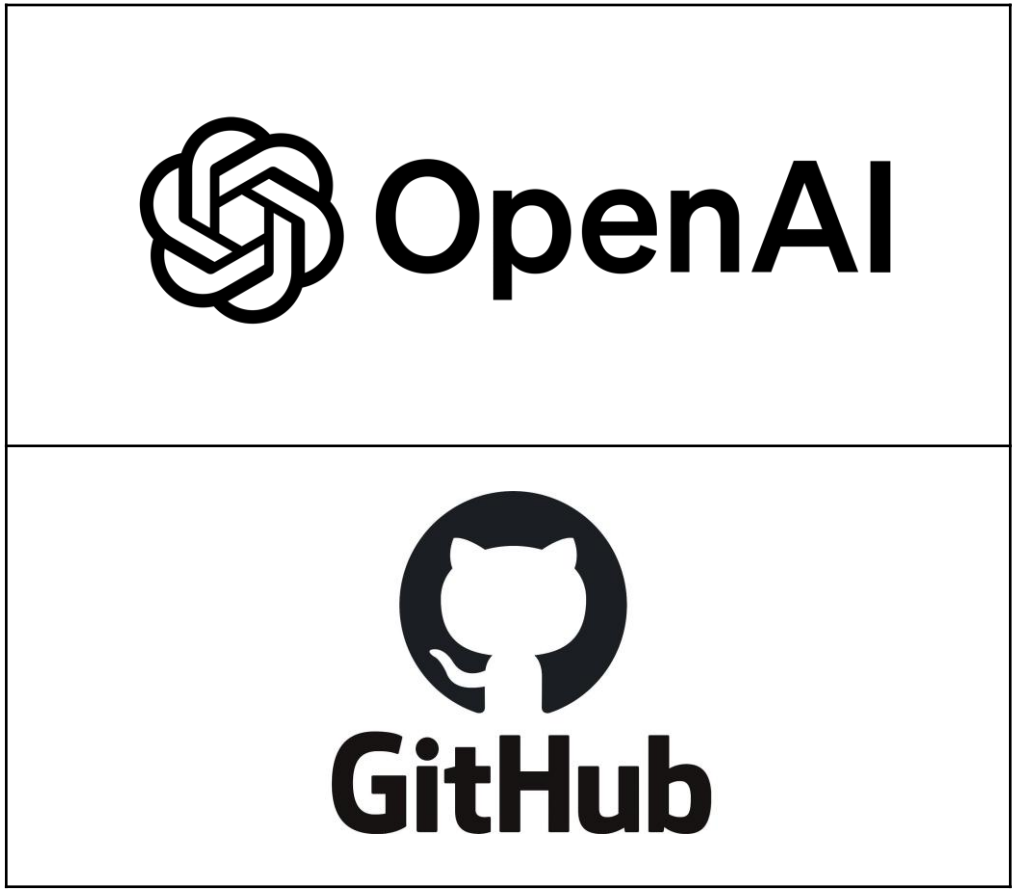
Copilot's strength lies in its ability to understand the context of the current codebase and produce recommendations that feel relevant, saving developers time and effort.

PLAIN ENGLISH

At its core, GitHub Copilot is designed to help developers write better, faster, and more consistent code. It leverages AI to understand your intent and recommend completions, functions, or entire modules that align with your project.

Its role in software development is to act as a productivity multiplier, helping developers focus on problem-solving rather than repetitive syntax or boilerplate code.

GITHUB AND OPENAI



GitHub Copilot emerged in 2021 from a collaboration between GitHub and OpenAI, combining GitHub's massive code base with OpenAI's language model expertise. Built on the Codex model, a GPT-3 descendant fine-tuned for coding, it was designed for programming tasks.

The partnership was strategic: GitHub provided billions of code examples, and OpenAI supplied generative AI research, becoming one of the first large-scale commercial uses of generative AI in software development.

GENERATIVE AI

Generative AI is a type of artificial intelligence that doesn't just analyze data, but creates new things—like text, images, music, or code—based on patterns it has learned. Instead of giving fixed answers, it generates original output that looks like it was made by a person.

Analogy: Imagine teaching a child to build with LEGO. At first, they copy the instructions you give them. But over time, after seeing many different LEGO sets, they start inventing their own creations—cars, castles, spaceships—by combining pieces in new ways. That's what generative AI does: it studies lots of examples, then uses that knowledge to "build" something new on its own.

GENERATIVE AI - 2

Generative AI in GitHub Copilot works by looking at patterns in millions of pieces of code it has learned from, then creating new code that fits what you're writing. It doesn't just copy; it predicts what might come next based on context.

Simple example: Suppose you start typing in Python:

```
def add_numbers(a, b):
```

Then pause and wait momentarily.

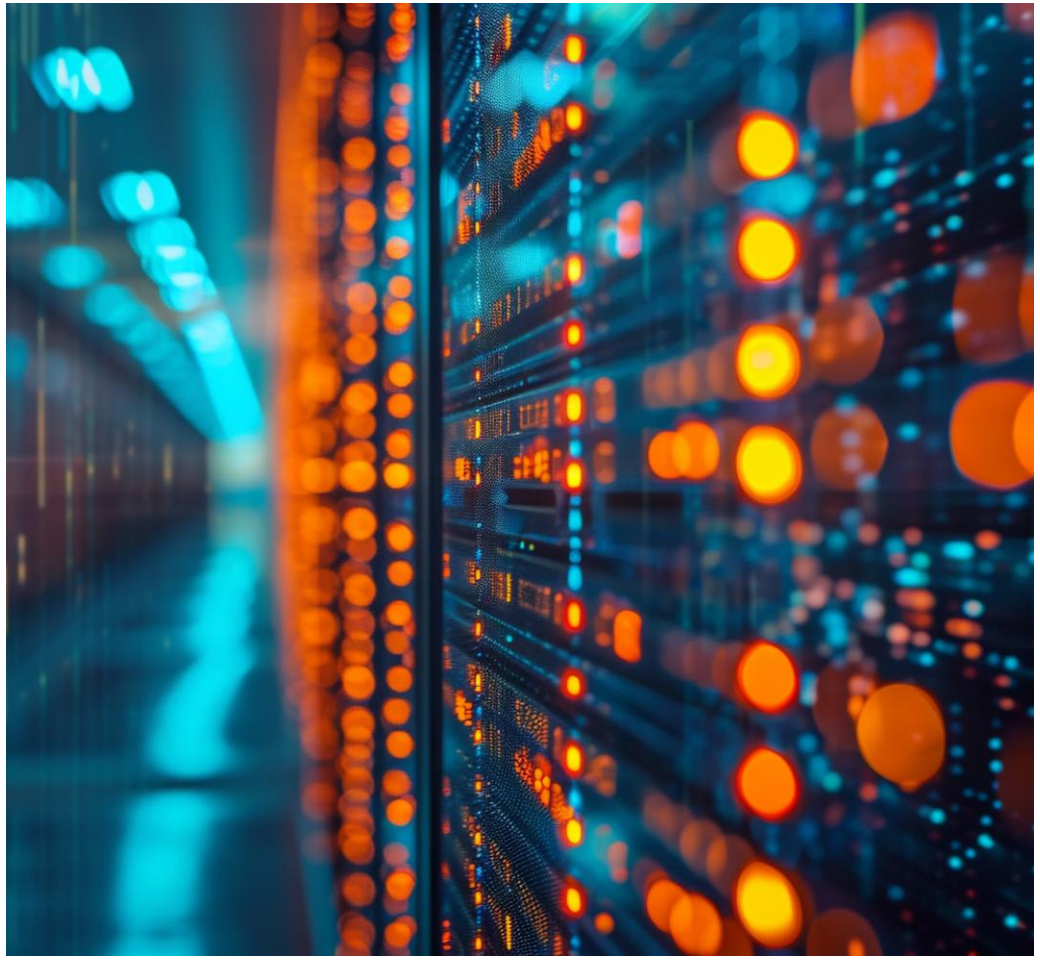
GENERATIVE AI - 3

Copilot might automatically suggest:

```
    return a + b
```

It “guessed” that since you named the function add_numbers with two inputs, you probably want to return their sum. That’s generative AI at work—it takes what you’ve written, combines it with what it has learned, and generates new code for you.

LLM



An LLM (Large Language Model) is like a giant library in the AI's brain. It has read billions of words, books, websites, and code, so it recognizes many patterns. Because it's so large, it can generate flexible, detailed answers—but it also needs more computing power. GitHub Copilot uses one or more LLMs, which is why it can suggest code in many languages and adapt to different styles.

An SLM (Small Language Model) is more like a pocket guide. It's trained on less data and has fewer "pages" in memory, so it focuses on smaller, specific tasks. It runs faster and uses less computing power, but it won't know as much or be as creative.

LLM - SOURCES

Many assume GitHub Copilot is trained solely on GitHub repositories, but the reality is broader. Sources include:

- Public GitHub repositories
- Open-source projects hosted outside GitHub
- Programming Q&A sites
- Technical documentation and manuals
- Educational resources such as tutorials and coding books

This variety ensures Copilot can generalize across multiple programming domains and coding conventions, rather than being biased toward a single platform or community.

CODEX

OpenAI's Codex, released in 2021, is the specialized large language model behind GitHub Copilot. As a descendant of GPT-3, it was fine-tuned for programming using natural language and billions of lines of publicly available code. Codex can take plain English instructions, map them to programming concepts, and generate code in many languages, effectively bridging human intent and machine-readable instructions.

GitHub Copilot, launched the same year, brings Codex into developer workflows through editors like Visual Studio Code. Acting as an AI pair programmer, it suggests lines or blocks of code, completes functions, and generates boilerplate from comments. More than autocomplete, it adapts to coding style and accelerates development. By combining Codex's AI with GitHub's massive code ecosystem, Copilot became one of the first major tools to make generative AI practical for everyday software development.

LLM – STALENESS

Copilot's training data is not continuously updated. Instead, it often lags one to two years behind current coding trends. This means it may recommend outdated functions, deprecated APIs, or older coding patterns.

To counter this, developers should cross-check suggestions with up-to-date references such as official documentation, blogs, or community discussions. Incorporating these external sources ensures that projects remain aligned with current standards.

HALLUCINATIONS

Like any generative AI, GitHub Copilot can produce hallucinations—outputs that look correct but are factually or logically wrong. This happens because the AI predicts patterns rather than “understanding” correctness.

For developers, this means Copilot suggestions must always be reviewed carefully. Blindly accepting output can introduce bugs, security risks, or inefficiencies into production systems.a

POP QUIZ: LINES OF CODE

How many lines of code can the average developer enter before creating a logical or syntactical error?

10 MINUTES

PAIR PROGRAMMING

GitHub Copilot acts as a virtual pair programmer, offering constant feedback, suggestions, and ideas without requiring another human to be physically present. This helps developers explore multiple solutions, avoid common pitfalls, and maintain coding flow.

Pair programming traditionally involves two developers working side by side, one writing code while the other reviews and suggests improvements.

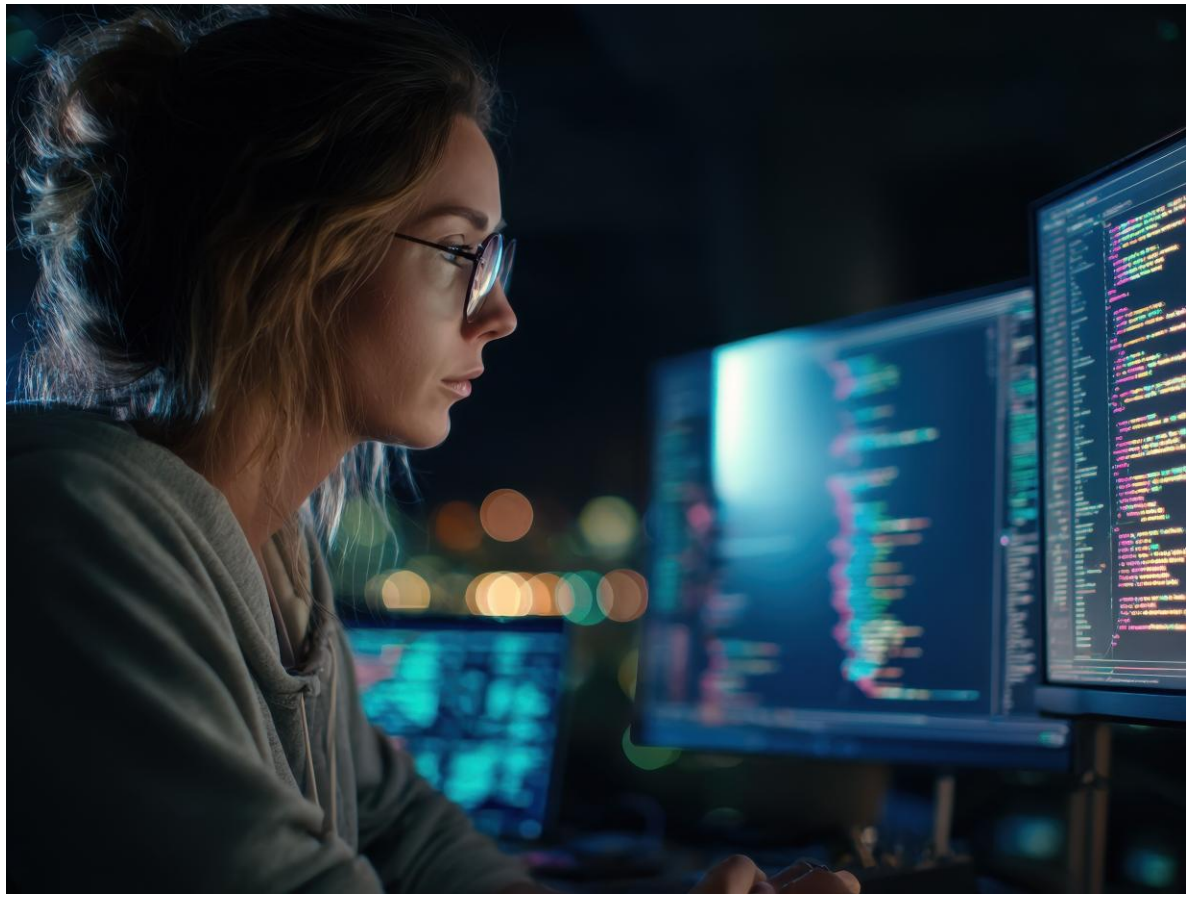
Note: Like a pair programmer, Copilot doesn't always provide the correct answer.

PAIR PROGRAMMING - 2

When using Copilot, developers must recognize their responsibility in guiding and validating output. If you consistently write poor-quality or insecure code, Copilot may mimic those patterns in its suggestions.

By following good coding standards, documenting clearly, and maintaining quality best practices, developers train Copilot to produce higher-quality suggestions in their specific projects.

CODE REVIEW



Just as pair programming requires reviewing each other's work, Copilot-generated code must be reviewed. Developers should treat AI suggestions as drafts that require human validation, testing, and refactoring.

This ensures that the final code meets project standards, security requirements, and long-term maintainability goals.

CODE PILOT CHAT

Copilot Chat extends the traditional completion model by enabling natural language conversations directly within the coding environment. Instead of just typing code and receiving suggestions, developers can ask Copilot questions like, “How do I implement OAuth2 authentication?” or “Explain what this function does.” This conversational interface makes problem-solving more intuitive and context-driven.

The importance of Copilot Chat lies in its ability to bridge the gap between coding knowledge and documentation search.

USE CASES

Boilerplate and Scaffolding

Copilot accelerates development by generating repetitive setup code such as class definitions, configuration files, or API route handlers. For instance, typing # create a Flask route for /hello can instantly produce a functioning endpoint without you writing the boilerplate manually.

Code Completion and Suggestions

It provides context-aware completions for functions, loops, and conditionals. Start typing for i in range(in Python, and Copilot may suggest the entire loop with a sensible body. This allows developers to move faster and maintain momentum.

Error Handling and Validation

Copilot generates robust error-handling blocks, helping developers avoid common pitfalls. For example, in JavaScript, adding a comment // handle fetch errors can prompt Copilot to create a try { ... } catch (error) { ... } structure.

USE CASES - 2

Learning and Language Assistance

Developers working with unfamiliar frameworks or languages can rely on Copilot to propose idiomatic patterns. Typing # connect to MongoDB may yield a correct connection snippet in Node.js, Java, or Python.

Test-Driven Development (TDD)

Copilot is excellent when writing tests first. Developers can describe intended behavior in comments or partial test code, and Copilot generates full unit tests or assertions.

USE CASES - 3

Finding Logic Errors and Reviewing Code

Copilot is not just a code generator; it can help detect subtle logic mistakes by suggesting corrections or alternative implementations. For example, if a loop doesn't terminate properly or a condition is inverted, Copilot often proposes the correct fix. It can also suggest more efficient or readable versions of code, acting like a lightweight peer review.

Source Control Assistance

Copilot integrates well with GitHub workflows, helping draft commit messages, suggesting summaries for pull requests, or scaffolding git commands in scripts.

POP QUIZ: USE CASES

What other use cases, if any, are important to mention?

10 MINUTES

FLEXIBLE / ADAPTABLE

LLM MODELS

GitHub has expanded Copilot beyond just OpenAI's GPT family. It now also supports models from Google (Gemini) and Anthropic (Claude).

LLM Family	Examples in Copilot
OpenAI	GPT-4.1, GPT-5 mini, GPT-5, o3, o4-mini
Anthropic (Claude)	Claude Sonnet 3.5, 3.7, 4, Claude Opus 4 & 4.1
Google (Gemini)	Gemini 2.0 Flash, Gemini 2.5 Pro

LANGUAGES SUPPORTED

GitHub Copilot supports a wide range of programming languages.

Language	Language
Python	JavaScript / TypeScript
Java	C#
Go	Ruby
PHP	C++
Rust	SQL
Shell scripting (Bash, PowerShell)	—

IDE SUPPORTED

Copilot currently integrates with:

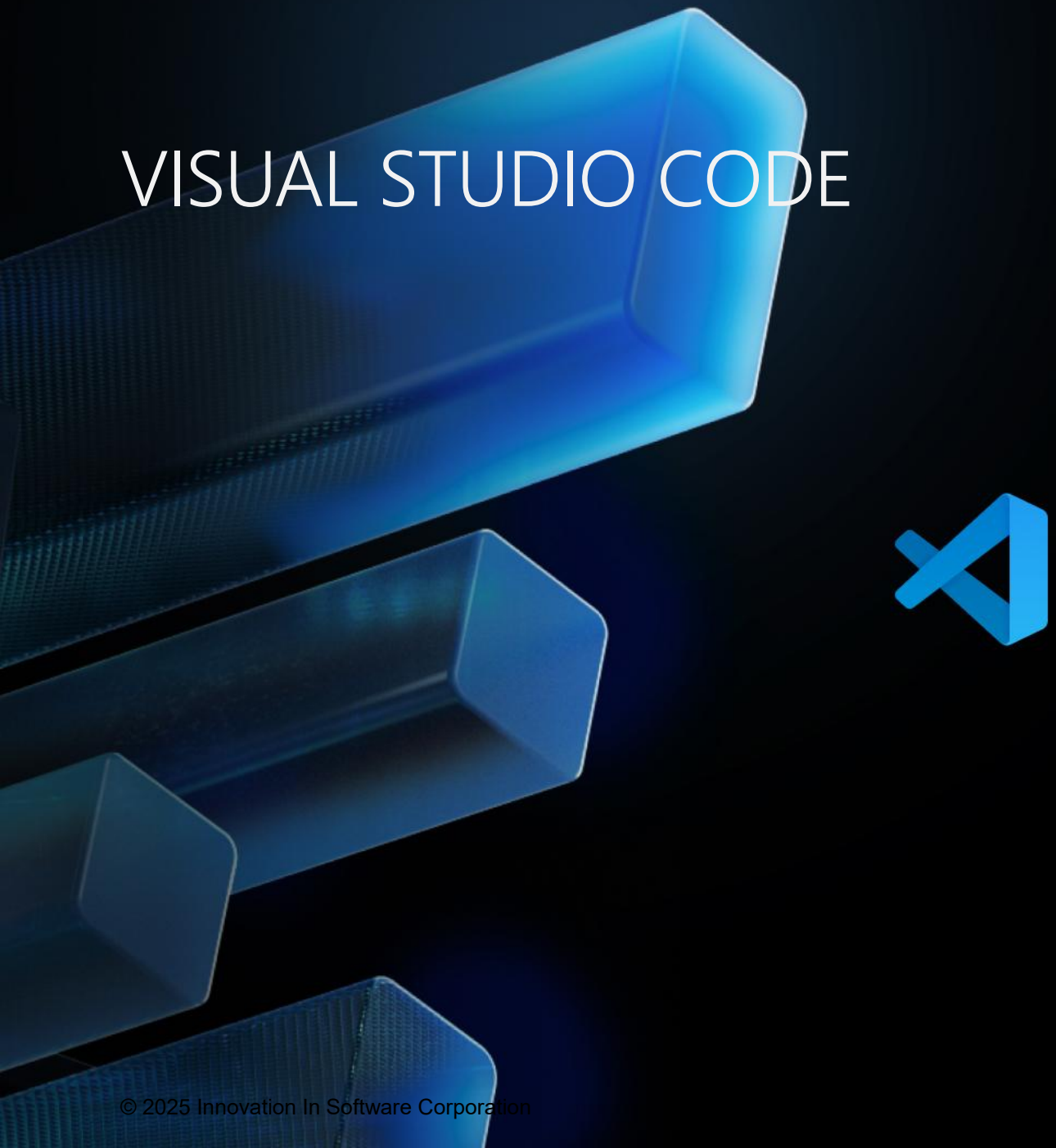
- Visual Studio Code
- Visual Studio (for .NET developers)
- Neovim
- JetBrains IDEs (IntelliJ IDEA, PyCharm, WebStorm, etc.)
- Eclipse

Note: Level of integration and support may vary. Be sure to check reference documentation.

COURSE AGENDA

1. Introduction to GitHub Copilot
2. Copilot setup and configuration
3. Basic code completion
4. Context-aware code suggestions
5. Debugging with Copilot
6. Writing functions and modular code
7. Code documentation
8. Collaboration with Copilot
9. Introduction to testing
10. Basic CI/CD concepts
11. Capstone project (optional)

VISUAL STUDIO CODE



Visual Studio Code (Vscode) is a lightweight yet powerful editor developed by Microsoft. It supports many programming languages and can be extended with thousands of extensions. Core features include IntelliSense code completion, built-in debugging, and Git integration, making it useful for .

A major strength of VS Code is its close integration with GitHub and GitHub Copilot. Developers can commit, push, and review code directly in the editor while Copilot provides AI-powered suggestions inline.

GITHUB ACCOUNT

A GitHub account is required to use GitHub Copilot.

- Subscription management: Copilot is a paid service (with free trials for students and certain accounts). GitHub needs your account to handle billing or free access eligibility.
- Authentication: You sign in with your GitHub account in your IDE (VS Code, JetBrains Visual Studio) to connect to Copilot's cloud models.
- Settings & preferences: Your GitHub account stores Copilot settings, like language/file-type controls, telemetry, and whether Copilot Chat is enabled.
- Enterprise access: For business/enterprise users, Copilot is tied to an organization's GitHub account and licensing.

GITHUB LICENSE

GitHub Copilot is a subscription service linked to your GitHub account. Plans vary for individuals, teams, and enterprises, each with different features and management controls.

Plan	Intended For	Key Features
Copilot Individual	Solo developers	Code completions, Copilot Chat, personal settings
Copilot Business	Teams & organizations	Includes Individual features + policy controls, seat management
Copilot Enterprise	Large organizations	Includes Business features + enterprise context in Chat, advanced management

LABS

Learning is better when hands-on. Some of the modules have a companion lab reinforcing a kinesthetic learning experience.

- Labs review and reinforce important concepts
- Don’t be surprised - many of the labs extend the topics introduced in the module
- The labs offer an opportunity for real-world experience
- Pair programming can be effective when working on the labs

YOUR CLASS!

Yes, this is your class. What does this mean? You determine the value.

- What is the most important ingredient of class – your participation!
- Your feedback and questions are always welcomed.
- There is no protocol in class. Speak up anytime!
- We welcome your comments during and after class. Just email dmarshall@innovationinsoftware.com.

CHATGPT

ChatGPT can be used similar to GitHub Copilot — both rely on large language models to generate context-aware code and explanations.

While Copilot is tightly integrated into the coding workflow inside IDEs like VS Code,

ChatGPT excels at broader tasks such as debugging discussions, architectural guidance, and natural language exploration of problems.

CHATGPT - 2

GitHub Copilot's training is focused on public source code, especially GitHub repositories. It's optimized to recognize patterns in real-world examples, libraries, and frameworks, making it effective at suggesting boilerplate, API usage, and common idioms.

ChatGPT takes a broader approach, trained on both programming and natural language text (books, docs, forums). This makes it stronger for explaining concepts, providing architectural reasoning, or step-by-step debugging.

CHATGPT - 3

Aspect	Copilot	ChatGPT
Integration	Built into IDEs (VS Code, JetBrains).	Web/app chat; extensions exist.
Focus	Inline code completion.	Broader Q&A and reasoning.
Context	Uses nearby code (FIM).	Uses conversation history.
Ease	Fast, automatic as you type.	Manual prompts, copy-paste code.
Strengths	Scaffolding, TDD, fixes.	Explaining, design, reviews.
Pros	Seamless coding flow.	Detailed answers, flexible.
Cons	Limited to coding tasks.	Less integrated with IDEs.

LAB 1 – PROGRAMMING

FACTORIAL

A factorial is a mathematical operation that multiplies a whole number by every positive whole number smaller than it, down to 1.

It is written with an exclamation point. For example:

5! (read “five factorial”) means:

$$5 \times 4 \times 3 \times 2 \times 1 = 120$$

3! is:

$$3! = 3 \times 2 \times 1 = 6$$

0! is defined as 1 by convention, which helps formulas work consistently.

CONFIRM PYTHON

1. Open a terminal

    On Windows: open Command Prompt (cmd) or PowerShell

    On macOS/Linux: open the built-in Terminal app.

2. Check the version

    python --version

    If that doesn't work, try: python3 --version

    You should see something like: Python 3.11.6

If you get an error such as "command not found", Python isn't installed. Download it from and install it before continuing:

    python.org/downloads

CONFIRM PYTHON EXTENSION

1. Open Visual Studio Code.
2. Go to the Extensions view
3. Click the Extensions icon on the left sidebar (it looks like four squares).

    Or press Ctrl+Shift+X (Windows/Linux) or Cmd+Shift+X (macOS).

    Search for “Python” the official extension (Python from Microsoft)

    Install the extension

FACTORIAL-LAB FILE

Create a folder for the lab

- VS Code → File → Open Folder…
- Create and select a folder (factorial-lab).
- Within the new folder, create a new file (factorial.py)

FACTORIAL FUNCTION

1. In the file, Import sys to access command-line arguments via sys.argv:

```
import sys
```

2. Define the factorial function. Then pause - the code implementation should appear. Tab to accept.

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
return result
```

MAIN FUNCTION

Create main function and after the function header add these comments “#” as scaffolding

1. Checks that exactly one argument (the number) is provided.
2. Converts that argument from a string to an integer.
3. Calls the factorial() function and print the result in the format n! = result.

Github Copilot should generate the main code. Tab to accept.

```
def main():
        if len(sys.argv) != 2:
        print("Usage: python factorial.py <number>")
        sys.exit(1)


        num = int(sys.argv[1])
        print(f"{num}! = {factorial(num)}")
```

TEST AND VALIDATE

Open Vscode terminal. Make sure factorial-lab is the current directory. Test the program:

Windows (PowerShell):

```
python factorial.py 10
```

macOS/Linux

```
python3 factorial.py 10
```

Expected:

10! = 3628800 (num) }")

Lab completed

FUNDAMENTALS

DETAILS AND WORKFLOW

INTRODUCTION

The module outlines GitHub Copilot as a cloud-based AI coding tool. It explains how prompts are enriched with context from files, functions, and project history, enabling features like fill-in-the-middle for relevant code generation.

The workflow includes creating a prompt, collecting context, secure cloud processing, and returning results as inline suggestions or chat responses.

It also highlights privacy, licensing, and security.

CLOUD-BASED GENERATIVE AI TOOL

GitHub Copilot is not just a local IDE plugin—it is a cloud-based service that relies on generative AI models hosted and maintained by GitHub and Microsoft. Every time a developer enters a prompt or writes code, that context is securely transmitted to cloud servers where the model processes it and returns suggestions.

Being cloud-based ensures scalability and continuous improvement. The models can be updated, retrained, and fine-tuned without requiring developers to manually install updates.

PROMPT

Prompts include more than what you type.

When you type a comment or partial line of code, that is only part of the prompt. Copilot automatically enriches it with additional context from the surrounding code, open files, and repository history. This means your “real prompt” is much larger than the few words you typed, leading to more accurate predictions.

This is why Copilot can sometimes anticipate what you want several lines ahead—it is interpreting not just your last keystrokes but the broader coding environment.

CONTEXT

Context is critical to how GitHub Copilot works. The tool doesn't just generate random code—it looks at the file you're writing, the function names, other files you have open, etc. This context tells Copilot what you're trying to build, so its suggestions are relevant. Without context, its output would be generic and often wrong.

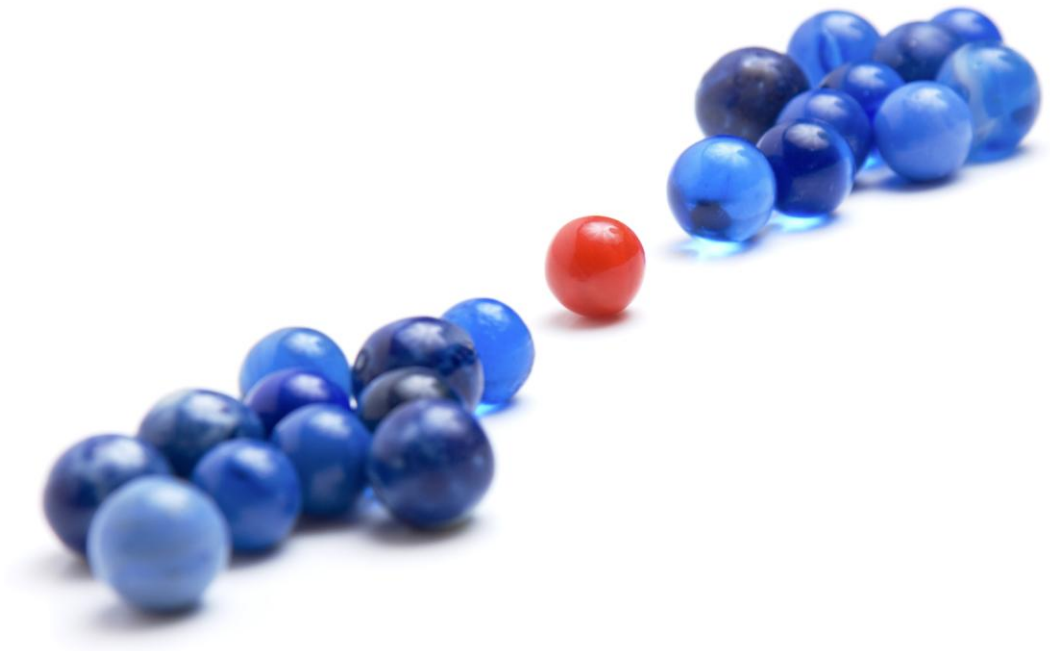
For example, if you write a comment saying # function to calculate tax, Copilot uses that clue along with your code structure to suggest a tax calculation function instead of something unrelated.

CONTEXT - 2

Copilot gathers the context from multiple sources, some more influential than others.

- Current file content
- Function and variable names
- Comments and docstrings
- Other open files in the editor
- Project-level context (local index)
- File name and file type
- Coding conventions

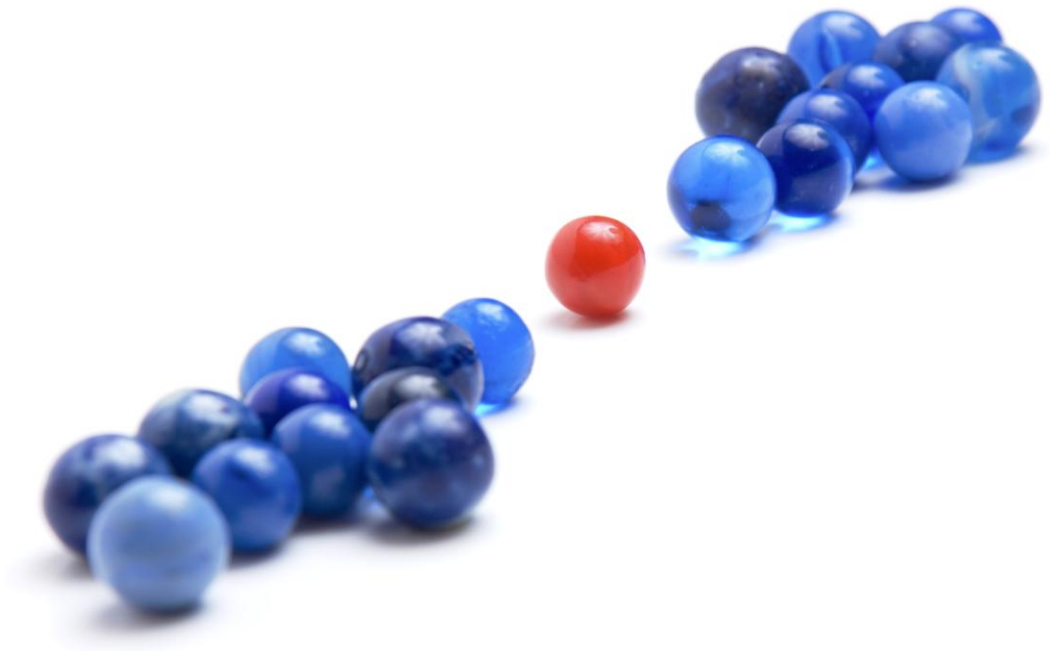
FILL-IN-THE-MIDDLE



Copilot looks at both the code before and after the cursor position to generate a relevant suggestion. This is called Fill-in-the-Middle (FIM) and means it doesn't just rely on what you've already typed—it also pays attention to the surrounding context to "fill the gap."

For example, if you have a function stub with a comment describing what it should do, Copilot can fill in the function body based on that description and any hints from code above or below.

FILL-IN-THE-MIDDLE - 2



FIM is particularly powerful for scaffolding code, refactoring, or when you're editing large files. Instead of writing code strictly top-to-bottom, you can leave placeholders or comments in the middle of a source file and let Copilot intelligently fill them in.

This encourages a workflow where you guide Copilot with intent signals (comments, partial code, or function headers), and it provides the in-between logic.

FIM EXAMPLE

Here is a FIM example:

```
def add_numbers(a, b):
        # return the sum of a and b
```

If your cursor is inside the function, Copilot may suggest:

```
    return a + b
```

Because of FIM, Copilot sees the function definition above, the comment, and the empty body, and generates the missing piece.

COPILOT WORKFLOW – STEP 1 - CREATE A PROMPT

A prompt can be:

- Code you begin typing
- A function name or variable name
- A comment describing what you want
- Even a natural language question inside Copilot Chat

Copilot uses this as the main clue for what to generate next.

STEP 2 – COPILOT COLLECTS CONTEXT

Before sending anything to the cloud, Copilot gathers context from your environment:

- The current file (especially lines near your cursor)
- Other open files in your editor
- Comments and docstrings
- The project's structure and imported libraries
- File names and file type (Python, JavaScript, HTML, etc.)

This helps Copilot understand not just what you typed, but where it fits in your project.

STEP 3 AND STEP 4

Step 3: Prompt + context sent securely

The prompt and gathered context are encrypted and sent to GitHub's servers. This ensures that your data is transmitted securely, never in plain text.

Step 4: AI model generates results

On GitHub's side, one of the supported LLMs (such as GPT-4, GPT-4o, Gemini, or Claude, depending on your settings) processes the input.

- The model generates one or more code suggestions.
- Optional post-processing may occur, such as basic vulnerability checks or formatting adjustments, before sending results back.

STEP 5 – RESULTS RETURNED TO IDE

The generated suggestion flows back into your IDE (e.g., Visual Studio Code, JetBrains). You'll see it as:

- An inline completion (gray "ghost text")
- A list of alternative suggestions
- A full explanation or snippet in Copilot Chat

You can accept, modify, or reject the suggestions as needed.

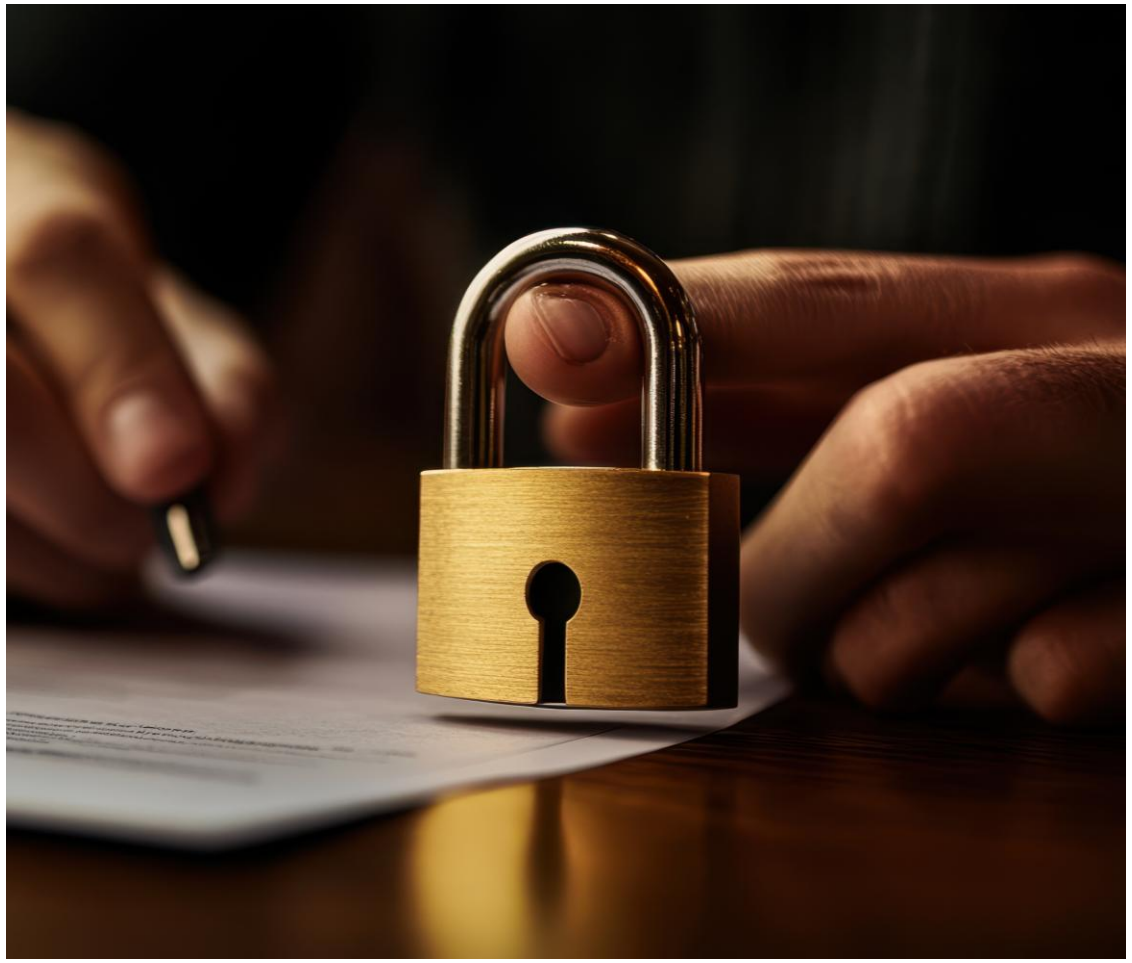
STEP 5 – RESULTS RETURNED TO IDE

The generated suggestion flows back into your IDE (e.g., Visual Studio Code, JetBrains). You'll see it as:

- An inline completion (gray "ghost text")
- A list of alternative suggestions
- A full explanation or snippet in Copilot Chat

You can accept, modify, or reject the suggestions as needed.

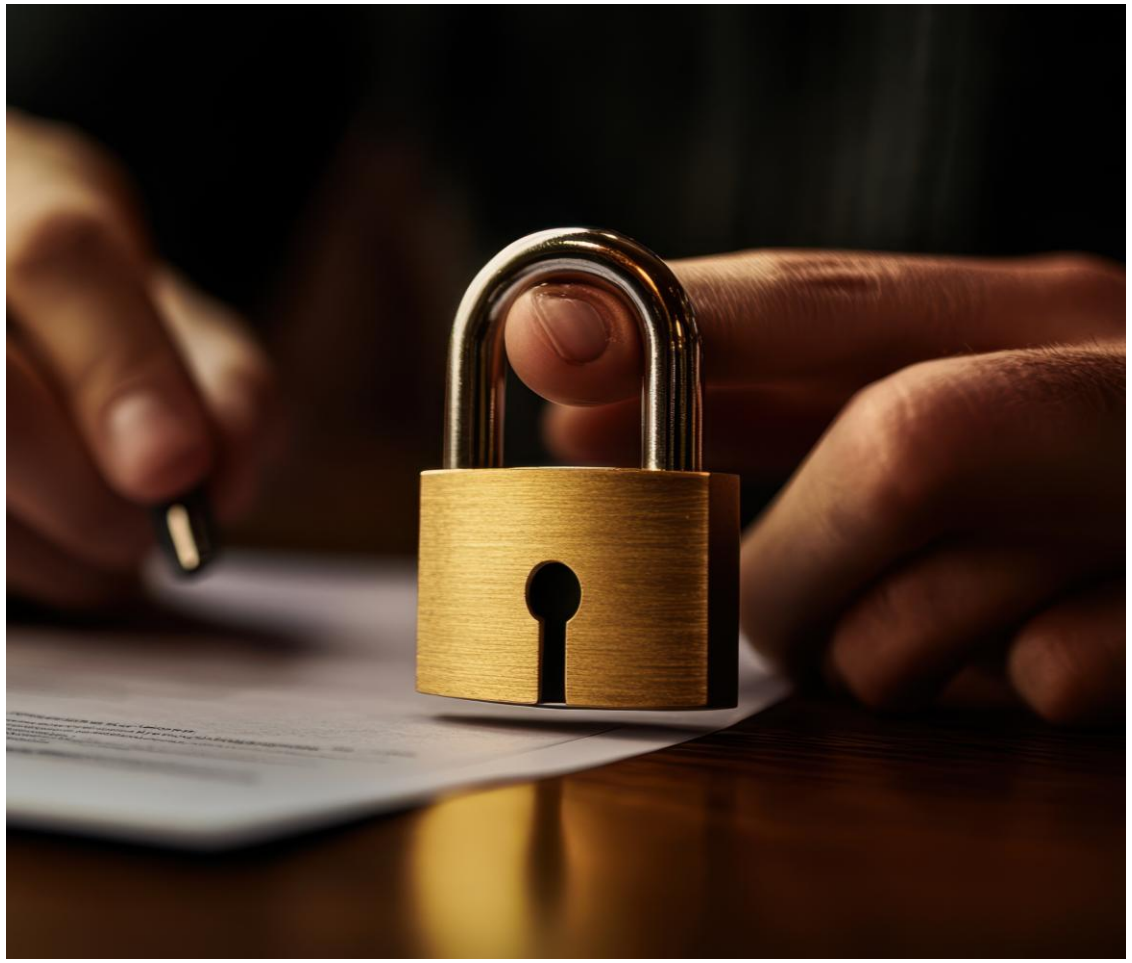
PRIVACY



Privacy is a major consideration when using AI-assisted coding tools. Copilot may collect snippets of your code to provide context, raising concerns about sensitive or proprietary data. GitHub provides an option in settings to disable code retention for training purposes, giving organizations more control over their intellectual property.

You should configure these options carefully to balance functionality with privacy and compliance requirements.

PRIVACY - 2



User engagement—such as whether you accept, modify, or reject suggestions—feeds back into the learning process for Copilot. This helps the system refine its outputs over time. However, it's important to note that feedback is aggregated and anonymized.

This engagement loop ensures that Copilot becomes more useful the more it is used.

LICENSES CONTENT

Another concern is that Copilot's training data may include code under restrictive licenses. While GitHub asserts that Copilot generates new content rather than copying directly, there is debate over whether certain outputs might inadvertently resemble copyrighted or unlicensed code.

Developers should be aware of these risks and watch for potential license compliance issues. GitHub has added features such as filtering to block suggestions that match public code verbatim.

LICENSES CONTENT - 2

- Verbatim: blocked if identical.
- Modified: same logic, different style.
- Concept-only: inspired by the idea but applied differently.

SECURITY

The prompt and code should never include sensitive data such as passwords, API keys, or confidential business logic. Even though Copilot transmits information securely, providing such details still increases the chance of exposure or accidental misuse. Developers should treat all prompts and context as if they may be seen outside the immediate coding session.

AI-generated code can also introduce security vulnerabilities, including insecure SQL queries without parameterization, weak authentication flows, or outdated cryptographic functions. To help, GitHub has added post-processing checks like vulnerability detection and filtering for unsafe code patterns.

These safeguards are limited. The primary responsibility for security remains with the development team, who must carefully review every suggestion, apply secure coding practices, and ensure that Copilot is used as a supportive tool rather than a replacement for sound engineering judgment.

SCOPE OF SUGGESTIONS

This is about where and what Copilot can suggest:

File scope:

Copilot considers the active file and lines around your cursor most heavily.

Project scope:

It can look at other open files in your editor session. Copilot can also consider project-level context, such as imports or dependencies.

SCOPE OF SUGGESTIONS - 2

Prompt scope:

Comments or function names you type strongly shape suggestions.

Example: typing # function to calculate factorial will cause a factorial implementation suggestion.

Language scope:

Copilot supports many languages, but scope is controlled — you can enable or disable suggestions by language or file type.

SETTINGS / CONFIGURATION

PRIVACY

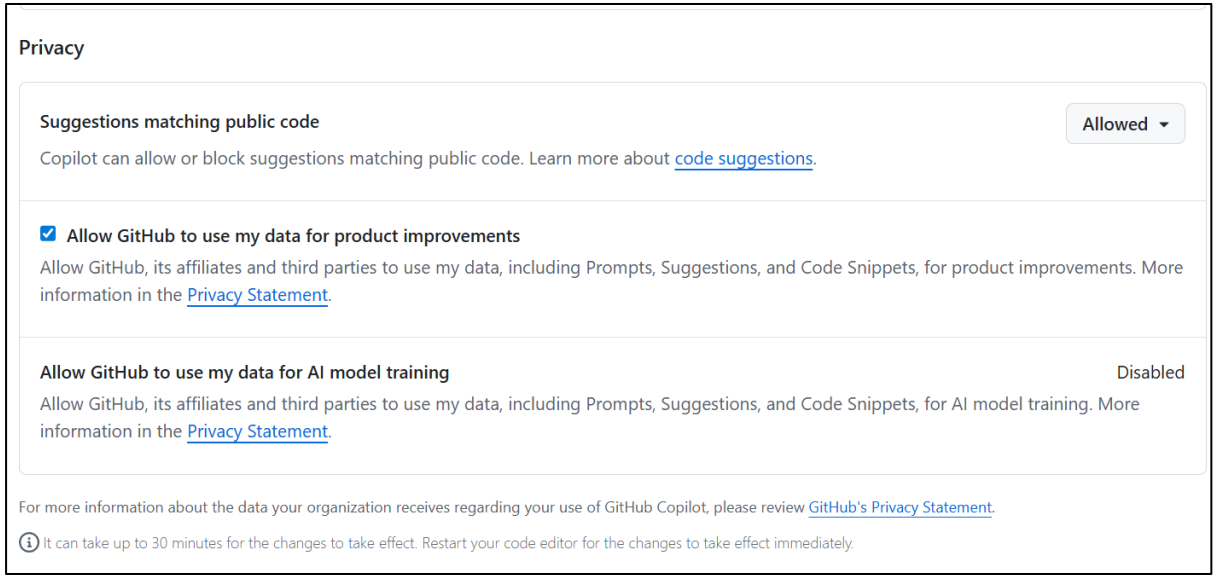
Privacy panel in GitHub Copilot settings controls how your code and interactions with Copilot are handled.

Suggestions matching public code: You can choose whether Copilot is allowed to show you completions that are very similar to existing public code on GitHub. Blocking this reduces the chance of receiving verbatim code from open repositories.

Allow GitHub to use my data for product improvements: If enabled, GitHub may use your prompts, suggestions, and snippets to analyze how Copilot is used, fix issues, and improve features.

Allow GitHub to use my data for AI model training: If enabled, your prompts and snippets could also be used to further train GitHub's AI models. With it disabled, your data won't be included in training.

PRIVACY - 2



How to get to the Privacy panel.

- Log in to GitHub
- Go to github.com and sign in.
- Open Copilot Settings
- Click your profile picture (top-right).
- Select Your Copilot.
- Scroll down right panel to find Privacy

ENABLE / DISABLE COMPLETION

The enable/disable completions toggle in GitHub Copilot controls whether inline code suggestions appear while you type. You can set it globally, per language, or for specific file types.

When completions are enabled, you get faster coding through boilerplate generation, context-aware help, and exposure to idiomatic patterns in unfamiliar languages. It feels like a pair-programmer suggesting alternatives. The downside is potential over-reliance, clutter from too many suggestions, and the risk of accepting inaccurate or irrelevant code.

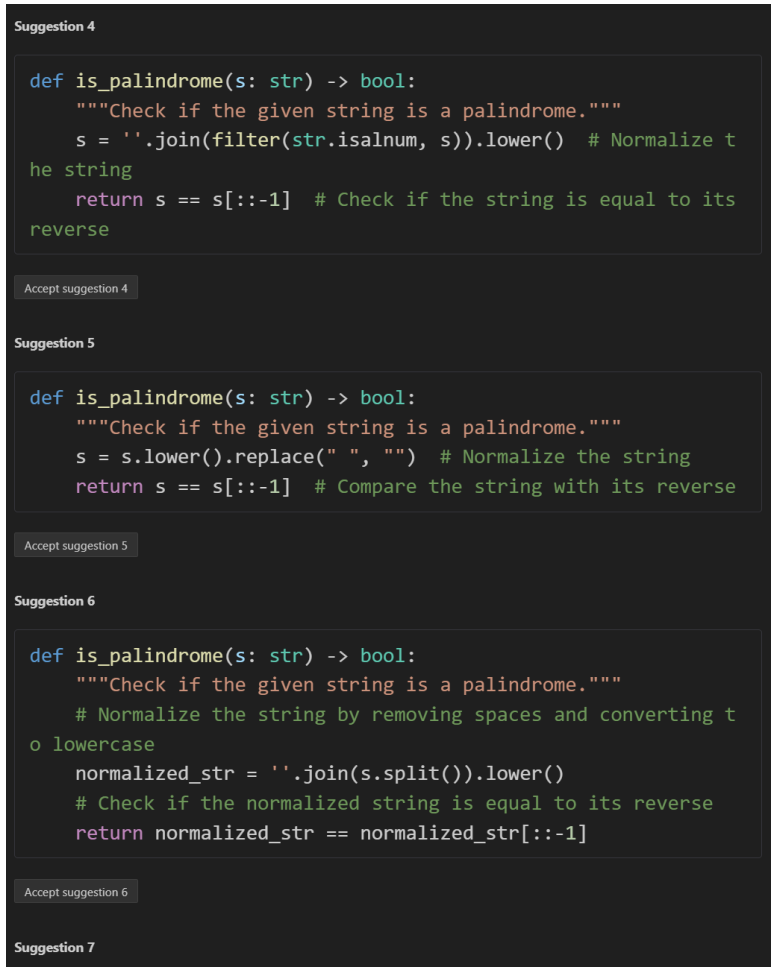
ENABLE / DISABLE COMPLETION – 2

When completions are disabled, you retain full control of your code with fewer distractions and better focus, which can be useful for teaching or training. However, you lose the productivity boost, brainstorming value, and language guidance Copilot provides.

You can still manually request suggestions. For VS Code, press Alt+\ (Windows/Linux) or Option+\ (macOS) to trigger Copilot inline suggestion.

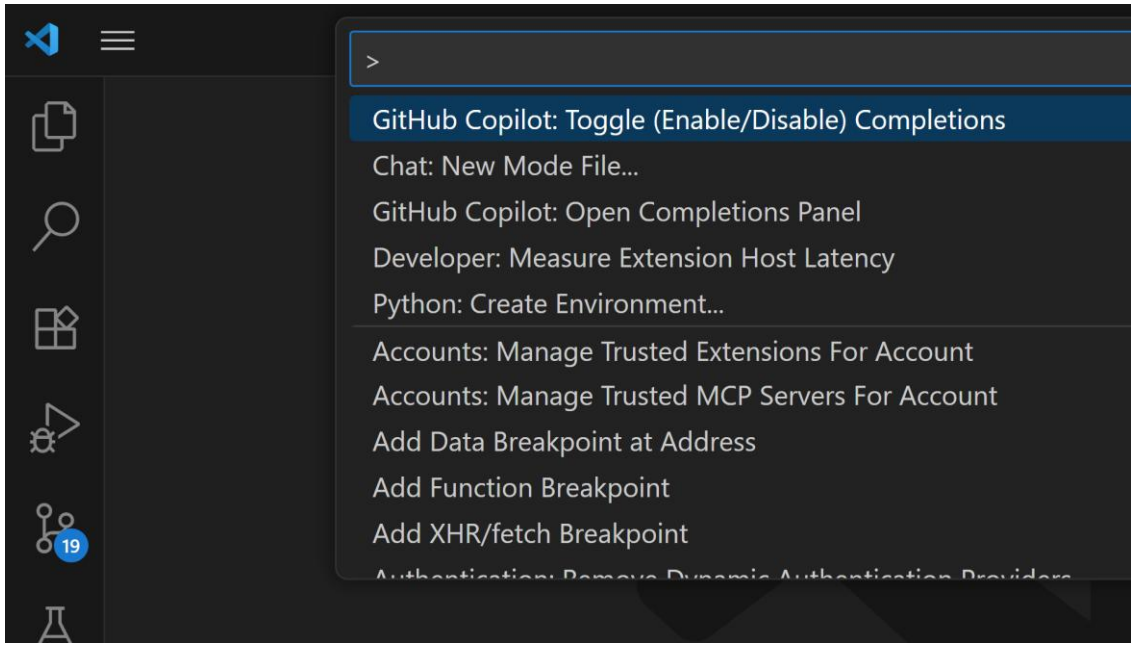
You can also use Ctrl+Shift-P and Github Copilot: Open Completions Panel (Windows/Linux) or Cmd+Enter (macOS) to open the Completions panel and see multiple alternative suggestions..

COMPLETION PANEL



Inline ghost text shows only one suggestion, while the completion panel offers several alternatives side by side. This helps you explore different idioms, like recursive, iterative, or library-based approaches. You can then accept, reject, or cycle through suggestions with the arrow keys for more control.

ENABLE / DISABLE COMPLETION – 3



Ctrl-Shift-P and GitHub Copilot: Toggle Enable / Disable Completions

DISABLE FOR A LANGUAGE

Disabling GitHub Copilot for certain languages is especially helpful when you need more control over security, focus, or code quality.

For sensitive or non-code files (such as .env, yaml, or markdown), it reduces the chance of leaking secrets, introducing misconfigurations, or cluttering documentation with unnecessary suggestions. In languages you already know well, it minimizes distractions and ensures that the code you write remains intentional. For teaching, learning, or secure environments, it helps avoid compliance risks, encourages independent problem-solving, and ensures AI isn't interfering where strict accuracy is required.

In short, turning Copilot off selectively gives you a better balance of productivity, safety, and intentional coding—leveraging AI where it's valuable

DISABLE FOR A LANGUAGE

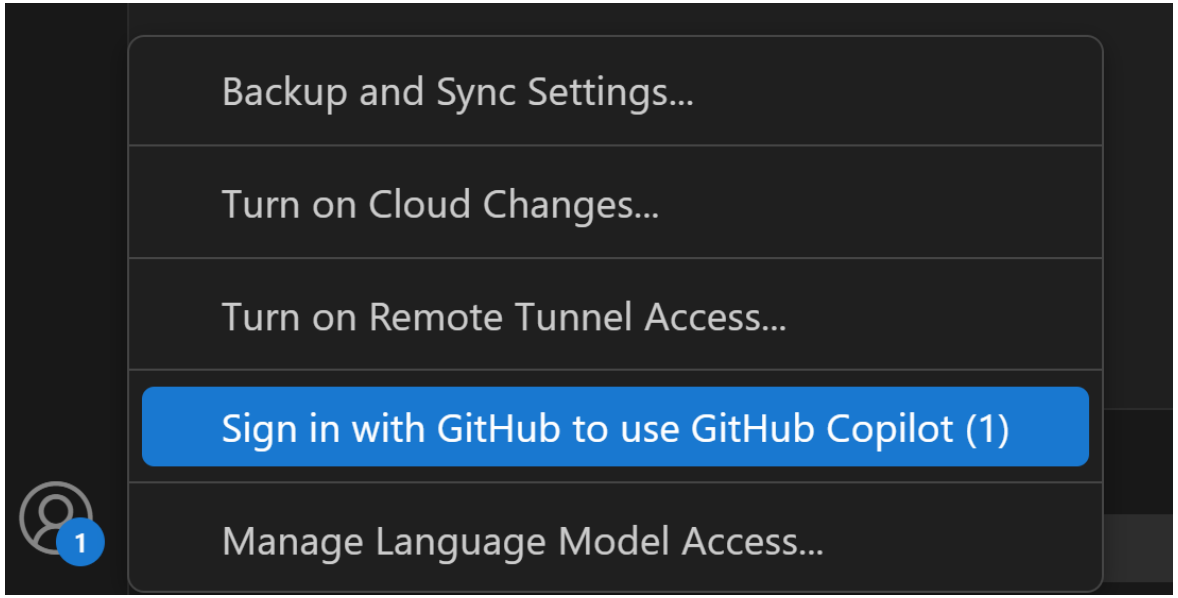
In VS Code, control this through your settings.json:

- Open Command Palette (Ctrl+Shift+P / Cmd+Shift+P).
- Type “Preferences: Open User Settings (JSON)” and select it.
- Add or adjust the github.copilot.enable section.

```
"github.copilot.enable": {
        "*": true,          // enable by default
        "plaintext": false, // disable for plain text
        "markdown": false,  // disable for markdown files
        "yaml": false       // disable for YAML files,
        "typescript": false // disable for typescript language
}
```

This ensures Copilot won’t suggest completions in sensitive or non-code files.

GITHUB ACCOUNT - LOGIN



1. In VS Code, look at the bottom left corner.
2. You'll see an Accounts icon (a little person silhouette). Click it.
3. Select Sign in with Github to use Github Copilot.
4. Alternatively, the Copilot status menu at the bottom right of the window in the status bar.

RETROSPECTIVE

This module showed how GitHub Copilot works as a cloud-based AI assistant, using context and fill-in-the-middle to generate relevant suggestions. We walked through its workflow—from prompts and context gathering to secure cloud processing and IDE results—highlighting that Copilot is more than code completion; it adapts to the developer's environment.

We also stressed responsibility and safeguards. Copilot boosts productivity, but privacy, licensing, and security require careful configuration and adhering to best practices.

LAB 2 – VSCODE

SAY HELLO

Use GitHub Copilot in VS Code to generate a Python program that displays “Hello, World”. Learn how to use comments, accept Copilot suggestions, and choose from alternatives.

SETUP

- Open Vscode

- In Vscode, login to Github using your assigned Github account for Github Copilot

- Create a new folder called:

  say-hello-world

- Create a Python File:

  say-hello-world.py

CREATE AND RUN PROGRAM

1. In the empty file, type a Python comment to display "hello, world". Make it short, concise, and direct.
2. After typing the comment, press Enter if the suggestion (ghost text) is not automatically provided.
3. Enter tab to accept the suggestion
4. Explore Alternatives (if shown)
   - Use the arrow keys to move through alternatives.
   - Press Tab to accept the one you like.
5. Run the simple program

EXTEND HELLO

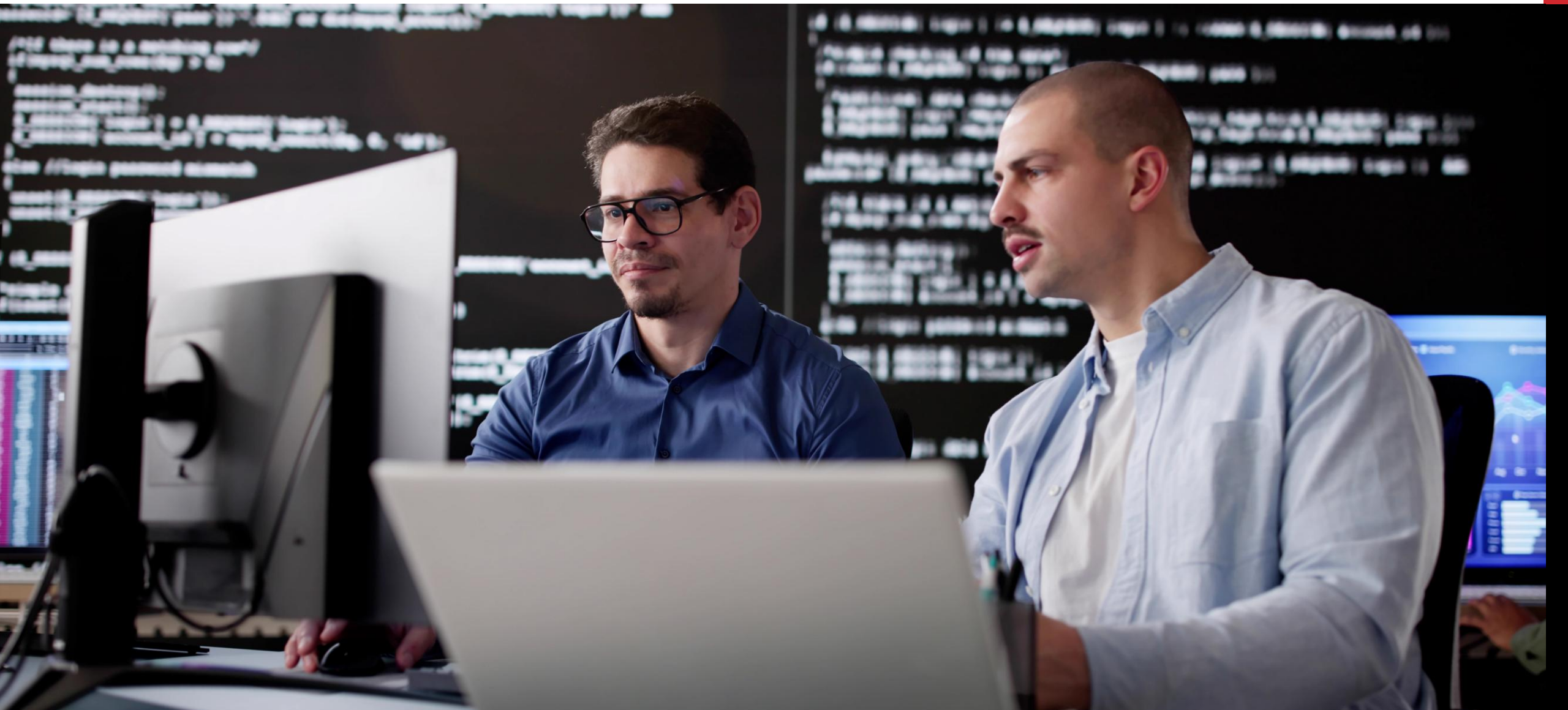
1. Delete the code
2. As multiple discrete comments, add the instructions to display hello in four different languages (your choice of language). Use natural language.
3. The comments should be generic not technical
4. Add any other comments you feel is necessary.

Run the program.

Lab completed

CODE COMPLETION

BASICS AND BEST PRACTICES

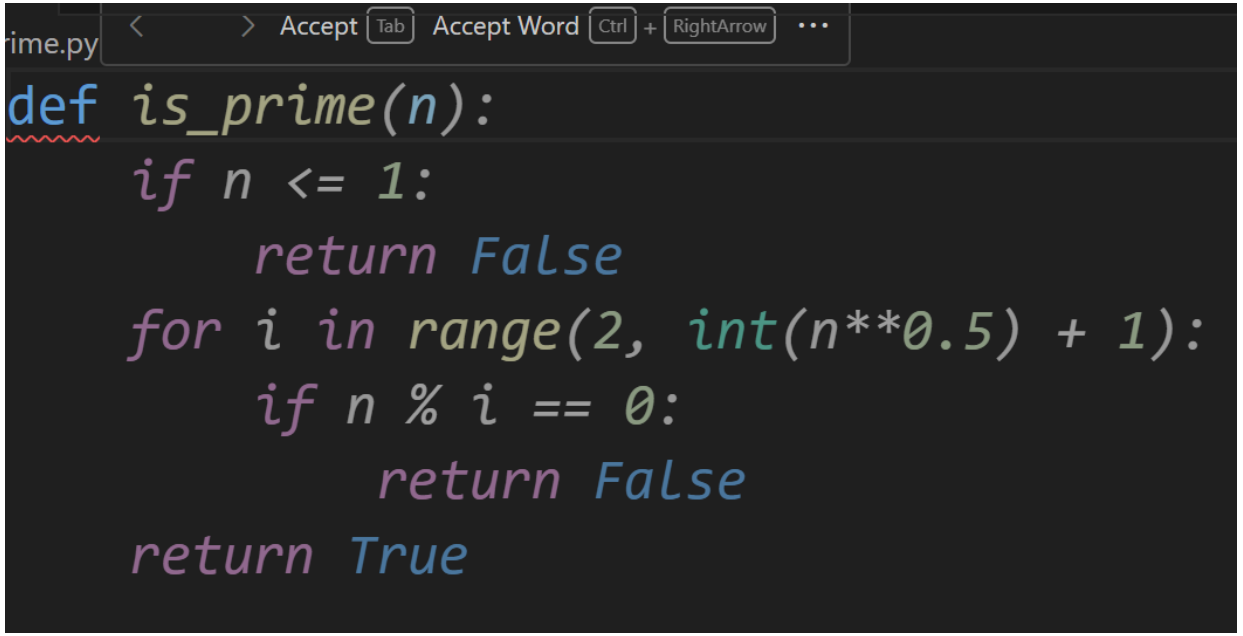


INTRODUCTION

The deck is a practical guide to using GitHub Copilot completions effectively. It explains ghost text and “getting started” habits (clear intent, pause for suggestions, accept/cycle), then shows how to sharpen results with context: meaningful names, comments/docstrings, and even adding imports to steer libraries.

It walks through scaffolding patterns (single- and multi-line), partial code guidance, and best-practices.

GHOST TEXT



Copilot predicts what you might type next, showing it as italic gray text. It appears as faint, gray, italicized text that appears inline in your editor as a code suggestion before you accept it.

- The text isn't "real code" yet — it won't run, save, or commit until you accept it.
- Accepting with a keystroke can save you multiple lines of typing.
- Safe to ignore: If you keep typing, the italic ghost text disappears automatically.
- Options: You can cycle through alternatives (next/previous suggestion) if more than one is available.

GETTING STARTED

Best practices for code completion:

- Start with a clear intent signal: type a descriptive comment or a function signature (e.g., # print "hello, world" or def add(a, b):).
- Pause after a newline—Copilot will show ghost text. Press Tab to accept, or use arrow keys to cycle alternatives (if offered).
- Prefer small, incremental prompts: write the next line or two, accept, run, repeat. Short loops of type → accept → test keep suggestions on track.
- Use meaningful names (files, functions, variables). Better names → better suggestions.

GUIDANCE

Provides context:

- Function and variable names (calculate_tax, db_client).
- Comments/docstrings that state what and how.
- Imports and existing patterns in your file.

If suggestions drift, narrow the context:

- Add a clarifying comment.
- Rename ambiguous identifiers.
- Close irrelevant files or move code into a focused file.

DOCSTRING

```
def division(a, b):
    """ This function divides two numbers.
            Before dividing check
            if the denominator is zero.
    """
    return a / b
```

In Python, docstrings (documentation strings) are special strings used to explain what a module, class, function, or method does. They serve as in-code documentation and are written inside triple quotes (""" or '''), immediately after the definition.

- A docstring is the first statement inside a function, method, class, or module.
- It's stored in the object's .__doc__ attribute.
- Unlike regular comments (#), docstrings are part of the object at runtime, so tools and developers can access them.

POP QUIZ: CONTEXT

On the previous slide, how could the context be improved?

10 MINUTES

IMPORT

This example may need an import to prevent ambiguity and the wrong choice. Without an Import:

```
def generate_random_number():
    """
    Generate a random integer between 1 and 10.
    """
```

Copilot may not know whether to use random, numpy, secrets, or another library. This could lead to different approaches: numpy, random, randint, secrets.rand, or random.randint.

COMMENTS

A second comment can help GitHub Copilot by narrowing the context and reducing ambiguity. While one comment may describe the general task, adding another can specify how it should be done, what tools or techniques to use, or what constraints to follow.

```
# Calculate factorial of a number
# Use recursion
def factorial(n):
```

SCAFFOLDING

Writing basic functions and scripts with scaffolding in comments. Wait for a moment and if not presented immediately then press Enter to invite a suggestion. For example:

```
# read a csv file and print first 5 rows
```

Guide with signatures:

```
def slugify(text: str) -> str:
```

Add a short docstring to steer edge cases (whitespace, punctuation).

MULTI-LINE SCAFFOLDING

```
# Step 1: define a function to calculate the area of a circle
# Step 2: use the formula area = pi * r^2
# Step 3: import math for pi
# Step 4: prompt user for a radius and display the result
```

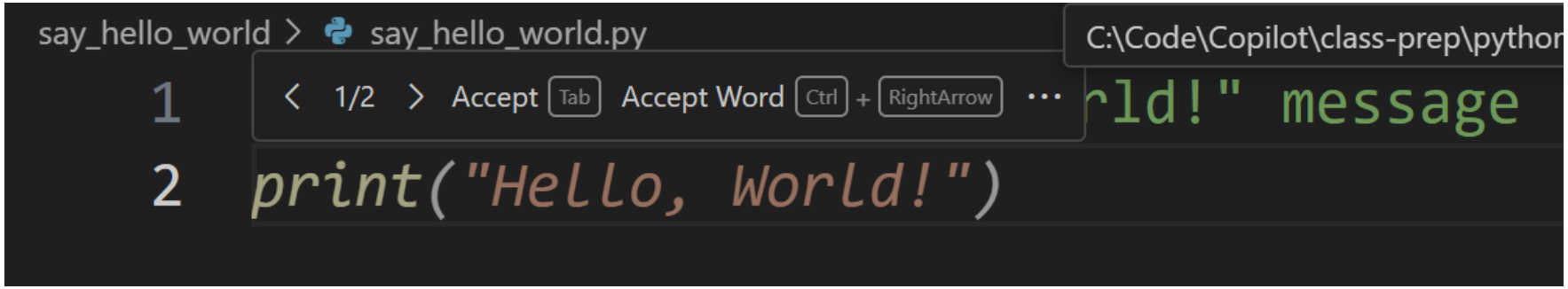
PARTIAL CODE GUIDANCE

Partial code guidance means you don't have to fully describe or write the whole solution—just give GitHub Copilot a hint through a short comment, function signature, or partial line of code. Copilot will then try to complete it based on the context.

```
# Example 1
# Return the largest number in a list
def find_max(numbers):


# Example 2
# Print numbers 1 to 5
for i in range(
```

ALTERNATIVES BAR



- Arrows (◀ ▶): to cycle through different completion sources (for example, between Copilot's inline suggestion, IntelliSense, or snippets).
- Accept (Tab): indicates you can press Tab (or Enter) to accept the current suggestion.
- Word: switches the suggestion source to word-based completions (from text in the open file).

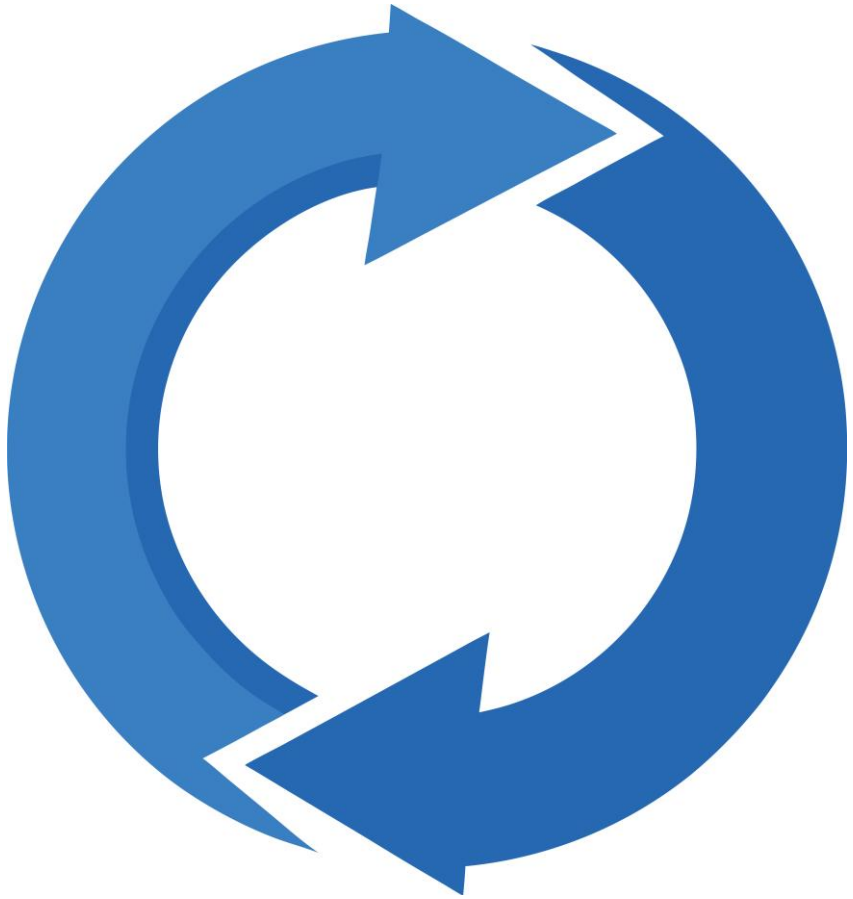
ALTERNATIVES BAR - 2

Accept Line. Let's you accept just the current line of Copilot's suggestion instead of the entire multi-line completion. This is handy when Copilot offers a large block of code.

Always Show Toolbar. Toggles whether the Alternatives Bar is always visible or only appears on **hover.**

Send Copilot Completion Feedback. Opens GitHub's feedback mechanism so you can give a quick thumbs-up/down and optionally share comments.

ITERATE



Building iteratively with Copilot (**minimum prompt** → **accept** → **run** → **refine**) gives you tighter control, fewer wrong turns, and faster feedback.

Benefits: you steer suggestions with small "intent signals," quickly test each piece, and use follow-up comments/docstrings to nudge edge cases (rounding, errors, performance) without rewriting everything.

BEST PRACTICES

Here are some best practices for Github Copilot:

- Correctness: Does it meet your intent?
- Security: Parameterize queries, validate input, avoid secrets in code.
- Style: Matches your project conventions?
- Edit actively: If it's 70% right, accept, then fix the rest—this teaches Copilot your style.

BEST PRACTICES - 2

- Request alternatives: Add/adjust comments or partially type the target line to reshape the suggestion.
- Keep prompts safe: Don’t include sensitive data (passwords, keys, proprietary algorithms).
- Validate: Run, lint, and test. Copilot accelerates typing; you own the quality.
- Feedback loop: Good names, clear structure, and small steps improve the next suggestion.

RETROSPECTIVE

This module showed practical ways to use GitHub Copilot completions effectively. We saw how ghost text works and how intent signals—comments, docstrings, imports, and function signatures—improve results. Techniques like scaffolding, partial code hints, and test-driven scaffolding emphasized that small, clear prompts generate better completions.

Reviewing output for correctness, security, and style, while avoiding sensitive data in prompts.

LAB 3 – CAGR

COMPOUND ANNUAL GROWTH RATE

Compound Annual Growth Rate (CAGR) is a way to describe growth as if it happened steadily each year, even when real results bounced around. You give it a starting value, an ending value, and how many years passed; CAGR answers, "What single yearly growth rate would get me from start to finish if it repeated every year?" It turns a jagged journey into a smooth, easy-to-compare story.

Why it's useful: it lets you compare investments or business metrics across different time periods without being distracted by big ups and downs. For example, if $1,000 becomes $1,500 in three years, CAGR says that's like growing about 14–15% per year, even if one year was great and another was weak. Just remember, it hides volatility, ignores extra deposits or withdrawals, and assumes you reinvest gains—so use it as a summary, not the whole picture.

COMPOUND ANNUAL GROWTH RATE

Step 1 – In cagr.py, provide the intent

```
# Calculate CAGR (Compound Annual Growth Rate)
def cagr(begin, end, years):
```

Copilot likely fills:

```
    return (end / begin) ** (1 / years) - 1
```

Step 2 – Refine with a second hint - make begin parameter the float type

Does anything else change

MAIN AND TEST

Step 3 – Another function by describing the intent.

# Calculate CAGR as a percentage and return a string
def cagr_percent(begin, end, years):

Step 4 – Ask copilot to create a main function to test the program with these values. Test both functions.

cagr(1000, 1500, 3)
cagr(2000, 3000, 5)
cagr(1500, 1000, 2)
. . .

Step 5 – Display the results in a nice format. Of course, using Copilot.

Step 6 – Run the program

Lab completed

COPILOT CHAT

INTEGRATION

INTRODUCTION

The presentation introduces GitHub Copilot Chat in VS Code as a conversational coding assistant. It expands beyond inline completions by supporting natural language queries, code edits, and even multi-step workflows through Ask, Edit, and Agent modes

It also highlights practical tools such as checkpoints to track and roll back edits, action controls to apply or discard suggestions, and inline chat for quick changes. Features like slash commands, #mentions for context, and variable awareness show how Copilot integrates smoothly into the coding workflow.

COPILOT CHAT

Learn how to use GitHub Copilot Chat in VS Code, contrasting the full Chat View with the lightweight Inline Chat (selection-tied, accept/discard, no history).

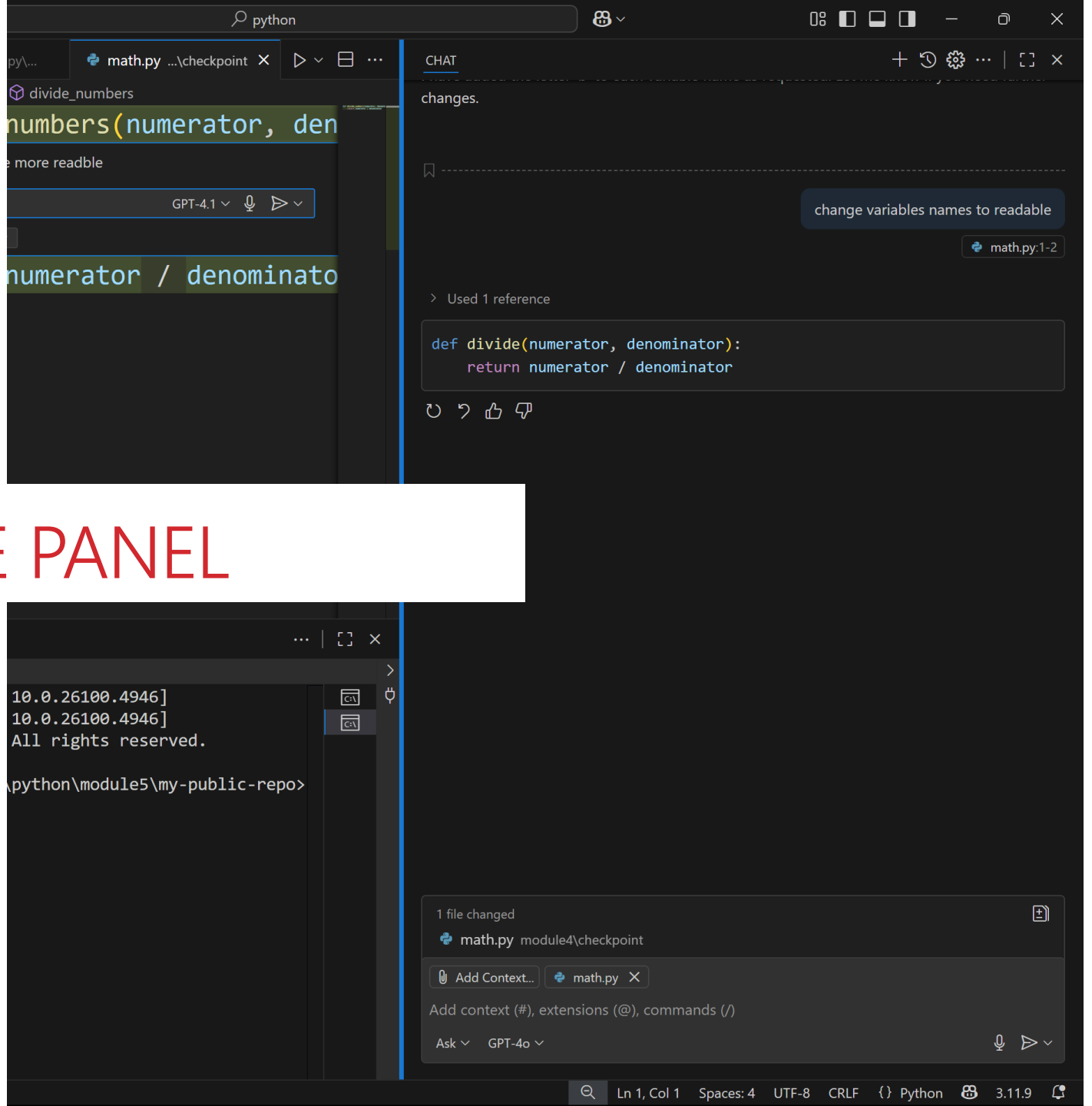
This module covers key controls—model choice, enable/disable completions, shortcuts, settings, diagnostics/logs—and two modes: Ask (Q&A) and Edit (apply changes).

HIGHLIGHTING SOURCE

When you highlight a specific block of code, Copilot knows exactly what part of the file you want it to focus on. This narrows the context and reduces ambiguity, making its output more relevant. For example, highlighting a function and asking Copilot to “optimize for readability” ensures that only that function is considered, rather than the entire file.

In addition, safer edits and clear boundaries and leads to faster review and refined control.

COPILOT CHAT SIDE PANEL



COPILOT CHAT

Open the Chat panel:

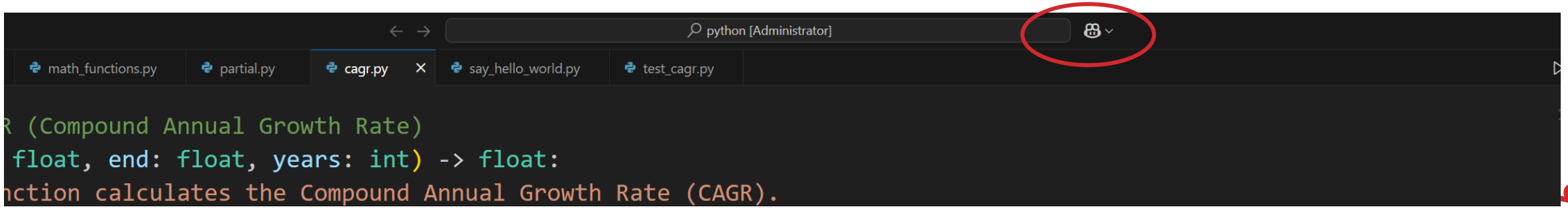
Once installed, you'll see a Copilot icon (a swirl-like symbol) in the Activity Bar on the left sidebar.

Click it to open the Chat View.

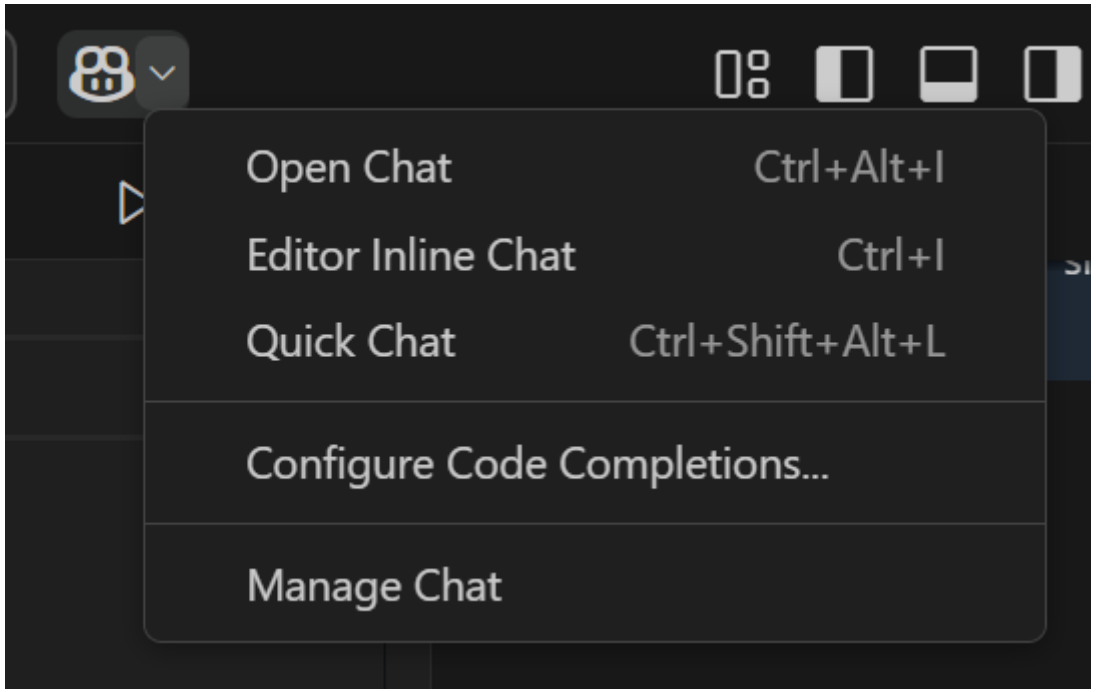
Alternatively, you can open chat directly with:

Ctrl+Alt+I (Windows/Linux)

^⌘I (macOS).



COPILOT CHAT MENU

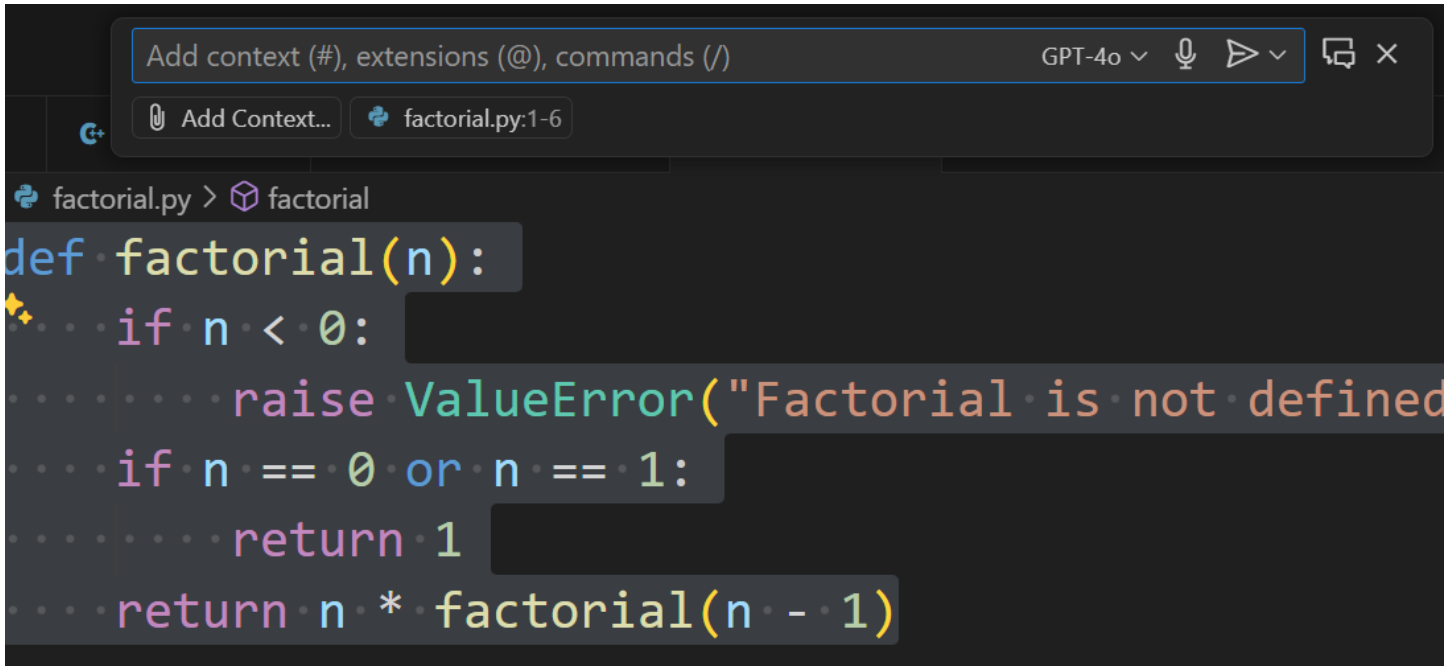


Open Chat (Chat View) is the full Copilot panel for longer conversations. You can open it from the Copilot menu or with Ctrl + Shift + I (Windows/Linux) or ^⌘I (macOS).

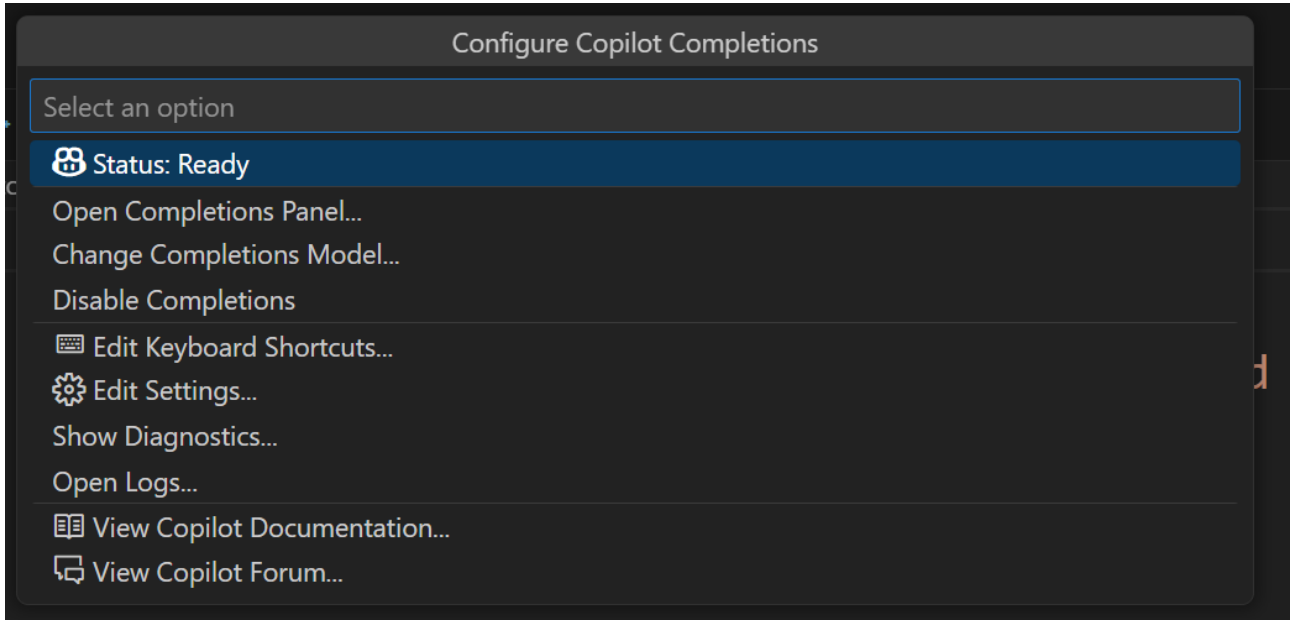
Editor Inline Chat brings Copilot directly into the editor or terminal at your cursor. Triggered by Ctrl + I (Windows/Linux) or ⌘I (macOS), it's best for quick, context-specific tasks.

QUICK CHAT

Quick Chat is a mini Chat side panel, accessed with Ctrl + Shift + Alt + L (Windows/Linux) or ⇧⌥⌘L (macOS). It opens a small box for single questions or snippets. Unlike Open Chat, it doesn't keep conversation history—ideal for quick lookups or short code suggestions.



CONFIGURE COPILOT COMPLETIONS MENU



The Configure Copilot Completion from the Copilot Chat menu is focused on how inline suggestions (ghost text) behave while you type, including the current status.

- Status – Displays whether Copilot completions are currently active. It shows if completions are enabled globally.
- Open Completions Panel – Opens a dedicated panel where you can see multiple alternative .

CONFIGURE COPILOT COMPLETIONS MENU - 2

- Disable Completions – Provides a quick way to temporarily turn off inline completions. Useful if you don't want Copilot generating ghost text while you type.
- Edit Keyboard Shortcuts – Opens the Keyboard Shortcuts editor with Copilot commands pre-filtered, so you can assign or change hotkeys for things like triggering completions or cycling through suggestions.
- Edit Settings – Opens the Copilot-related settings (in the GUI or settings.json) where you can control behavior such as enabling completions by language, automatic vs. manual triggers, and inline vs. panel display.

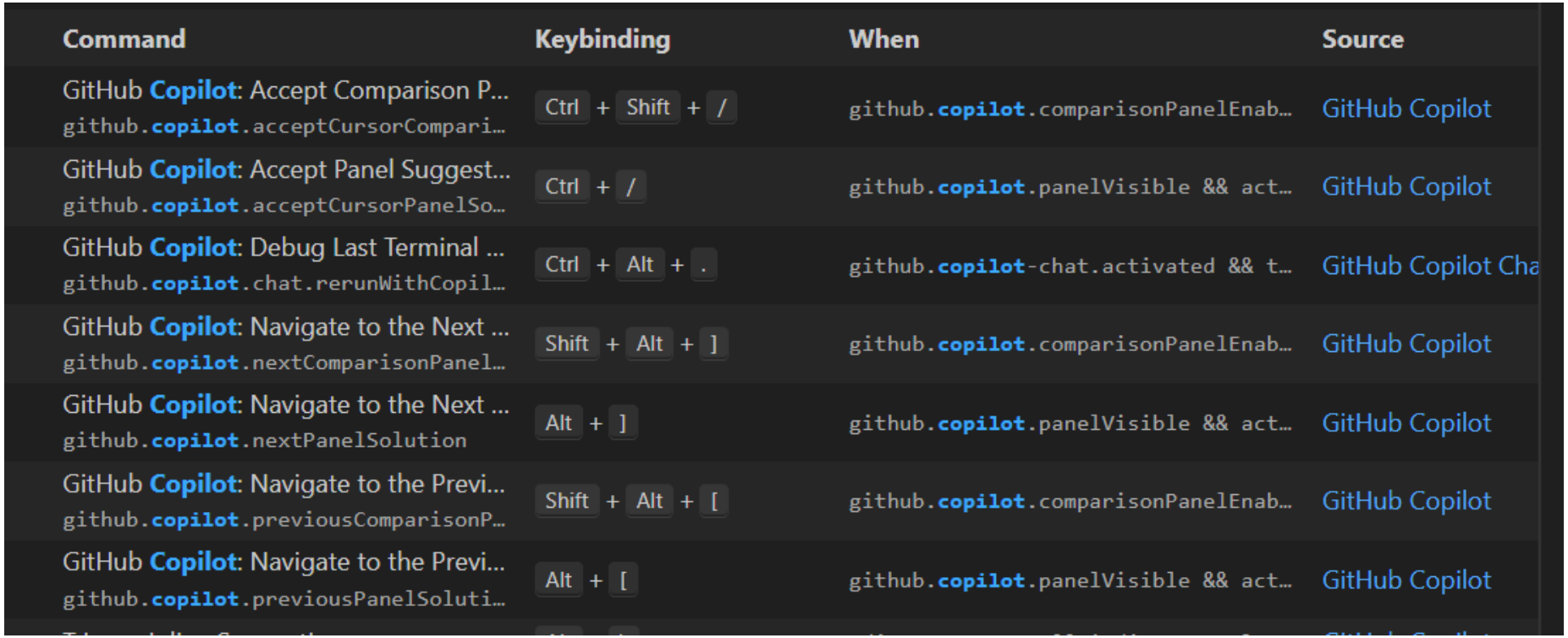
CONFIGURE COPILOT COMPLETIONS MENU - 3

- Show Diagnostics (static) – Displays diagnostic information about Copilot's state and connectivity. This can help confirm whether the extension is connected properly and if requests are being sent and received.

- Open Logs (real time) – Opens the Output panel in VS Code filtered to GitHub Copilot logs. These logs capture startup messages, network errors, or authentication issues and are helpful for troubleshooting.

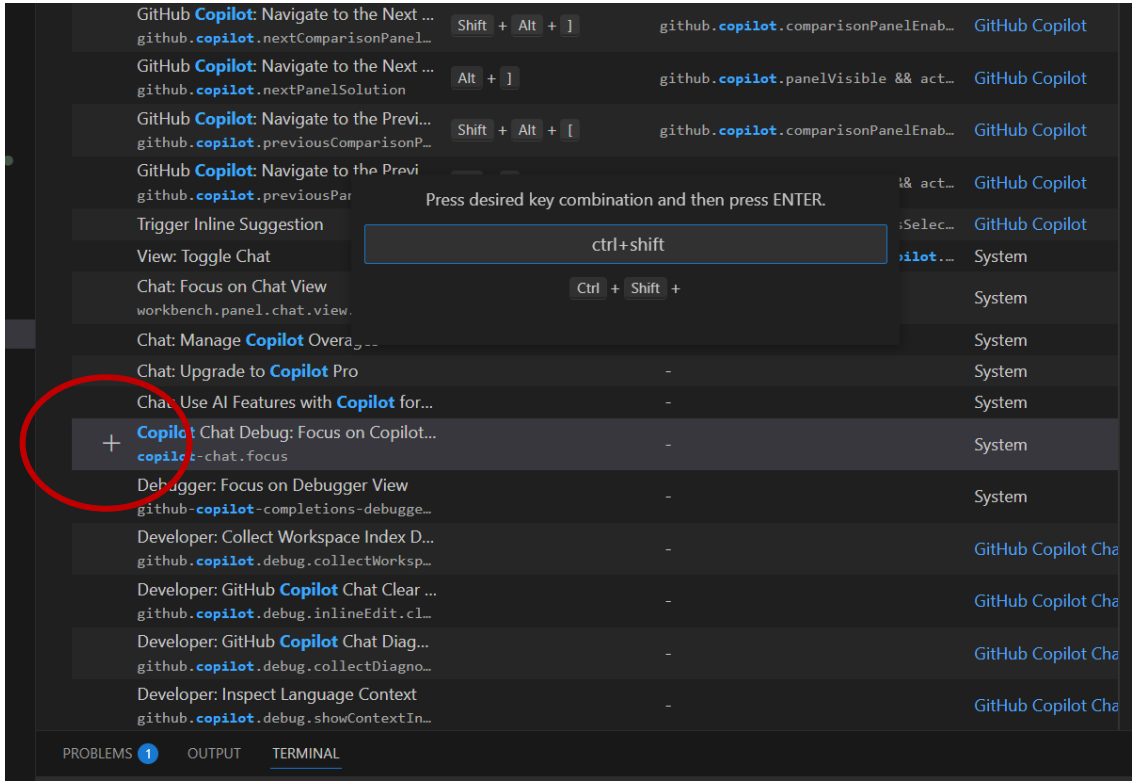
EXAMPLE - EDIT KEYBOARD SHORTCUTS

The Edit Keyboard Shortcuts option in the Copilot Chat menu opens VS Code's Keyboard Shortcuts editor with Copilot commands pre-filtered. From there, you can assign, change, or remove shortcuts



Command	Keybinding	When	Source
GitHub **Copilot**: Accept Comparison P... github.**copilot**.acceptCursorCompari...	Ctrl + Shift + /	github.**copilot**.comparisonPanelEnab...	GitHub Copilot
GitHub **Copilot**: Accept Panel Suggest... github.**copilot**.acceptCursorPanelSo...	Ctrl + /	github.**copilot**.panelVisible && act...	GitHub Copilot
GitHub **Copilot**: Debug Last Terminal ... github.**copilot**.chat.rerunWithCopil...	Ctrl + Alt + .	github.**copilot**-chat.activated && t...	GitHub Copilot Cha
GitHub **Copilot**: Navigate to the Next ... github.**copilot**.nextComparisonPanel...	Shift + Alt +]	github.**copilot**.comparisonPanelEnab...	GitHub Copilot
GitHub **Copilot**: Navigate to the Next ... github.**copilot**.nextPanelSolution	Alt +]	github.**copilot**.panelVisible && act...	GitHub Copilot
GitHub **Copilot**: Navigate to the Previ... github.**copilot**.previousComparisonP...	Shift + Alt + [	github.**copilot**.comparisonPanelEnab...	GitHub Copilot
GitHub **Copilot**: Navigate to the Previ... github.**copilot**.previousPanelSoluti...	Alt + [	github.**copilot**.panelVisible && act...	GitHub Copilot

COMMON CUSTOM KEY BINDINGS



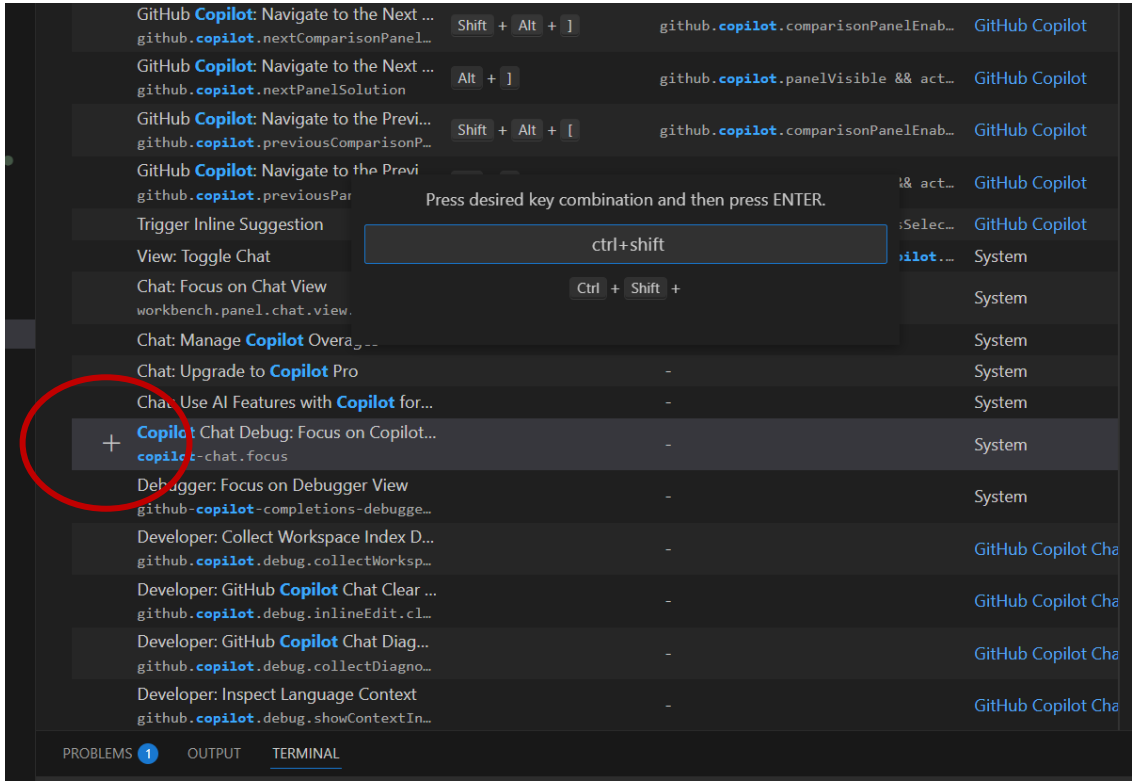
GitHub Copilot: Enable/Disable Completions – Toggle inline suggestions globally.

GitHub Copilot: Explain with Copilot – Ask Copilot to explain highlighted code.

GitHub Copilot: Accept Suggestion – Accept the current inline completion.

GitHub Copilot: Trigger Inline Suggestion – Manually request a suggestion on demand.

COMMON CUSTOM KEY BINDINGS - 2



GitHub Copilot: Generate Tests – Ask Copilot to generate unit tests for selected code.

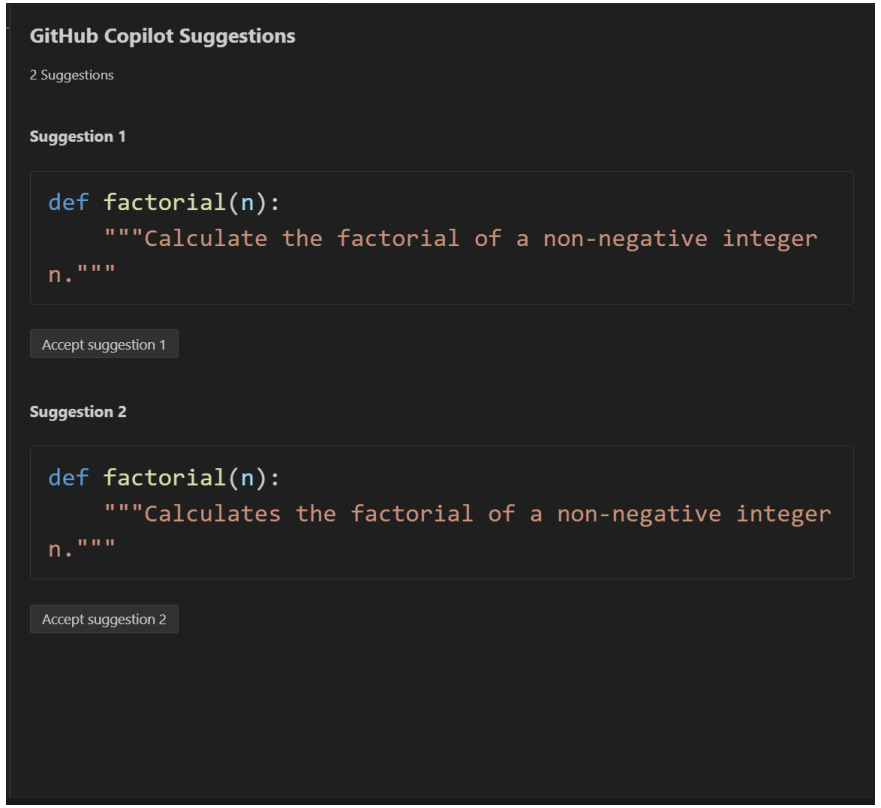
GitHub Copilot: Insert at Cursor / Apply to Selection – Choose how Copilot applies generated code.

EXAMPLE - OPEN LOG

The Copilot log in VS Code is a diagnostic record of what the extension is doing behind the scenes. It captures events like when Copilot starts up, sends or receives a completion request, or encounters an error. You can view it by opening the Output panel and selecting GitHub Copilot from the dropdown list.

These logs are mainly for troubleshooting. If Copilot isn't generating suggestions or fails to connect, the log can reveal issues such as authentication errors, network timeouts, or extension crashes.

EXAMPLE - OPEN COMPLETIONS PANEL…



Select the header for the Factorial function and choose the Open Completions Panel.

CHAT MODES

Vscode supports multiple built-in chat modes tailored to different tasks. Here are the main modes:

Ask Mode (Default)

Ideal for asking questions or explanations about code, technology, best practices, or debugging.

Responses may include code snippets or concepts; you apply changes manually

Edit Mode

Suited for applying code changes across multiple files.

You supply context, and Copilot will generate edits. You can review and accept changes

Agent Mode

For autonomous, multi-step workflows. Copilot determines necessary tasks, executes commands (e.g., build, tests), applies and iterates until completion

ASK MODE

In GitHub Copilot Chat, the default interaction mode is Ask mode.

That means when you open Copilot Chat in VS Code and type a question or prompt, you're by default in a conversational Q&A flow — asking Copilot about code, documentation, debugging, or general programming topics. Other modes, like /explain, /tests, or /fix, are invoked explicitly with slash commands, but Ask mode doesn't require a prefix — you just type naturally.

EDIT MODE

In GitHub Copilot Chat, Edit mode lets you directly modify code in your editor using natural language instructions. Instead of just asking Copilot questions (Ask mode), you highlight a block of code, then tell Copilot what you want changed. Copilot will generate a replacement or modification, and you can preview and accept it.

This mode uses NLP (natural language processing) to interpret your intent. For example, you might say “optimize this function for readability” or “add error handling for division by zero,” and Copilot edits the selected code accordingly.

Syntax: /edit *chat instruction*

ASK VERSUS EDIT MODE

In the current version of GitHub Copilot Chat, there is little distinction between Ask mode and the /edit command. When you type an instruction in Ask mode, Copilot automatically interprets your intent based on context. If you highlight code, it treats your request as an edit; if no code is selected, it generates explanations, examples, or new snippets. This makes Ask mode flexible and able to handle both conversational queries and direct code edits without requiring a special command.

The /edit command still exists, but its function has been streamlined into the same workflow as Ask mode. Using /edit explicitly signals that you want an edit, yet the outcome is effectively identical to giving an instruction in Ask mode. Both methods produce inline previews with options to accept or discard, making the experience consistent and simpler. In practice, you can rely on Ask mode alone for most tasks, since it now covers the same ground as /edit.

CHECKPOINTS

In the Copilot Chat side panel, when you use the /add or /edit commands, the history of your interactions is preserved right in the chat thread. Each request and Copilot's corresponding suggestion are shown in sequence, making it easy to scroll back and review prior edits or additions without losing context. This gives you a running log of how your code evolved through chat-based instructions.

At the same time, checkpoints are placed directly in your code editor at small dotted line markers. These indicate where Copilot made a change. If you hover over a dotted line, you'll see the checkpoint, which allows you to quickly roll back to the earlier version of your code. This provides a convenient safety net, ensuring that edits suggested through the chat panel are reversible.

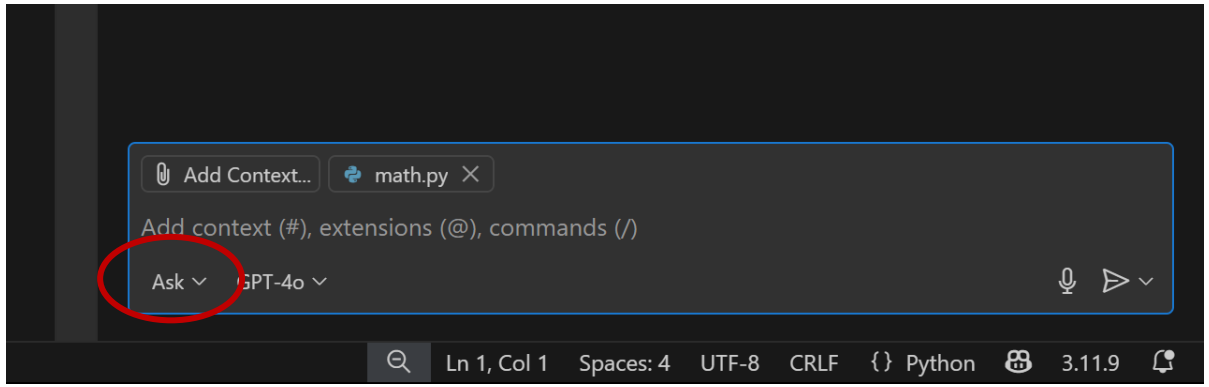
COMMANDS

You can use the slash commands to steer Copilot into specific behaviors.

Common Copilot Chat Commands

- **/ask: Default mode. Ask a question about code, concepts, or documentation.**
- **/edit: Edit selected code directly with your instruction.**
- /fix: Suggest fixes for errors or issues in the selected code.
- /explain: Explain what the selected code does in plain language.
- /tests: Generate unit tests for the selected code.

ASK MODE



At the bottom of the chat window, there is the command pane, where commands are entered. You can change the mode here also.

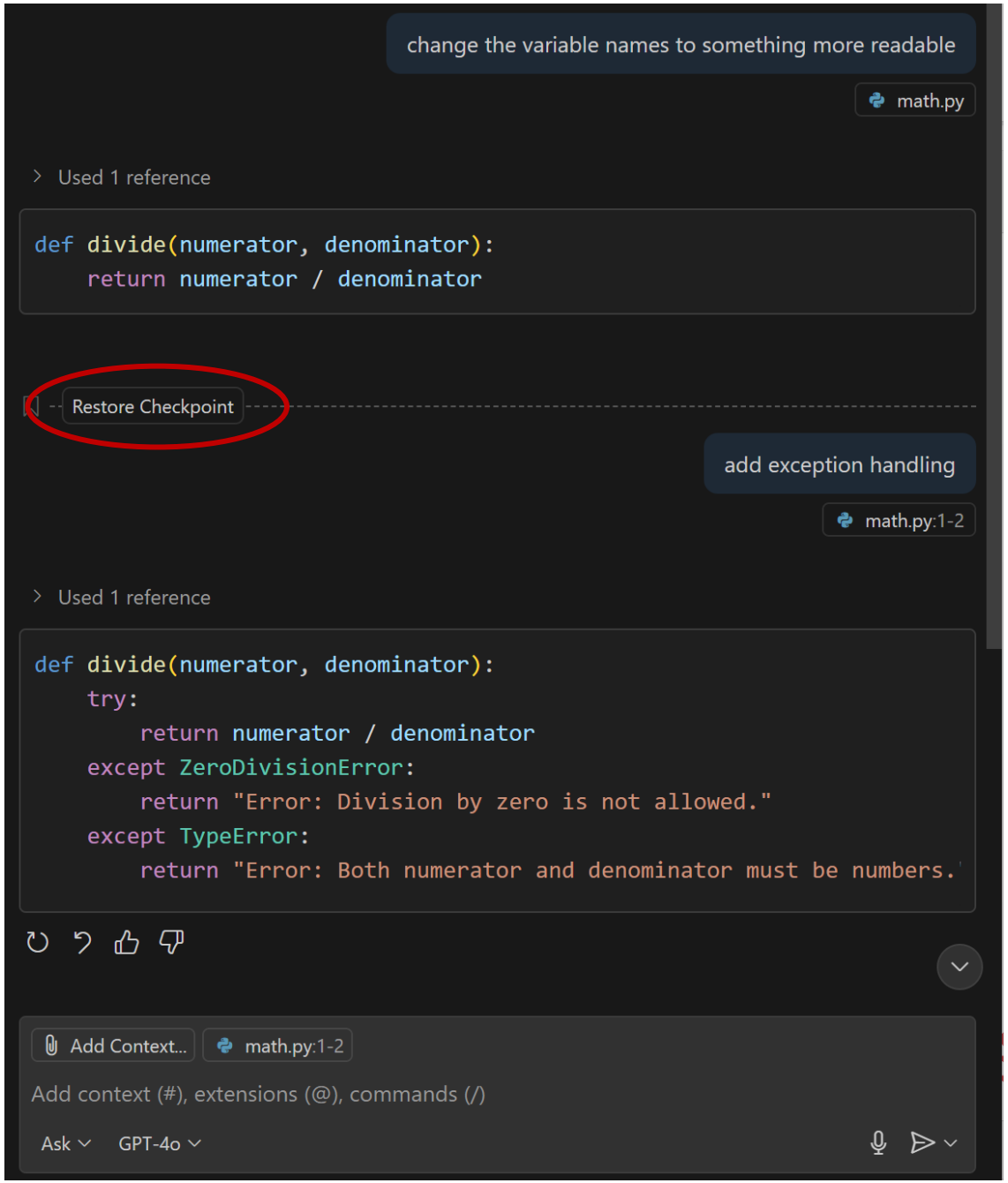
- Ask
- Edit
- Agent (discussed later)

As shown, the default is Ask, which is more advisory or view mode.

CHECKPOINTS

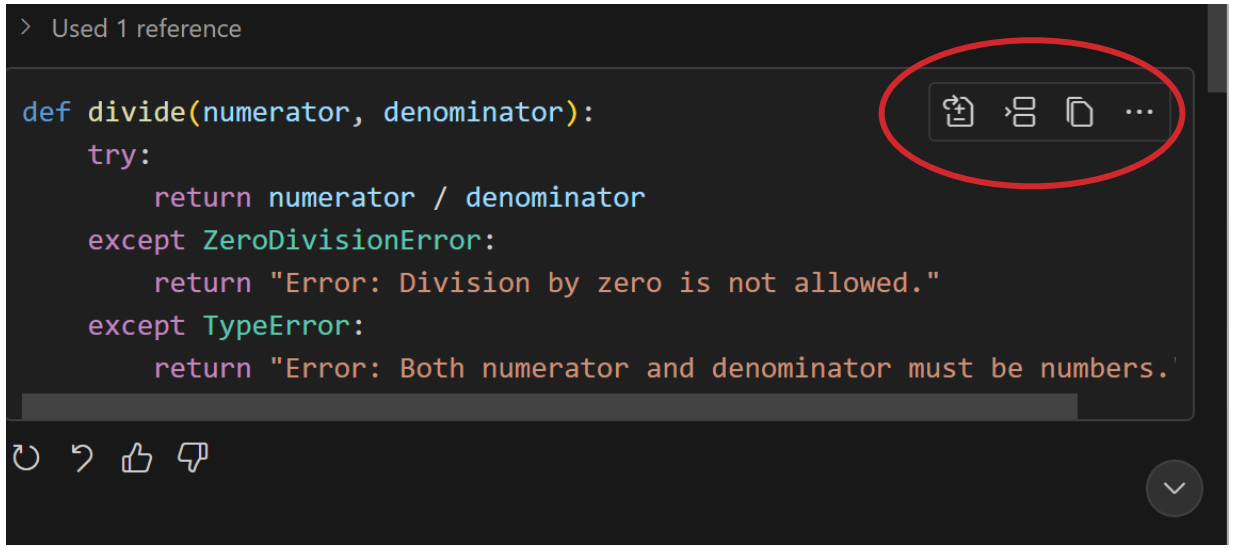
Here we have applied to Add commands to a divide function. Both are shown in the history.

The checkpoints are also displayed as dotted lines. However, over the lower checkpoints enables the Restore Checkpoint button.



ACTION BAR

When you run an /add or /edit command in Copilot Chat, Copilot's suggestion appears in the chat side panel. Hovering over that suggestion reveals four action buttons that let you decide how to use the generated code.



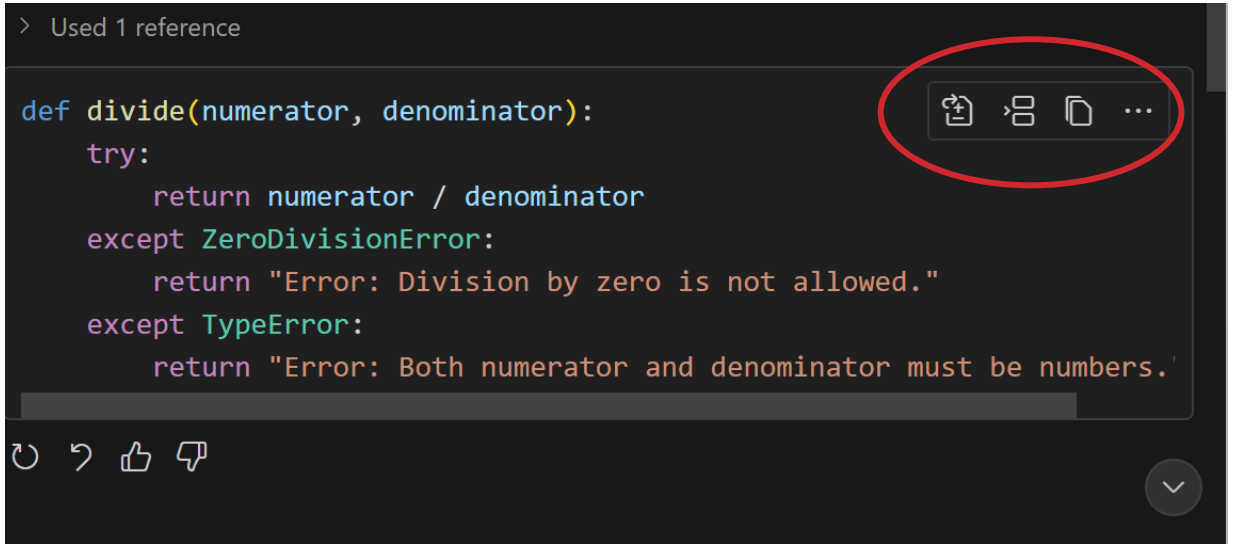
ACTIONS

Apply To: Applies the suggestion directly to the code you had selected in the editor.

Insert At: Lets you insert the suggestion at your current cursor location in the open file.

Copy: Copies the suggestion to your clipboard.

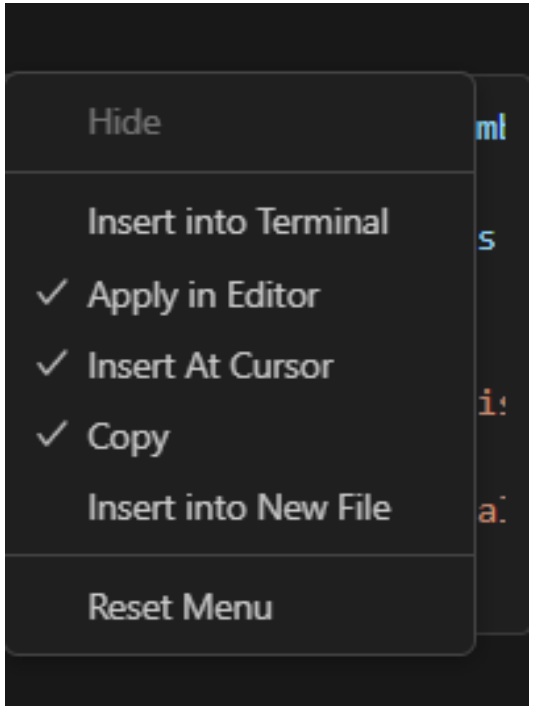
Short menu: Insert into Terminal and Insert into New File.



ACTIONS - 2

Insert into Terminal: add display to terminal window.

Insert into New File: create a new untitled file and insert

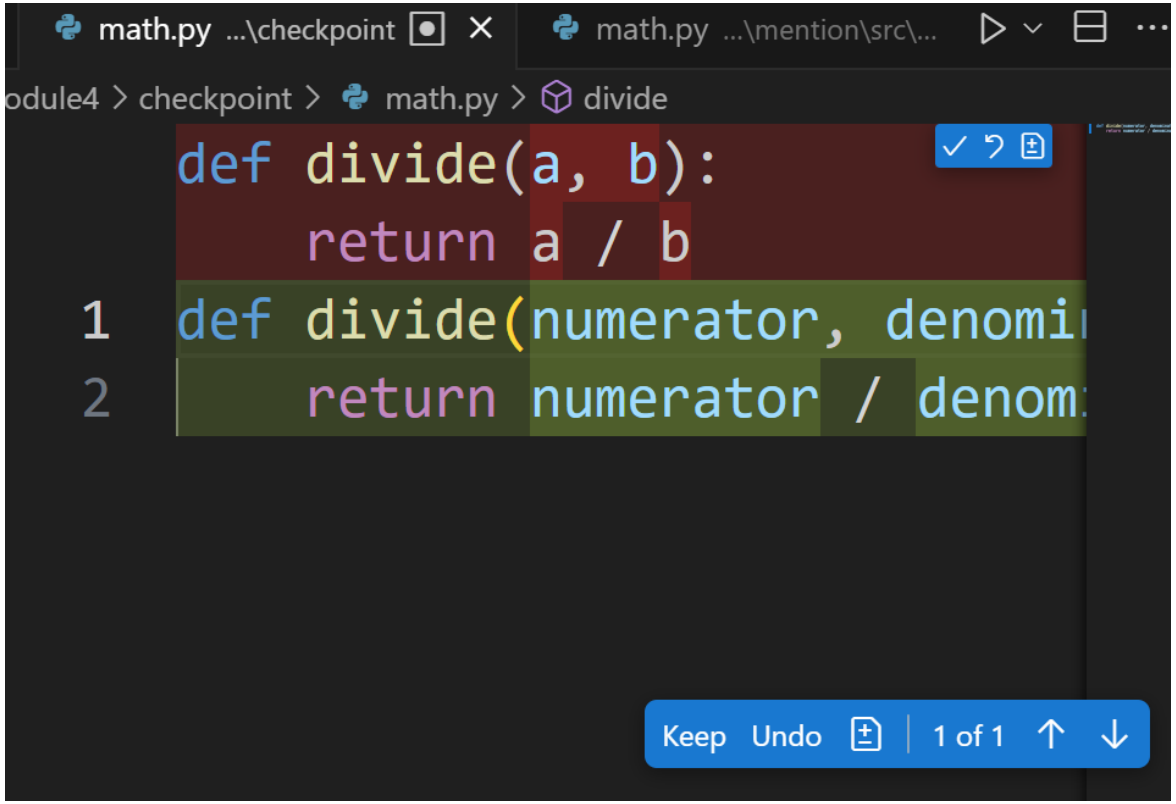


DIFF VIEW

If you decide to apply the suggestion, then the suggestion is presented within a diff view, with the suggestion shown below the original. Here are the allowable actions:

- Keep the suggestion
- Reject the suggestion
- Toggle editor marks
- Scroll through suggestions

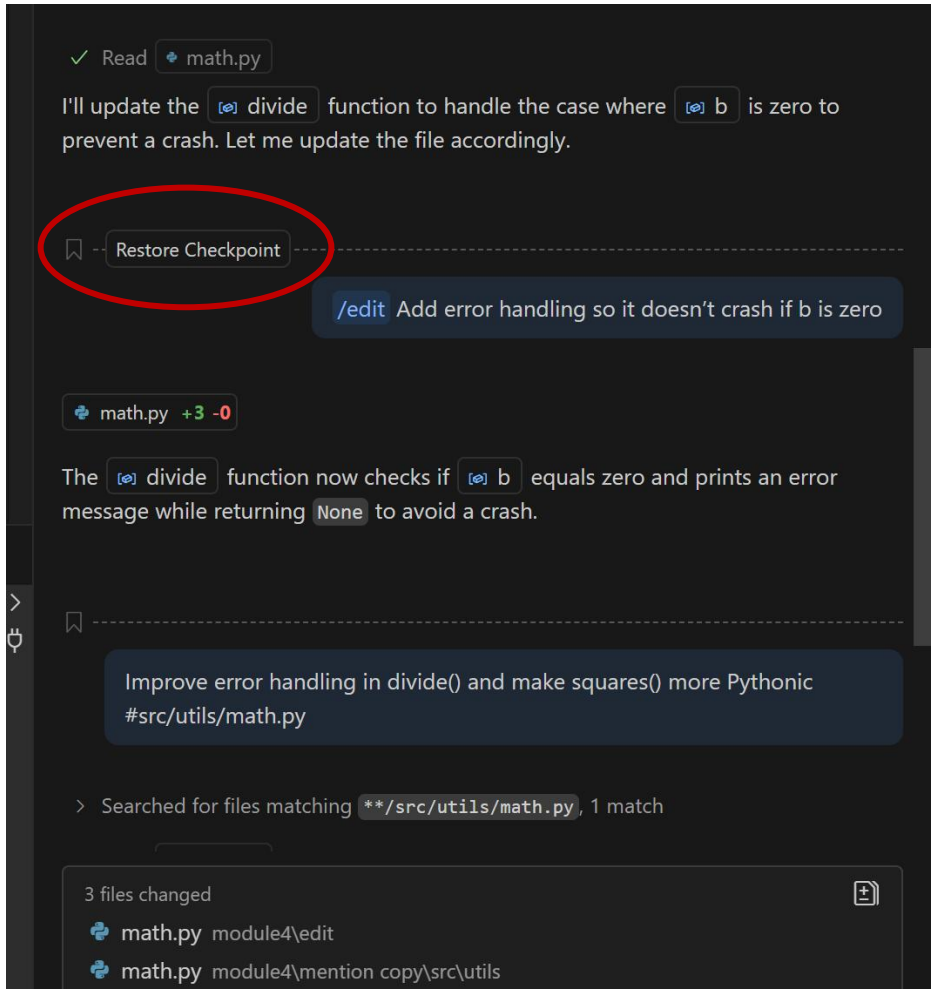
KEEP CONTROL



The Keep control allows:

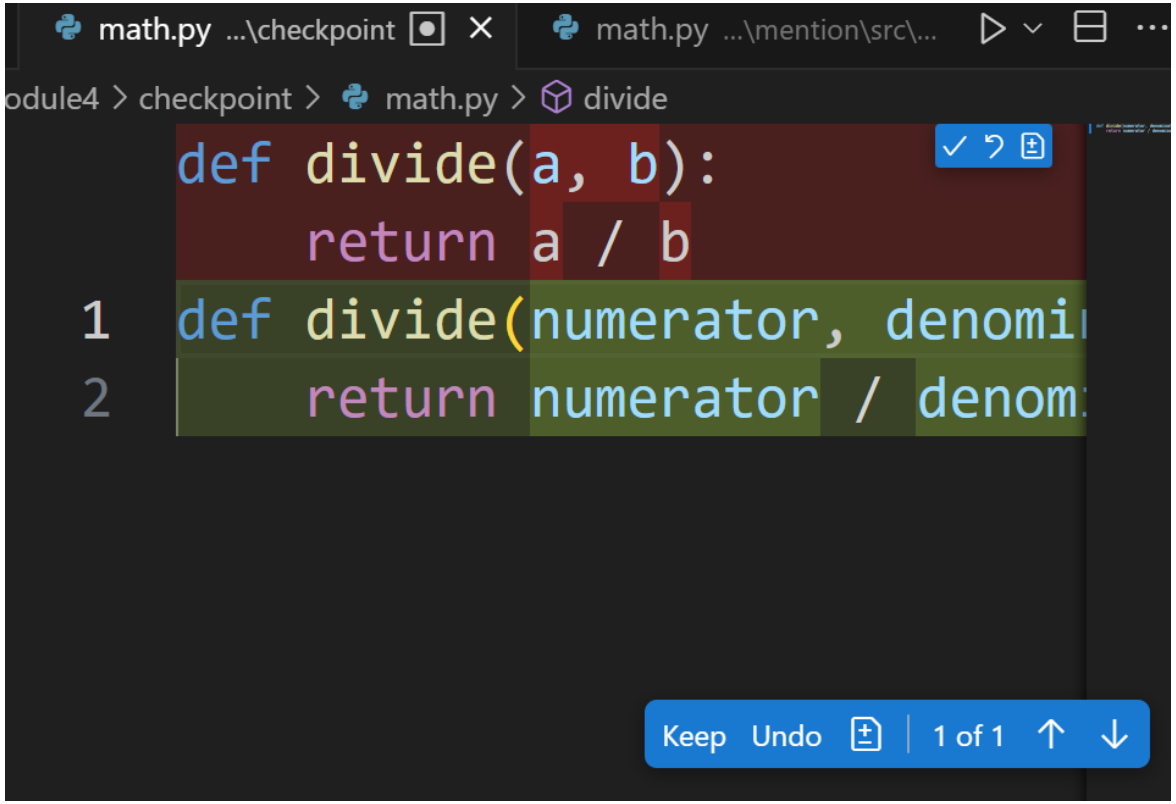
- Keep a suggestion
- Undo (reject) a suggestion
- Arrow through available suggestions

CHAT PANE



In the Copilot Chat pane, checkpoints will also appear when hovering over the related action.

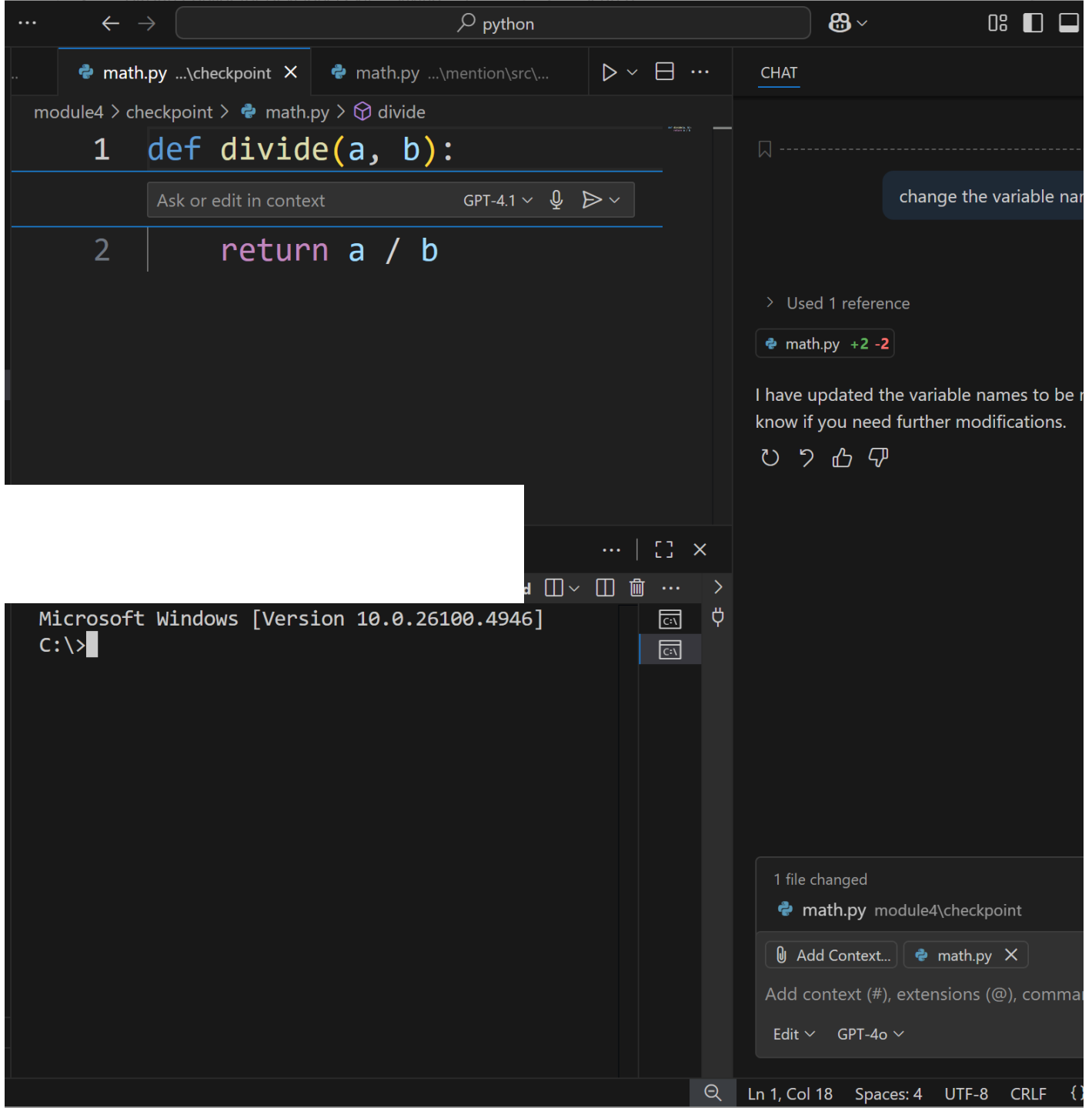
EDIT MODE



With the edit mode, you skip the staging (i.e., Add mode) and the suggestion is immediately presented in the Diff view.

You can then accept or reject the suggestion with the Diff Editor control.

INLINE EDITOR

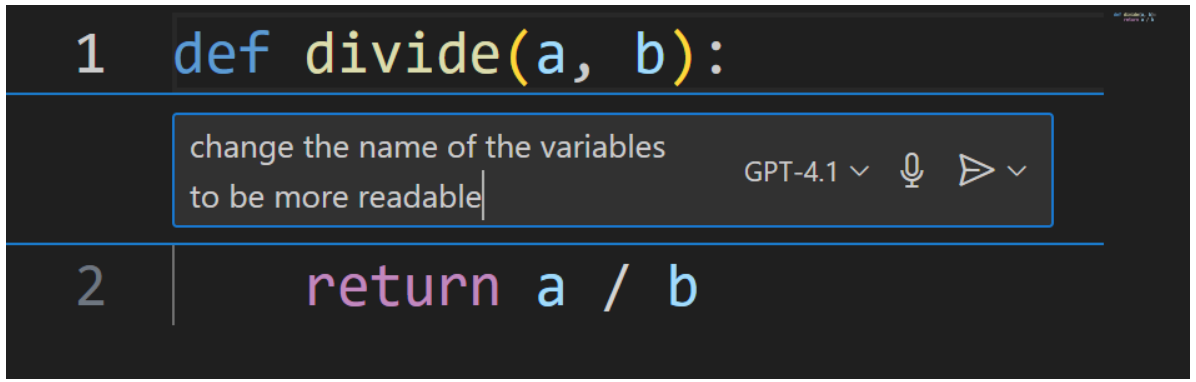


INLINE CHAT WINDOW

The inline overlay is a lightweight tool meant for quick edits or explanations directly in your code. It has no conversation history — once closed, the exchange is gone — and doesn't support context tools like #file or #codebase. It generates a simple preview with Accept/Discard, tied only to the code you highlighted. Checkpoints are limited here: you can undo changes using VS Code's native undo, but you don't get the richer checkpoint markers or history that appear in the side panel.

- Much simpler: just type instructions in a small inline box.
- No conversation history — once you close it, it's gone.
- Generates a preview with Accept/Discard only, tied to the code you highlighted.
- No extra context tools (#codebase, #file, etc.) in this view.

INLINE CONTROLS



The first button allows you to change the LLM.

After entering a command, accept the command with the Enter key or the Send and Dispatch button.

CHECKPOINT EXAMPLE

Highlight the divide function, press Ctrl+I (Windows/Linux) or Cmd+I (macOS), and type:

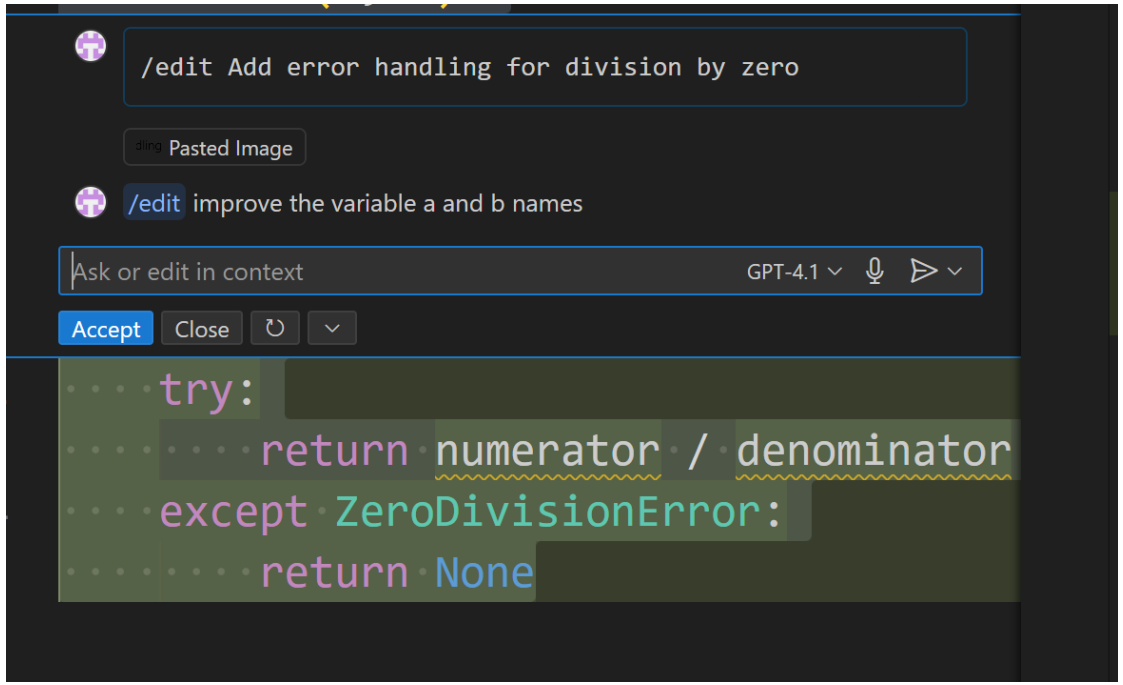
Add error handling for division by zero

Copilot Suggestion:

```
def divide(a, b):
    if b == 0:
        return "Error: Division by zero"
    return a / b
```

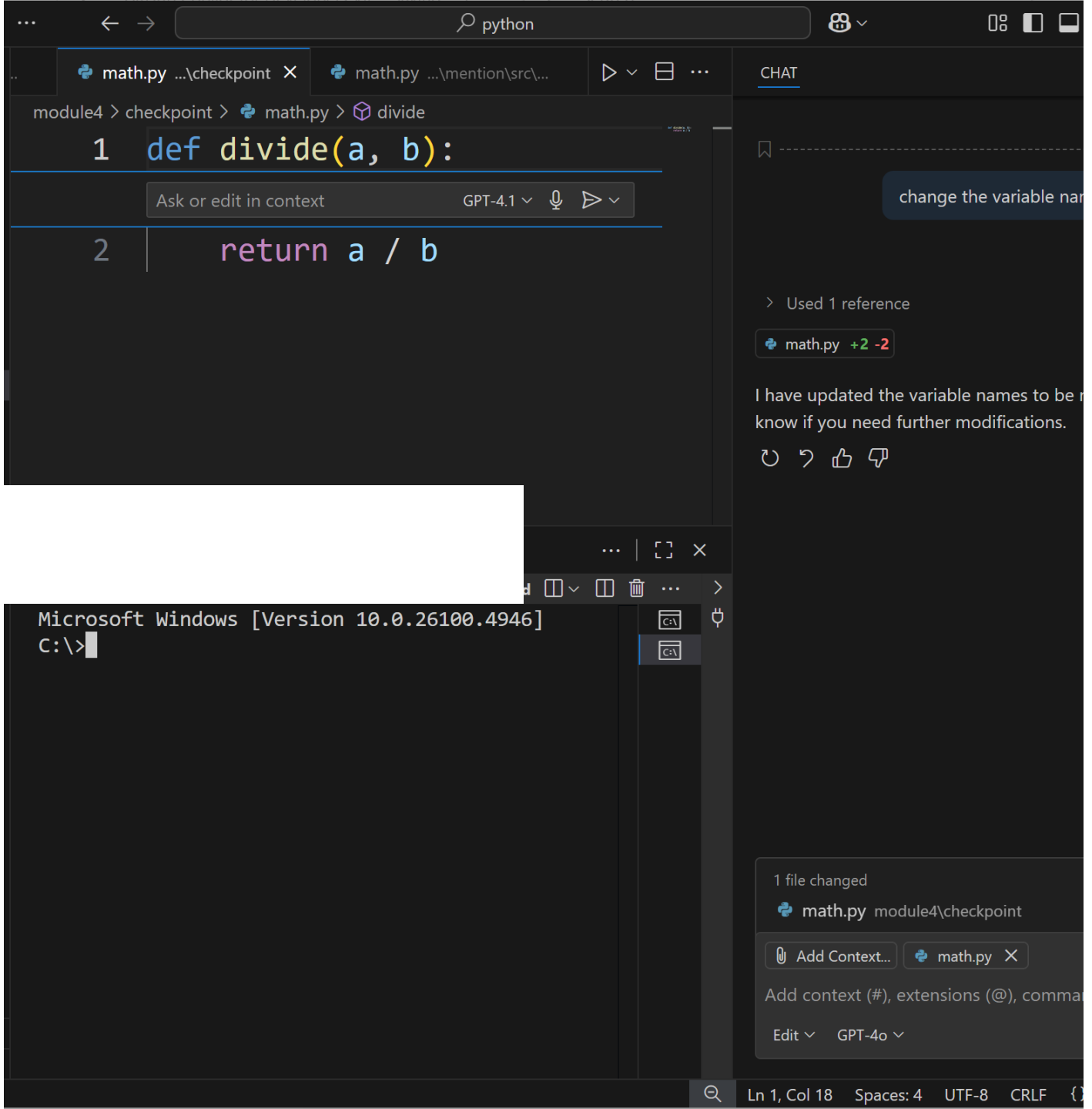
A checkpoint is automatically created, storing the original version before applying this change.

CHECKPOINT EXAMPLE



You'll see a small "checkpoint" marker above the edited section.

OTHER FEATURES



VARIABLES

Copilot fully understands variables you've declared in the codebase.

Example:

```
total = price * quantity
```

If you then write a comment like

```
# calculate tax on total
```

Copilot knows total is a variable and will suggest code like:

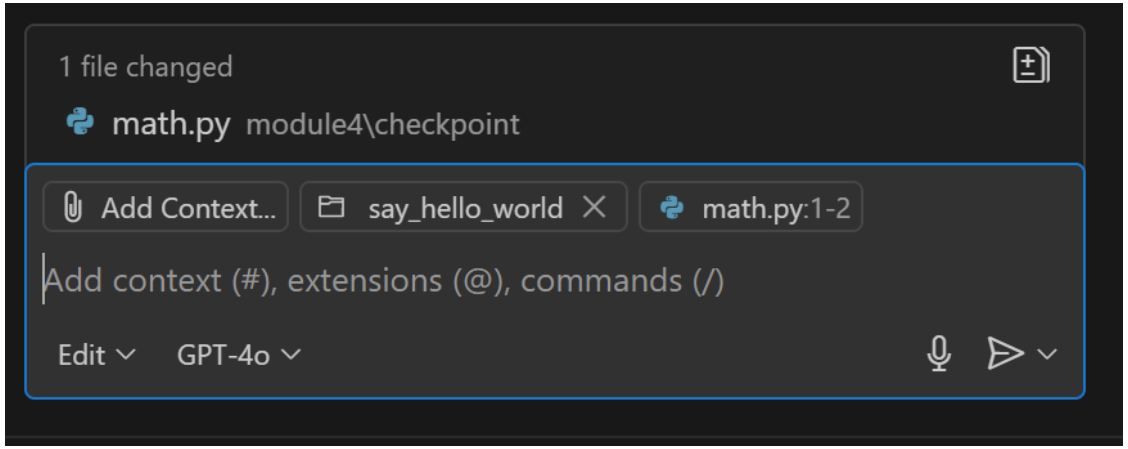
```
tax = total * 0.08
```

Copilot's completions are context-aware. It looks at variable names, types, and surrounding code when generating suggestions.

#MENTION

The “# mentions” method lets you manually give Copilot Chat extra context by referencing files, your whole workspace, or external resources directly in your prompt. This is useful if you don’t see the Add Context button in your version of VS Code

CONTEXT BUTTON



The Add Context button, which is a paperclip, is provided to add items to the context, such as files.

It will open a dialog listed resources that can be selected.

MENTION

The "# mentions" method lets you manually give Copilot Chat extra context by referencing files, your whole workspace, or external resources directly in your prompt. This is useful if you don't see the Add Context button in your version of VS Code.

- Add # followed by the context you want:
- #codebase: makes Copilot aware of the entire workspace/project.
- #<filename>: points to a specific file, e.g. #app.js.
- #styles.css: adds supporting files like CSS or config files.

MENTION EXAMPLE

Here's a short example of using file path mentions in Copilot Chat: Here is the project structure:

```
project/
  src/
    utils/
      math.py
  styles/
    main.css
```

Prompt in Copilot Chat:

> Explain how the functions are organized #src/utils/math.py.
>
> Suggest improvements to button styling #styles/main.css.

Result:

> Copilot looks at math.py to describe your utility functions. It also uses main.css to suggest design tweaks.

#SELECTION

```python
def process_data(records):
    cleaned = []
    for r in records:
        if r is not None and r != "" and r.strip() != "":
            cleaned.append(r.strip().lower())
    # ... more logic for filtering, transforming, exporting ...
    return cleaned
```

You can focus suggestions on selected code with #selection, such as a portion of a function.

For example:

#selection optimize

#SELECTION #STYLE EXAMPLE

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Button Example</title>
  <!-- Later we will replace style.css with new-styles.css -->
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <button class="primary-btn">Submit</button>
  <button class="secondary-btn">Cancel</button>
</body>
</html>
```

```
/* Base button template */
button {
  font-family: Arial, sans-serif;
  font-size: 14px;
  padding: 10px 20px;
  margin: 5px;
  cursor: pointer;
}
```

NEW-STYLES.CSS

```css
/* Styles for the primary button */
.primary-btn {
  background-color: blue;
  color: white;
  border: none;
  border-radius: 8px;
  padding: 10px 20px;
  font-size: 16px;
  cursor: pointer;
}

/* Styles for the secondary button *
.secondary-btn {
  background-color: gray;
  color: black;
  border: none;
  border-radius: 0;
  padding: 10px 20px;
  font-size: 16px;
  cursor: pointer;
}
```

#selection

#file:style.css

Create new-styles.css for index.html with styles for .primary-btn (blue, white text, rounded) nd .secondary-btn (gray, black text, square).

/EXPLAIN

```
def calculate_prime(n: int) -> bool:
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Explains the code you have selected or the file that's active in the editor. Great for onboarding to an unfamiliar area or validating logic before changes.

/FIX

```
def divide_numbers(a, b):
    return a / b


print(divide_numbers(10, 0))
```

Analyzes the selected code and proposes a patch for errors or suspicious patterns. You'll get a diff you can accept or tweak.

/FIXTESTFAILURE

```
def parse_int_list(s: str) -> list[int]:
    return [int(x) for x in s.split(",")]
```

Targets a failing test and suggests code changes to make it pass. Use when your test runner (or CI) is red and you want a focused remedy.

/TESTS

```
def factorial(n: int) -> int:
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Generates unit tests for your active file or selection, including typical and edge cases. You can specify frameworks or constraints.

POP QUIZ: /TESTS COMMAND

Why is unittest imported in the test_factorial.py file?

10 MINUTES

/EXPLAIN

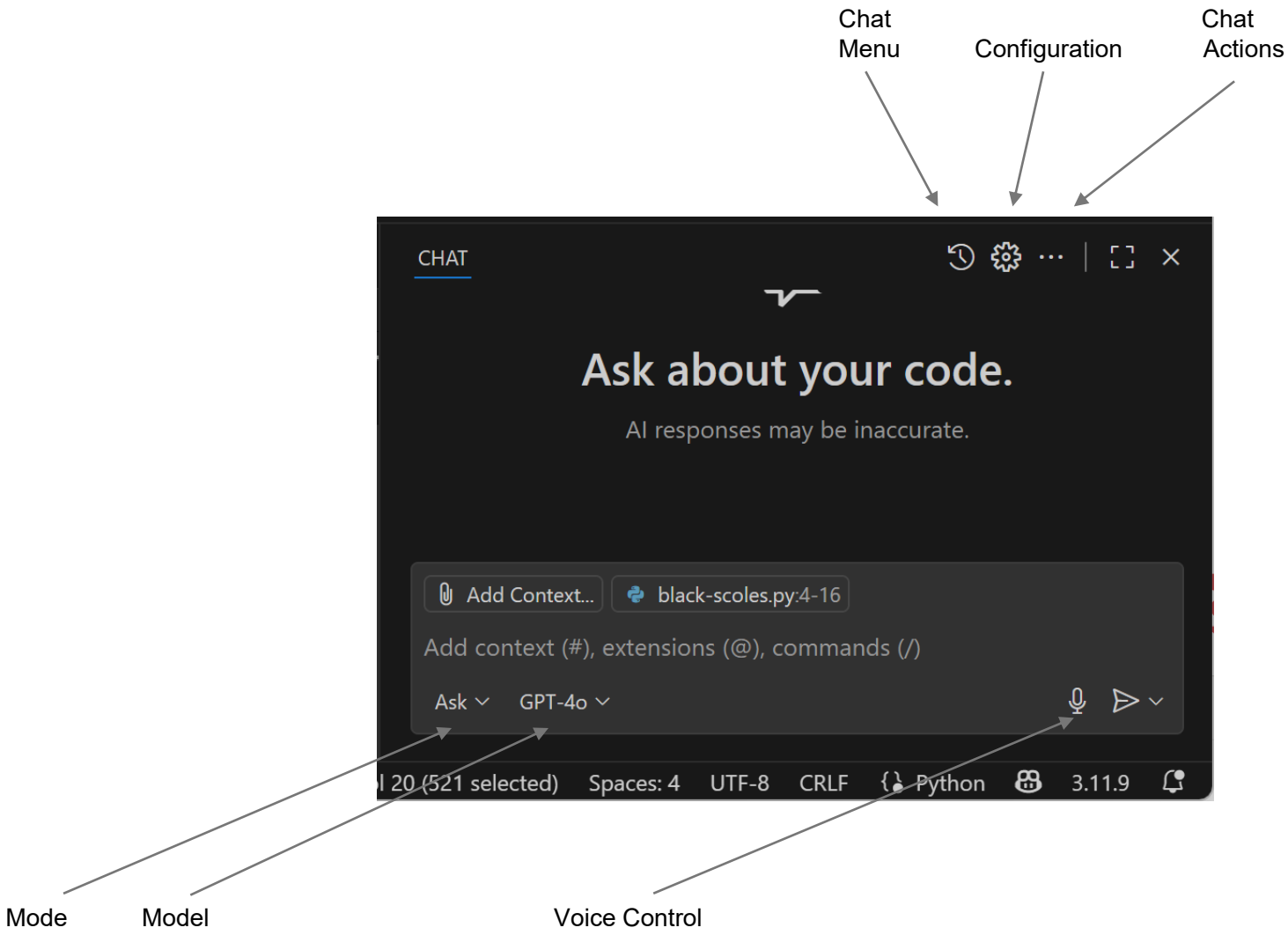
```
import math
from scipy.stats import norm

def black_scholes_call(S, K, T, r, sigma):
    try:
        if S <= 0 or K <= 0 or T <= 0 or sigma <= 0:
            raise ValueError("All inputs must be
positive.")

        d1 = (math.log(S / K) + (r + 0.5 * sigma**2)
* T) / (sigma * math.sqrt(T))
        d2 = d1 - sigma * math.sqrt(T)

        call_price = S * norm.cdf(d1) - K *
                math.exp(-r * T) * norm.cdf(d2)
        return round(call_price, 4)
    except Exception as e:
        print(f"Error calculating Black-Scholes
                price: {e}")
        return None
```

CHAT INPUT BOX



SHORTCUT KEYS

Action	Windows/Linux	macOS
Open Copilot Chat panel	`Ctrl` + `I`	`Cmd` + `I`
Open Inline Chat (editor overlay)	`Alt` + `Enter`	`Option` + `Enter`
Accept suggestion	`Tab`	`Tab`
Cycle inline suggestions	`Alt` + `[` / `Alt` + `]`	`Option` + `[` / `]`
Submit prompt in chat	`Enter`	`Enter`
Newline in chat prompt	`Shift` + `Enter`	`Shift` + `Enter`
Open chat with context	`Ctrl` + `Shift` + `\\`	`Cmd` + `Shift` + `\\`
Open chat from selection	`Right-click > Copilot: Explain this code`	Same

RETROSPECTIVE

This module explains how to use GitHub Copilot Chat in VS Code across Chat View, Inline Chat, and Quick Chat, and it helpfully walks through the Copilot Chat/Completions menus (status, open completions panel, disable completions, shortcuts, settings, diagnostics/logs).

It also covers modes (Ask, Edit, Agent), core slash commands (/fix, /tests, /explain, /edit), checkpoints and the action bar

LAB 4 – MOBY DICK

The goal of this lab is file processing – starting with Moby Dick.

MOBY DICK

Call me Ishmael. Some years ago—never mind how long precisely—having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off—then, I account it high time to get to sea as soon as I can. This is my substitute for pistol and ball…

FILE PROCESSING

Goal: Use Copilot Chat to perform text processing

1. Create a lab directory. In the lab directory, add a new file called text_summary.py.
2. Add a comment:

- # Read a file and summarize the number of lines, words, and characters
- Accept Copilot's function suggestion.

3. Copy mobydick.txt from the Lab directory to your directory, if not the same.
4. Run the script and verify output.

REVISING

Ask Copilot Chat:

- Suggest improvements for handling empty files
- Add error handling if mobydick.txt is missing
- Display the run time of the script in milliseconds

2. Refactor the script to support command-line arguments so different file names can be provided.

3. Use the /tests command to create unit tests

4. Run the unit test(s)

Lab completed

COMON TASKS

ERROR HANDLING, SYNTAX CHECKING, AND MORE

INTRODUCTION

The presentation introduces GitHub Copilot Chat in VS Code as a conversational coding assistant. It expands beyond inline completions by supporting natural language queries, code edits, and even multi-step workflows through Ask, Edit, and Agent modes

It also highlights practical tools such as checkpoints to track and roll back edits, action controls to apply or discard suggestions, and inline chat for quick changes. Features like slash commands, #mentions for context, and variable awareness show how Copilot integrates smoothly into the coding workflow.

COPILOT CHAT

This module introduces GitHub Copilot Chat as a practical debugging partner: it spots and explains common mistakes (e.g., removing items while iterating a list), scaffolds real error handling with clear try/except patterns, helps fix syntax slips across languages, and—most importantly—catches "silent" logic errors that still run but produce wrong results.

EXAMPLE

```
def remove_negatives(numbers):
    for n in numbers:
        if n < 0:
            numbers.remove(n)
    return numbers

data = [3, -1, 5, -2, 7]
print(remove_negatives(data))
```

POP QUIZ: PROBLEM

From the previous slide, do you know the problem in this code?

10 MINUTES

COPILOT CHAT – THE FIX

```
def remove_negatives(numbers):
    return [n for n in numbers if n >= 0]


data = [1, -1, -2, 3, -4, -5, 6]


print(remove_negatives(data))
```

The issue is that you're modifying the list while iterating over it. When you remove an element during iteration, Python shifts the remaining items left, so the loop skips over some elements.

ERROR HANDLING

Error handling is an area where Copilot shines by quickly generating try-except or try-catch structures. Many developers overlook robust error handling in the early stages of coding.

Copilot can encourage better practices by suggesting error handling blocks automatically.

EXAMPLE

```
def compute_average(filename):
    with open(filename, "r") as f:
        lines = f.readlines()
        total = sum(int(line.strip()) for line in lines)
        avg = total / len(lines)
return avg
```

WITH ERROR HANDLING

```
def compute_average(filename):
    try:
        with open(filename, "r") as f:
            lines = f.readlines()
            if not lines:
                raise ValueError("The file is empty.")
            total = sum(int(line.strip()) for line in lines)
            avg = total / len(lines)
            return avg
    except FileNotFoundError:
        print(f"Error: The file '{filename}' was not
found.")
    except ValueError as ve:
        print(f"Error: {ve}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

Error handling is an example of repetitive code, such as try-except or try-catch structures, that GitHub Copilot handles exceptionally well.

SYNTAX ERRORS

Syntax errors are among the most common issues developers encounter, from missing colons in Python to mismatched brackets in JavaScript. Copilot helps resolve these by offering corrected syntax based on language rules. For example, if you mistakenly write if (x > 10) in Python, Copilot may recognize the missing colon and suggest if x > 10:. Similarly, if you open an HTML tag and forget to close it, Copilot often generates the closing tag for you.

Another example is in Java: if you write System.out.println("Hello World") without a semicolon, Copilot can highlight the issue by suggesting the corrected line. These corrections are particularly useful for beginners or for developers working in a language they do not use frequently.

MUST BE GOOD!

This is especially important for syntax errors that do not cause compilation errors. In languages like Python, certain mistakes still produce valid code, so the program runs but behaves incorrectly.

These errors are much harder to detect because they don't trigger obvious failures — they just produce wrong results.

EXAMPLE

```
def check_value(x):
    message = ""
    magic_number = 10
    if (y := magic_number):
        if x > y:
            message = f"x is greater than {y}"
        elif x < y:
            message = f"x is less than {y}"
        else:
            message = f"x is equal to {y}"
    else:
        message = "Condition failed"
    return message
```

```
print(check_value(5))
print(check_value(10))
print(check_value(20))
```

FIXED

```
def check_value(x):
    message = ""
    magic_number = 10

    if x > magic_number:
        message = f"x is greater than
{magic_number}"
    elif x < magic_number:
        message = f"x is less than
{magic_number}"
    else:
        message = f"x is equal to
{magic_number}"

    return message
```

GitHub Copilot would quickly find and fixed the syntax / logical error.

GENERAL DEBUGGING

Copilot can also assist in debugging simple programs by suggesting possible reasons for logic errors and recommending alternative approaches. For example, if your loop is written incorrectly, such as for i in range(1, 10) when you intended to include 10, Copilot might suggest adjusting the loop or using range(1, 11). Similarly, if you implement a sorting function but forget to return the sorted list, Copilot may propose adding a return sorted_list statement at the end of the function.

This becomes more powerful when combined with descriptive comments. Writing # fix the bug in this function above a function definition may lead Copilot to review the logic and suggest corrections.

EXAMPLE

Copilot can assist in debugging simple programs by suggesting possible reasons for logic errors and recommending alternative approaches. For example, if your loop is written incorrectly, such as for i in range(1, 10) when you intended to include 10, Copilot might def fibonacci(n):

```
def fibonacci(n):
    return fibonacci(n - 1) + fibonacci(n - 2)
```

RETROSPECTIVE

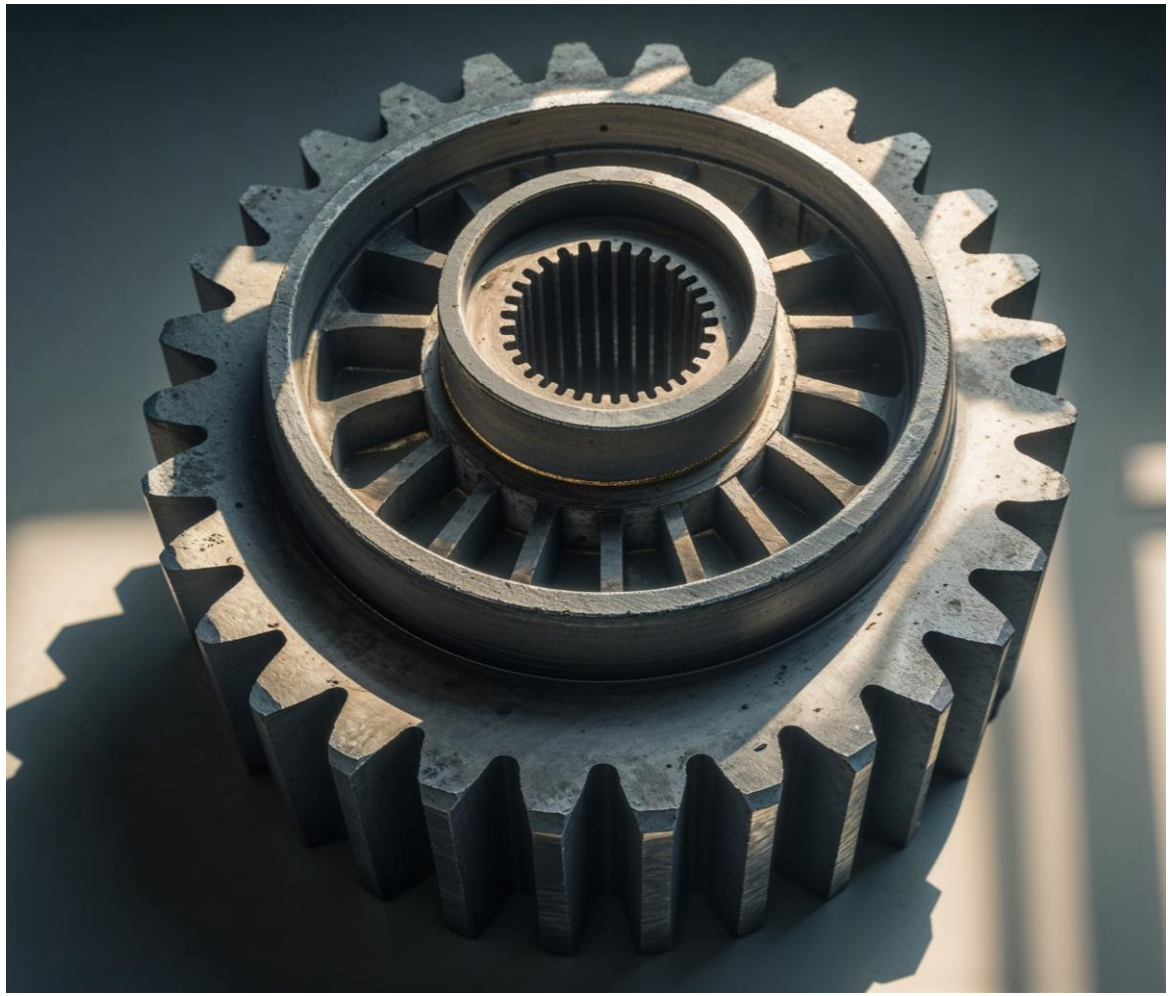
This module clearly shows how Copilot Chat helps you catch and fix everyday mistakes: mutating a list while iterating, undefined or incorrect calls, and common syntax slips; it then levels up practice with real error handling (file I/O guards, empty-file checks via try/except) and highlights “silent” logic errors

LAB 5 – COMMON TASKS

For Python applications, complete common tasks using GitHub Copilot.

Lab instructions also in the readme.md file in the same directory.

SETUP



In Vscode, open the Lab folder for Module 5.

There are several files in the lab that require some assistance from GitHub Copilot.

The goal is to successfully execute each of the programs, in some manner.

PART A

A1 — part_a1.py

Run python part_a1.py and observe the failure.

Use Copilot to introduce the missing data so the greeting can be constructed correctly.

Re-run until it prints a valid welcome message.

A2 — part_a2.py

Run python part_a2.py and observe the error.

Use Copilot to correct the function call so it supplies appropriate arguments.

Re-run and verify the formatted name output looks correct.

PART B

Create a grades.txt file in this folder. Try with it empty; then with some invalid and then valid lines.

Valid grades.txt file:

90
85
95
100
80

Run python part_b1.py and note different failure modes.

Use Copilot to add robust error handling for file/parse issues and add a safe guard to avoid dividing by zero.

Re-run with valid numeric data and verify an average is produced.

PART C

Part C — Syntax Fixes

C1 — part_c1.py

Run python part_c1.py to see syntax failures.

Use Copilot to fix all syntax issues and indentation so the program runs.

Re-run and verify it prints a message and a small sequence of numbers.

PART D

D1 — part_d1.py (Factorial)

Run python part_d1.py and inspect the output/behavior.

Use Copilot to correct the logic so the function returns the correct factorial result.

(Optional) Ask Copilot for a non-recursive version to improve efficiency.

D2 — part_d2.py (Fibonacci Sequence)

Run python part_d2.py and examine the length/content of the output.

Use Copilot to adjust the logic so the function returns exactly n terms.

Re-run and verify the sequence length matches the input.

Lab completed

CODE DOCUMENTATION

INTRODUCTION

This module is a hands-on guide to using GitHub Copilot Chat for fast, consistent documentation and clearer code understanding. You'll practice context-aware generation of docstrings and short README snippets, refine drafts through quick iterative prompts, and learn simple prompt patterns that keep Copilot focused

DOCUMENTATION

Copilot now uses context-aware generation, meaning it considers the function signature, surrounding code, and even related files to suggest meaningful comments or documentation.

For example, when you begin typing a docstring in Python or a Javadoc comment in Java, Copilot can propose a full template that includes a description, parameter explanations, and return values.

SAMPLE CODE

```
def calculate_area(width: float, height: float) -> float:
    """
    Calculate the area of a rectangle.

    Args:
        width (float): The width of the rectangle.
        height (float): The height of the rectangle.

    Returns:
        float: The calculated area of the rectangle.
    """
    return width * height
```

ITERATIVE

Copilot Chat further enhances this process by letting developers ask questions such as “Generate docstrings for all functions in this file” or “Explain what this class does in one sentence.” Copilot analyzes the file and responds with suggested documentation that can be inserted directly into the code.

This workflow makes documentation creation interactive—developers can refine or regenerate explanations until they meet project standards, all without leaving the editor.

VEHICLE CLASS

```
class Vehicle:
    """Base class representing a general vehicle."""

    def __init__(self, brand: str, year: int):
        self.brand = brand
        self.year = year

    def start(self):
        """Start the vehicle."""
        return f"{self.brand} is starting."

    def stop(self):
        """Stop the vehicle."""
        return f"{self.brand} is stopping.
```

CAR CLASS

```
class Car(Vehicle):
    """Intermediate class representing a car, inherits from Vehicle."""

    def __init__(self, brand: str, year: int, doors: int):
        super().__init__(brand, year)
        self.doors = doors

    def honk(self):
        """Honk the car horn."""
        return f"{self.brand} goes 'beep beep!'"
```

ELECTRICCAR

```
class ElectricCar(Car):
    def __init__(self, brand: str, year: int, doors: int,
                 battery_capacity: int):
        super().__init__(brand, year, doors)
        self.battery_capacity = battery_capacity

    def charge(self):
        """Recharge the electric car's battery."""
        return f"{self.brand} is
                 charging with {self.battery_capacity} kWh capacity."
```

POP QUIZ: QUESTIONS

What questions and documentation do you have of the previous code?

10 MINUTES

OBJECT ORIENTED PROGRAMMING

In multilevel inheritance, each class in the hierarchy can add attributes, override methods, or change initialization logic. By the time you reach the most descendant class (the deepest child), its state is a mix of:

- Properties defined in the base class.
- Properties added by intermediate parent classes
- Overrides or redefinitions in subclasses.

QUESTIONS

This layering makes it difficult to immediately know:

- Which attributes exist (some may be shadowed or overwritten).
- Where each attribute was introduced (base vs. intermediate vs. child).
- What initialization code actually ran (since constructors in Python, Java, C++, etc. may call super() in different ways).

DOCUMENTATION - BEST PRACTICES

- Write the docstring first: start a docstring/Javadoc header and let Copilot scaffold params/returns, then edit for brevity.
- Constrain scope with context: use #selection (only selected code) or @file/@workspace to feed the right amount of code; smaller context → tighter docs.
- Seed with style hints: a first line like “Use imperative mood, ≤80-char lines, include examples” steers output.

PROMPT - BEST PRACTICES

- Iterate, don’t accept wholesale: accept a draft, then ask in Chat: “tighten language,” “remove redundancies,” “convert to bullets.”
- Template reuse via prompt files: keep a docstyle.md (tone, sections, examples). Reference it in prompts so Copilot stays consistent.
- Ask for contrasts/examples: “Add a 1-line example and a common pitfall” produces practical docs without bloat.

PROMPT - BEST PRACTICES - 2

docstring:

- #selection @file:utils.py Create a concise docstring: 1-line summary, args, returns, edge cases. Use imperative mood.

README snippet:

- @workspace Generate a minimal README section: install, run, test. Keep steps numbered and under 7 lines.

Refine for brevity:

- Tighten this to half the length; keep technical accuracy and example.

PROMPT - BEST PRACTICES - 3

Write the docstring first:

- Start a docstring/Javadoc header and let Copilot scaffold params/returns, then edit for brevity.

Constrain scope with context

- Use #selection (only selected code) or @file/@workspace to feed the right amount of code; smaller context → tighter docs.

Seed with style hints:

- A first line like “Use imperative mood, ≤80-char lines, include examples” steers output.

PROMPT - BEST PRACTICES - 4

Iterate, don’t accept wholesale:

- Accept a draft, then ask in Chat: “tighten language,” “remove redundancies,” “convert to bullets.”

Write the docstring first:

- Start a docstring/Javadoc header and let Copilot scaffold params/returns, then edit for brevity.

Constrain scope with context

- Use #selection (only selected code) or @file/@workspace to feed the right amount of code; smaller context → tighter docs.

Seed with style hints:

- A first line like “Use imperative mood, ≤80-char lines, include examples” steers output.

PROMPT - BEST PRACTICES - 5

Template reuse via prompt files:

- Keep a docstyle.md (tone, sections, examples). Reference it in prompts so Copilot stays consistent.

Ask for contrasts/examples:

- “Add a 1-line example and a common pitfall” produces practical docs without bloat.

RETROSPECTIVE

This module showed how to use GitHub Copilot for fast, consistent documentation: it demonstrated context-aware generation of docstrings that consider signatures, surrounding code, and related files.

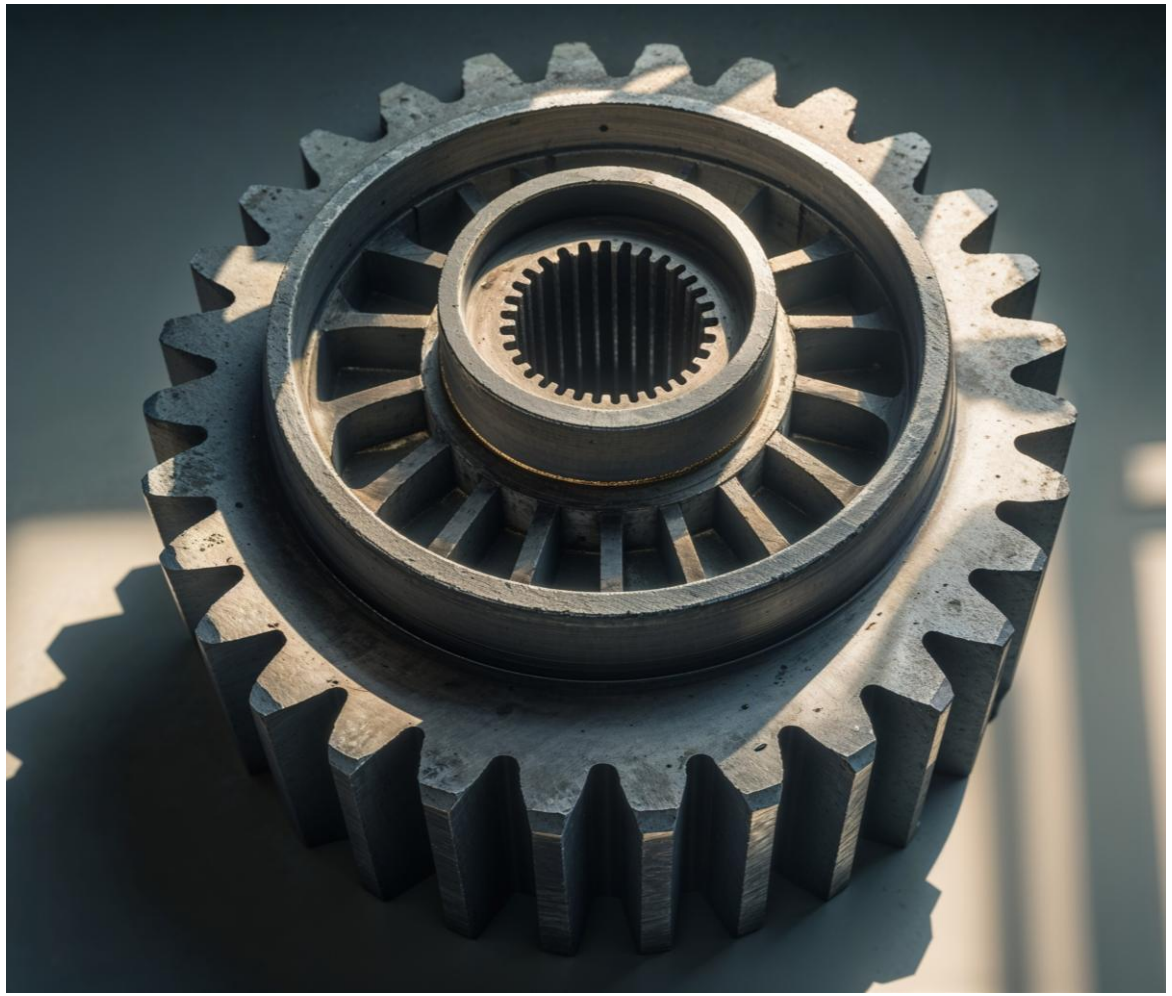
LAB 6 – ESCAPE VELOCITY

Document an application that computes escape velocity at a given altitude.

Estimates propellant mass to achieve that Δv via the rocket equation,

Uses Copilot to add docstrings and to create a README.md with a function table and a data table of examples.

SETUP



In Vscode, open the Lab folder for Module 5.

There are several files in the lab that require some assistance from GitHub Copilot.

The goal is to successfully execute each of the programs, in some manner.

SMOKE TEST

From the project folder, run in the module5/lab/space-lab directory:

```
python main.py
```

You should see three lines (escape speed in m/s, Δv in m/s, propellant in kg).

If it fails, fix runtime issues first (paths, Python version, etc.).

```
C:\Code\Copilot\class-prep\python\module6\lab\space-lab>python main.py
Escape speed: 11015 m/s
Δv (with margin): 12667 m/s
Propellant required: 66747 kg
```

Familiarize yourself with the files in the directory.

DOCSTRINGS

Goal! Add short docstrings to existing code and generate a simple README with a function table and a tiny data table — using Copilot Chat.

1) Docstring for AstroMath.escape_velocity

Do: Add a short, clear docstring.

Action: Open astro.py, select only the escape_velocity function.

Enter in Copilot Chat (A then B then C):

- Write a short docstring. One sentence.
- Add Args and Returns with units.
- Add formula text: v = sqrt(2*mu/r). Max 6 lines total.

DOCSTRINGS - 2

2) Docstring for AstroMath.propellant_needed

Do: Add a short docstring that explains the rocket equation result.

Action: Select only the propellant_needed function.

Enter in Copilot Chat (A then B then C):

- Short docstring. One sentence about the rocket equation.
- Add Args: dv m/s, isp s, dry_mass kg. Returns: propellant kg.
- Keep under 6 lines. No derivation.

DOCSTRINGS - 3

4) Docstring for MissionPlanner (class)

Do: Describe the pipeline and the loss margin.

Action: Select the class header and init together.

Enter in Copilot Chat (A then B then C):

- Short class docstring. One or two sentences.
- Say steps: compute escape speed, add dv margin, compute propellant.
- Explain loss_margin in plain English. Max 5 lines.

DOCSTRINGS - 4

Create README.md (sections + table + example)

Do: Generate a short README with simple words.

Action: Open empty README.md.

Enter in Copilot Chat (A then B then C):

- Create README with sections: Overview, Functions and Methods, Example. Use simple words.
- Add a table: Name | Purpose | Key args | Returns. Rows: AstroMath.escape_velocity, AstroMath.propellant_needed, MissionPlanner.fuel_to_escape.
- Keep under 30 lines.

TEST AND EXAMINE

Test the application. Naturally, it should still run.

View the files again to observe the requested changes.

Lab completed

ADVANCED

MENUS AND REFERENCE TOOLS