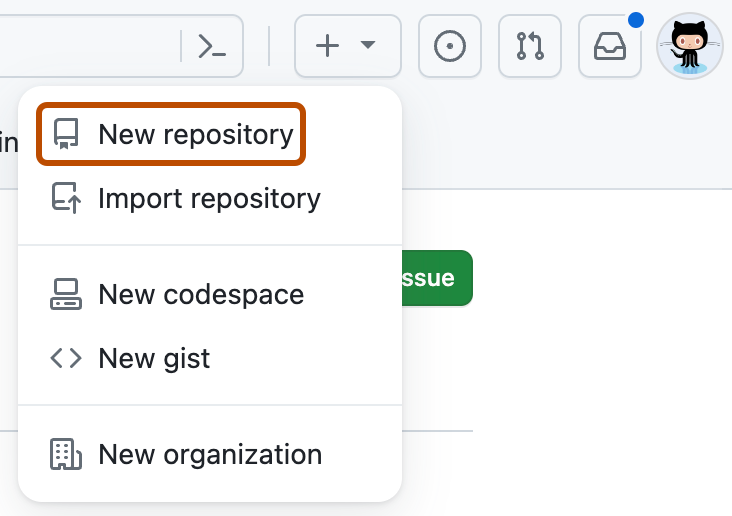


**Lab: Learn how to create and manage a repository on GitHub to enhance collaborative software development.**

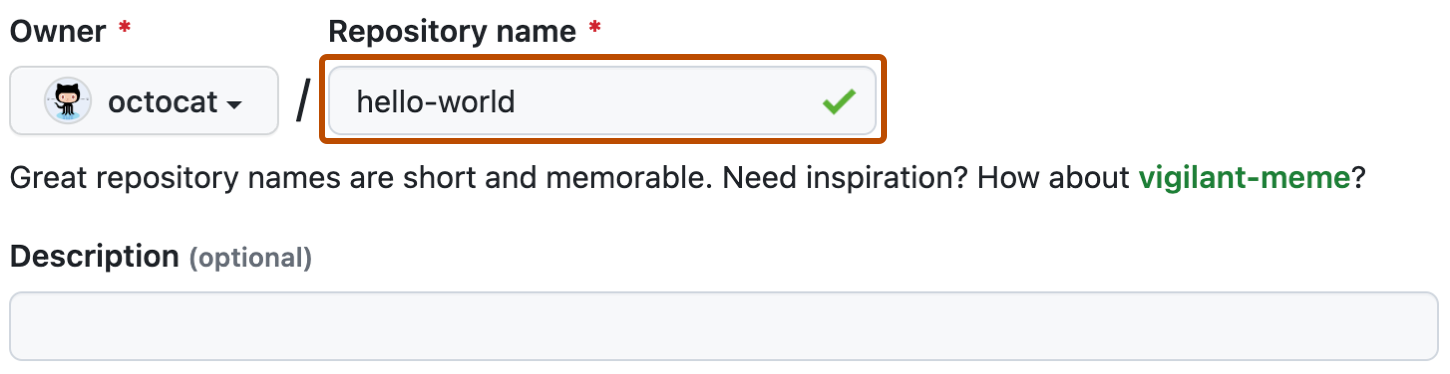
# [Create a repository](https://docs.github.com/en/repositories/creating-and-managing-repositories/quickstart-for-repositories#create-a-repository)

GitHub repositories store a variety of projects. In this guide, you'll create a repository and commit your first change.

1. In the upper-right corner of any page, select , then click **New repository**.



1. Type a short, memorable name for your repository. For example, "hello-world".



1. Optionally, add a description of your repository. For example, "My first repository on GitHub."
2. Choose a repository visibility. For more information, see "[About repositories](https://docs.github.com/en/repositories/creating-and-managing-repositories/about-repositories#about-repository-visibility)."
3. Select **Initialize this repository with a README**.
4. Click **Create repository**.

Congratulations! You've successfully created your first repository, and initialized it with a *README* file.

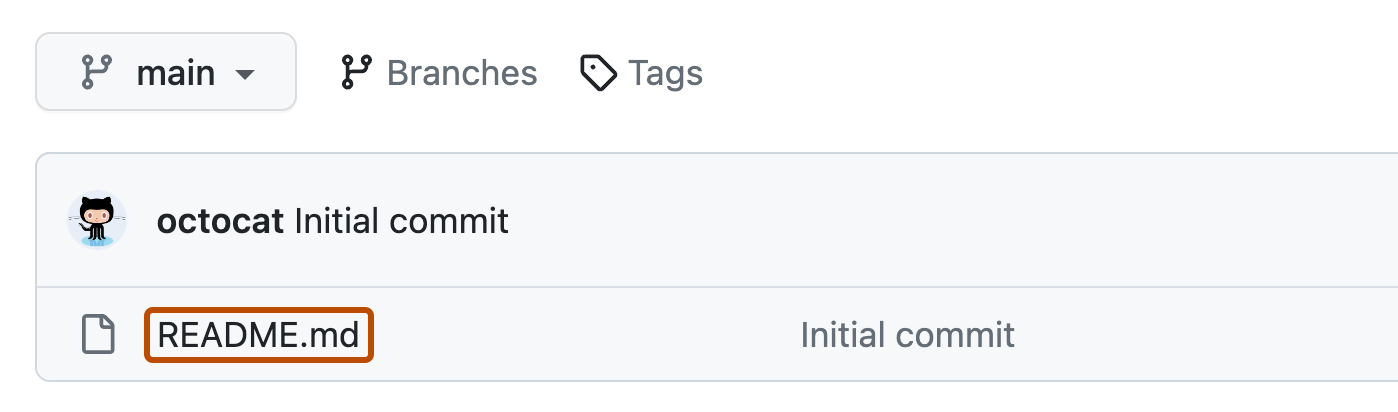
[**Commit your first change**](https://docs.github.com/en/repositories/creating-and-managing-repositories/quickstart-for-repositories#commit-your-first-change)

A [commit](https://docs.github.com/en/get-started/learning-about-github/github-glossary#commit) is like a snapshot of all the files in your project at a particular point in time.

When you created your new repository, you initialized it with a *README* file. *README* files are a great place to describe your project in more detail, or add some documentation such as how to install or use your project. The contents of your *README* file are automatically shown on the front page of your repository.

Let's commit a change to the README file.

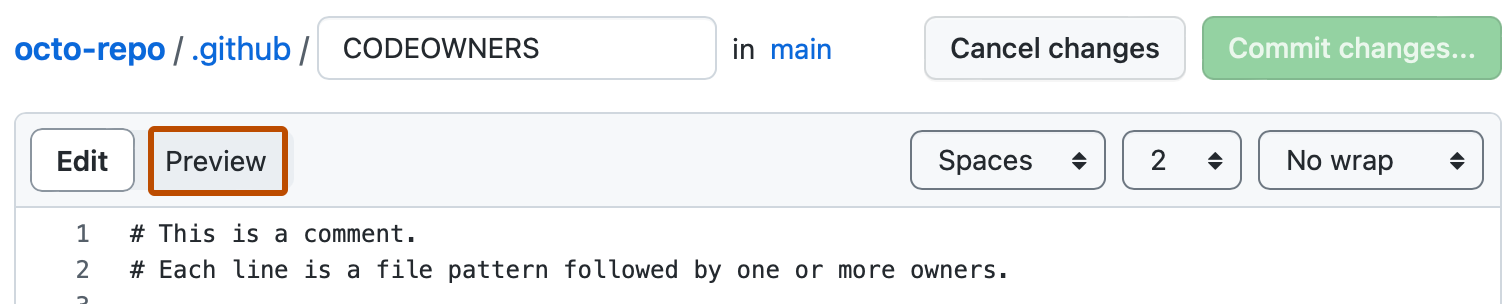
1. In your repository's list of files, select **README.md**.



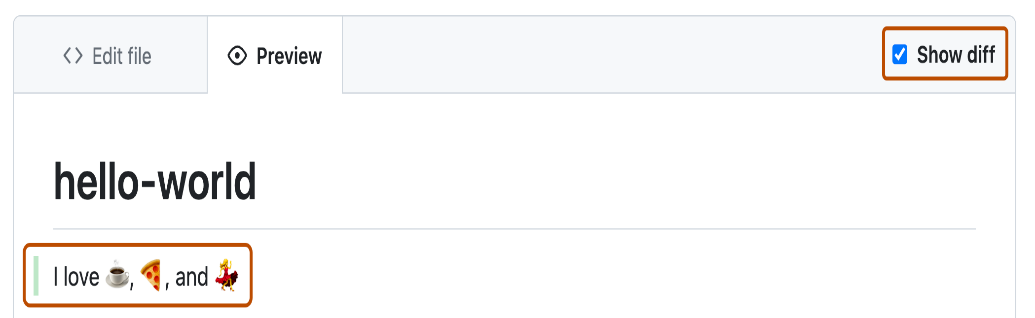
1. In the upper right corner of the file view, click  to open the file editor.



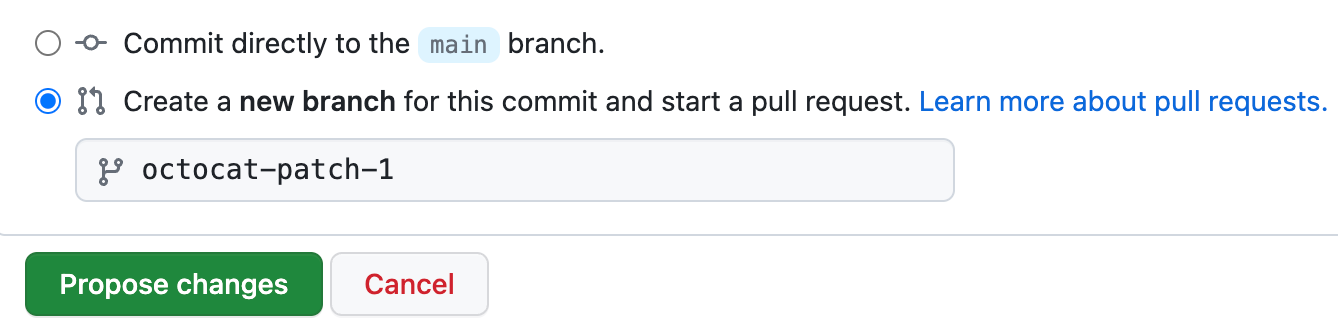
1. In the text box, type some information about yourself.
2. Above the new content, click **Preview**.



1. Review the changes you made to the file. If you select **Show diff**, you will see the new content in green.



1. Click **Commit changes...**
2. In the "Commit message" field, type a short, meaningful commit message that describes the change you made to the file. You can attribute the commit to more than one author in the commit message. For more information, see "[Creating a commit with multiple authors](https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/creating-a-commit-with-multiple-authors)."
3. Below the commit message fields, decide whether to add your commit to the current branch or to a new branch. If your current branch is the default branch, you should choose to create a new branch for your commit and then create a pull request. For more information, see "[Creating a pull request](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request)."

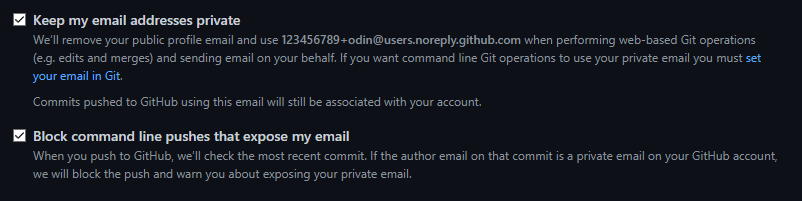


1. Click **Commit changes** or **Propose changes**.

# Configure Git and GitHub

**Create a GitHub account**

Go to GitHub.com and create an account! During the account setup, it will ask you for an email address. This needs to be a real email, and will be used by default to identify your contributions. If you are privacy conscious, or just don’t want your email address to be publicly available, make sure you tick the following two boxes on the Email Settings page after you have signed in:



Having these two options enabled will prevent accidentally exposing your personal email address when working with Git and GitHub.

You may also notice an email address under the Keep my email addresses private option. This is your private GitHub email address. If you plan to use this, make note of it now as you will need it when setting up Git in the next step.

**Setup Git**

For Git to work properly, we need to let it know who we are so that it can link a local Git user (you) to GitHub. When working on a team, this allows people to see what you have committed and who committed each line of code.

The commands below will configure Git. Be sure to enter your own information inside the quotes (but include the quotation marks)!

*git config --global user.name "Your Name"*

*git config --global user.email "yourname@example.com"*

If you opted to use the private GitHub email address, the second command will look something like this:

*git config --global user.email "123456789+odin@users.noreply.github.com" # Remember to use your own private GitHub email here.*

GitHub recently changed the default branch on new repositories from master to main. Change the default branch for Git using this command:

*git config --global init.defaultBranch main*

To enable colorful output with git, type

*git config --global color.ui auto*

You’ll also likely want to set your default branch reconciliation behavior to merging. You’ll learn what all those terms mean later in the curriculum, but for now just know that we suggest running the below command as part of the Git setup process when doing The Odin Project.

*git config --global pull.rebase false*

To verify that things are working properly, enter these commands and verify whether the output matches your name and email address.

*git config --get user.name*

*git config --get user.email*

# Commonly Used Git Commands

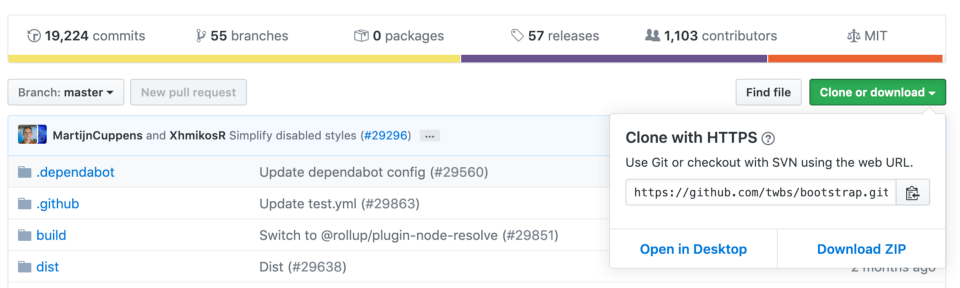
**1. Git clone**

Git clone is a command for downloading existing source code from a remote repository (like Github, for example). In other words, Git clone basically makes an identical copy of the latest version of a project in a repository and saves it to your computer.

There are a couple of ways to download the source code, but mostly I prefer the **clone with https** way:

*git clone <https://name-of-the-repository-link>*

For example, if we want to download a project from Github, all we need to do is click on the green button (clone or download), copy the URL in the box and paste it after the git clone command that I've shown right above.



**Bootstrap source code example from Github**

This will make a copy of the project to your local workspace so you can start working with it.

**2. Git branch**

Branches are highly important in the git world. By using branches, several developers are able to work in parallel on the same project simultaneously. We can use the git branch command for creating, listing and deleting branches.

**Creating a new branch:**

*git branch <branch-name>*

This command will create a branch **locally**. To push the new branch into the remote repository, you need to use the following command:

*git push -u <remote> <branch-name>*

**Viewing branches:**

*git branch or git branch --list*

**Deleting a branch:**

*git branch -d <branch-name>*

**3. Git checkout**

This is also one of the most used Git commands. To work in a branch, first you need to switch to it. We use **git checkout** mostly for switching from one branch to another. We can also use it for checking out files and commits.

git checkout <name-of-your-branch>

There are some steps you need to follow for successfully switching between branches:

* The changes in your current branch must be committed or stashed before you switch
* The branch you want to check out should exist in your local

**There is also a shortcut command that allows you to create and switch to a branch at the same time:**

git checkout -b <name-of-your-branch>

This command creates a new branch in your local (-b stands for branch) and checks the branch out to new right after it has been created.

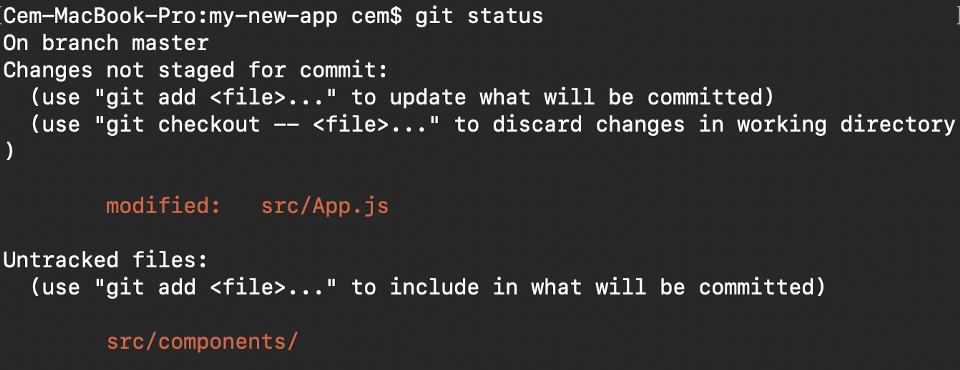
**4. Git status**

The Git status command gives us all the necessary information about the current branch.

*git status*

We can gather information like:

* Whether the current branch is up to date
* Whether there is anything to commit, push or pull
* Whether there are files staged, unstaged or untracked
* Whether there are files created, modified or deleted



**Git status gives information about the branch & files**

**5. Git add**

When we create, modify or delete a file, these changes will happen in our local and won't be included in the next commit (unless we change the configurations).

We need to use the git add command to include the changes of a file(s) into our next commit.

**To add a single file:**

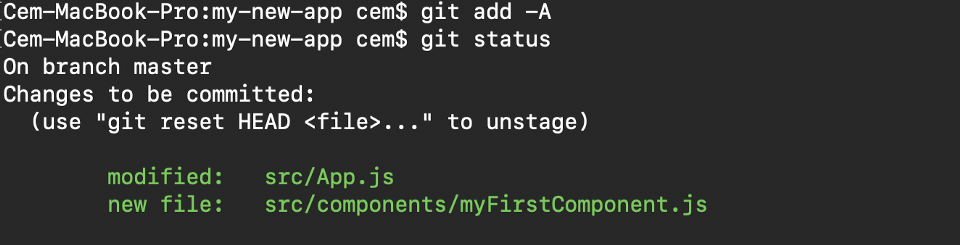
*git add <file>*

**To add everything at once:**

*git add -A*

When you visit the screenshot above in the 4th section, you will see that there are file names that are red - this means that they're unstaged files. The unstaged files won't be included in your commits.

**To include them, we need to use git add:**



Files with green are now staged with git add

**Important: The git add command doesn't change the repository and the changes are not saved until we use git commit.**

**6. Git commit**

This is maybe the most-used command of Git. Once we reach a certain point in development, we want to save our changes (maybe after a specific task or issue).

Git commit is like setting a checkpoint in the development process which you can go back to later if needed.

We also need to write a short message to explain what we have developed or changed in the source code.

*git commit -m "commit message"*

**Important: Git commit saves your changes only locally.**

**7. Git push**

After committing your changes, the next thing you want to do is send your changes to the remote server. Git push uploads your commits to the remote repository.

*git push <remote> <branch-name>*

However, if your branch is newly created, then you also need to upload the branch with the following command:

*git push --set-upstream <remote> <name-of-your-branch>*

or

*git push -u origin <branch\_name>*

**Important: Git push only uploads changes that are committed.**

**8. Git pull**

The **git pull**command is used to get updates from the remote repo. This command is a combination of **git fetch** and **git merge** which means that, when we use git pull, it gets the updates from remote repository (git fetch) and immediately applies the latest changes in your local (git merge).

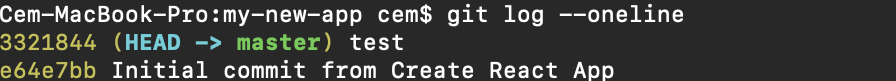
*git pull <remote>*

**This operation may cause conflicts that you need to solve manually.**

**9. Git revert**

Sometimes we need to undo the changes that we've made. There are various ways to undo our changes locally or remotely (depends on what we need), but we must carefully use these commands to avoid unwanted deletions.

A safer way that we can undo our commits is by using **git revert**. To see our commit history, first we need to use **git log -- oneline:**

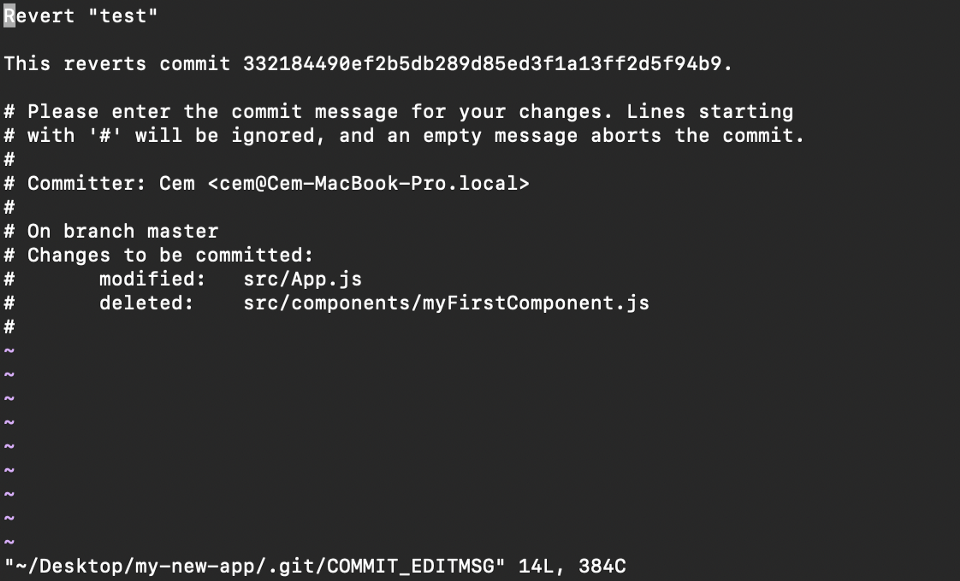


commit history of my master branch

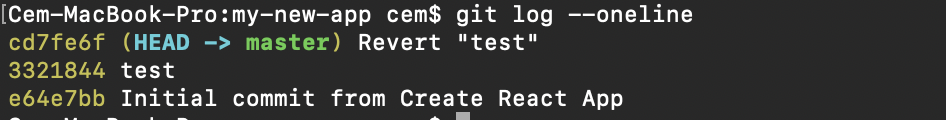
Then we just need to specify the hash code next to our commit that we would like to undo:

*git revert 3321844*

After this, you will see a screen like below - just press **shift + q** to exit:



The Git revert command will undo the given commit, but will create a new commit without deleting the older one:



new "revert" commit

The advantage of using **git revert** is that it doesn't touch the commit history. This means that you can still see all of the commits in your history, even the reverted ones.

Another safety measure here is that everything happens in our local system unless we push them to the remote repo. That's why git revert is safer to use and is the preferred way to undo our commits.

**10. Git merge**

When you've completed development in your branch and everything works fine, the final step is merging the branch with the parent branch (dev or master). This is done with the git merge command.

Git merge basically integrates your feature branch with all of its commits back to the dev (or master) branch. It's important to remember that you first need to be on the specific branch that you want to merge with your feature branch.

For example, when you want to merge your feature branch into the dev branch:

**First you should switch to the dev branch:**

*git checkout dev*

**Before merging, you should update your local dev branch:**

*git fetch*

**Finally, you can merge your feature branch into dev:**

*git merge <branch-name>*

# Collaborative development workflows

Make sure you have [git installed](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git) and that you have created an account on [GitHub](https://github.com/login).

**Starting your own project:**

1. Initiate git to create a local repository within the project directory using *git init*
2. [Create a new repository](https://help.github.com/en/github/creating-cloning-and-archiving-repositories/creating-a-new-repository) on GitHub
3. [Link the remote repository](https://help.github.com/en/github/importing-your-projects-to-github/adding-an-existing-project-to-github-using-the-command-line) on GitHub to your local repository
4. [Add Contributors](https://help.github.com/en/github/setting-up-and-managing-your-github-user-account/inviting-collaborators-to-a-personal-repository) to repository via Settings
5. Collaborators should [clone the repo](https://help.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository) to their create a local repository using *git clone repo-link.git*

**Doing work & updating GitHub**

Once you’ve got your local repository open, start with these steps to begin working on the project:

1. Create and start working on a local branch — *git checkout -b branch-name*
2. Make changes in working directory
3. Add changes to staging — *git add.*
4. Commit changes *git commit -m "Write changes here."*
5. Push changes to GitHub branch — *git push origin HEAD*
6. Create Pull Request to merge commit(s) to the main branch
7. Another collaborator reviews, resolves conflicts (if any), and merges PR
8. Switch to local main branch *git checkout main*
9. Pull down main changes from GitHub to local *git pull origin main*
10. Delete branch on GitHub & local if desired.

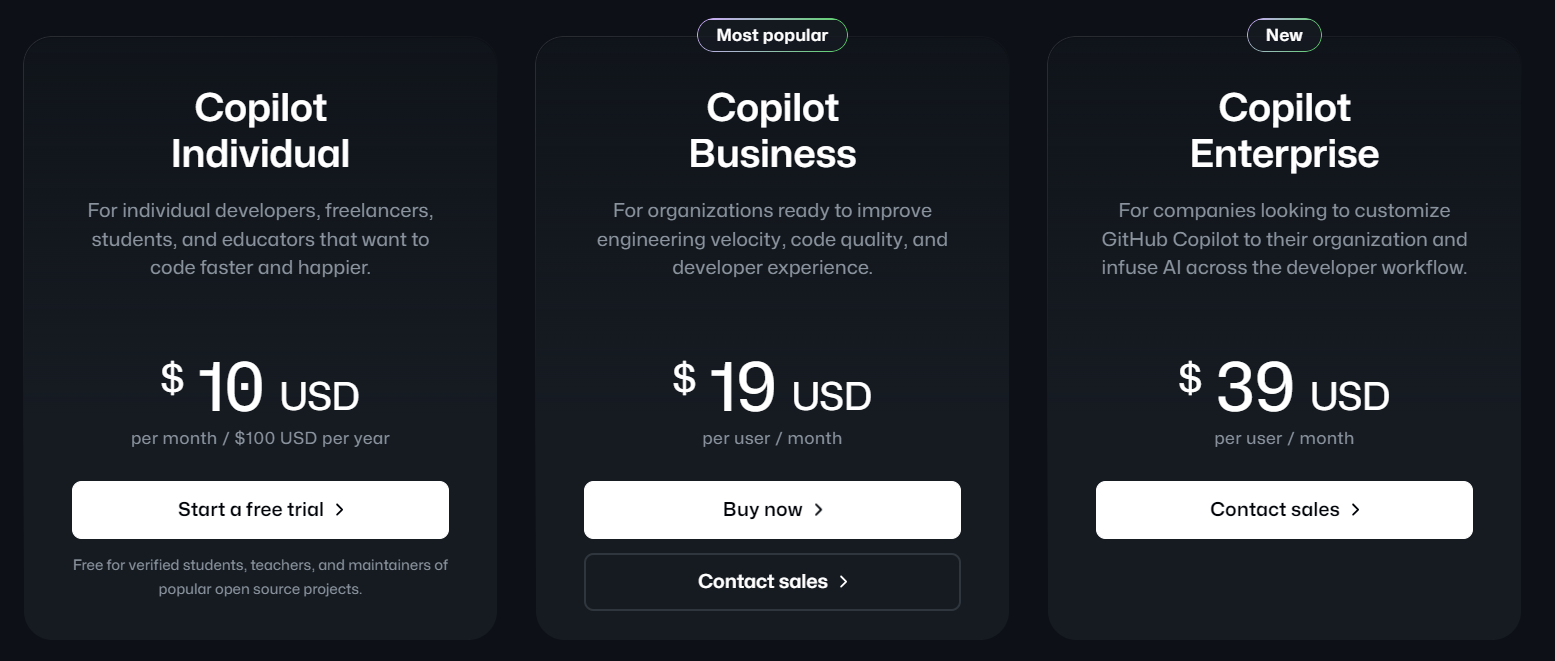
Repeat these steps as many times as necessary. Steps 2–4 can also be repeated in its own loop as it is important to commit often when you have working code. This will allow you to be descriptive in the changes that you’ve made and revert back to previous commits of working code if ever needed.

# Subscribe to GitHub Copilot

GitHub Copilot is an AI pair programmer that offers autocomplete-style suggestions as you code. You can receive suggestions from GitHub Copilot either by starting to write the code you want to use, or by writing a natural language comment describing what you want the code to do. GitHub Copilot analyzes the context in the file you are editing, as well as related files, and offers suggestions from within your text editor. GitHub Copilot is powered by a generative AI model developed by GitHub, OpenAI, and Microsoft.

GitHub Copilot is trained on all languages that appear in public repositories. For each language, the quality of suggestions you receive may depend on the volume and diversity of training data for that language. For example, JavaScript is well-represented in public repositories and is one of GitHub Copilot's best supported languages. Languages with less representation in public repositories may produce fewer or less robust suggestions.

GitHub Copilot is available as an extension in Visual Studio Code, Visual Studio, Vim, Neovim, the JetBrains suite of IDEs, and Azure Data Studio. For more information on using GitHub Copilot in Visual Studio Code, Visual Studio, Vim, Neovim, and JetBrains, see "[Getting code suggestions in your IDE with GitHub Copilot](https://docs.github.com/en/copilot/using-github-copilot/getting-started-with-github-copilot)." For more information on using GitHub Copilot in Azure Data Studio, see [GitHub Copilot extension](https://learn.microsoft.com/en-us/sql/azure-data-studio/extensions/github-copilot-extension?view=sql-server-ver16) in Microsoft Learn.

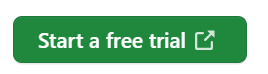


GitHub Copilot provides coding suggestions as you type in your editor. You can also ask Copilot coding-related questions, such as how best to code something, how to fix a bug, or how someone else's code works. For full details of what Copilot can do, see "[About GitHub Copilot](https://docs.github.com/en/copilot/about-github-copilot)."

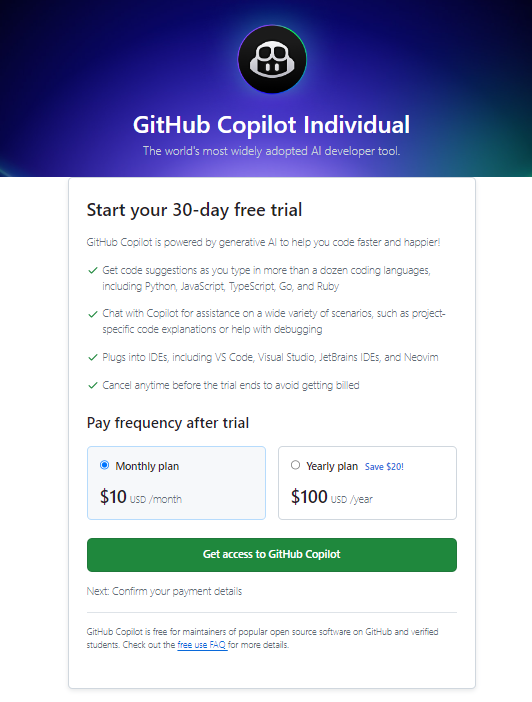
Instructions for using Copilot differ depending on the editor you use. This version of the quickstart is for JetBrains IDEs. Click the tabs above to see instructions for other editors.

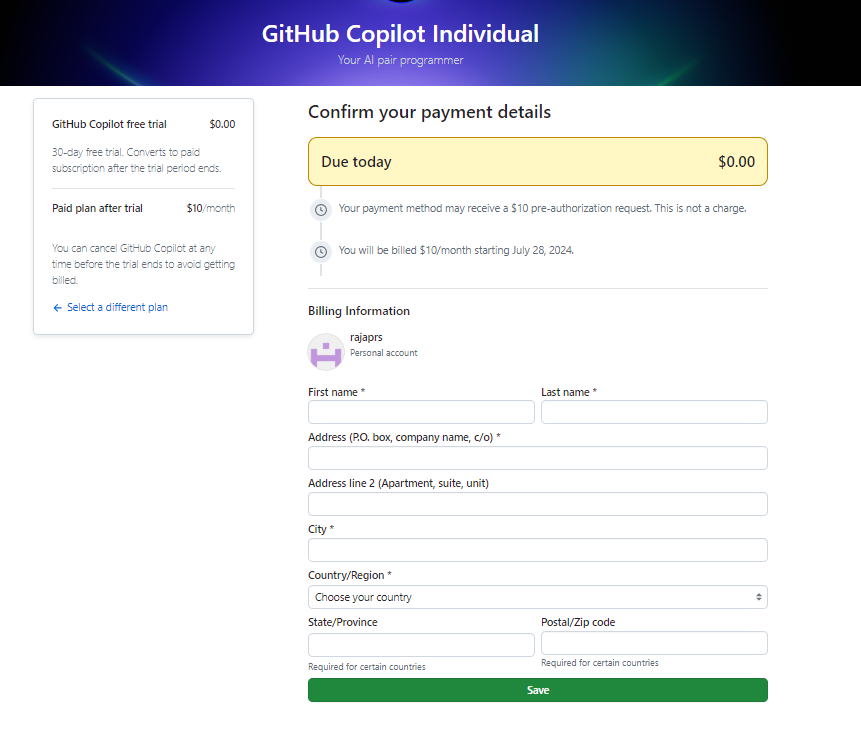
[**Sign up for GitHub Copilot**](https://docs.github.com/en/copilot/quickstart#sign-up-for-github-copilot-2)

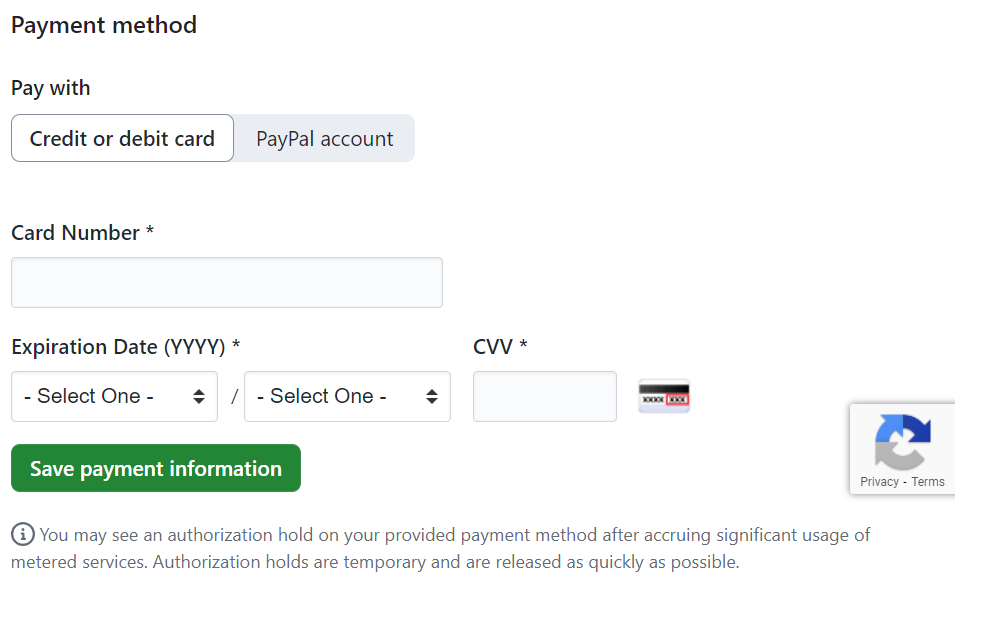
If you don't already have access to GitHub Copilot, you can set up a free trial or subscription for GitHub Copilot Individual on your personal GitHub account. For more information, see "[About GitHub Copilot Individual](https://docs.github.com/en/copilot/copilot-individual/about-github-copilot-individual)."

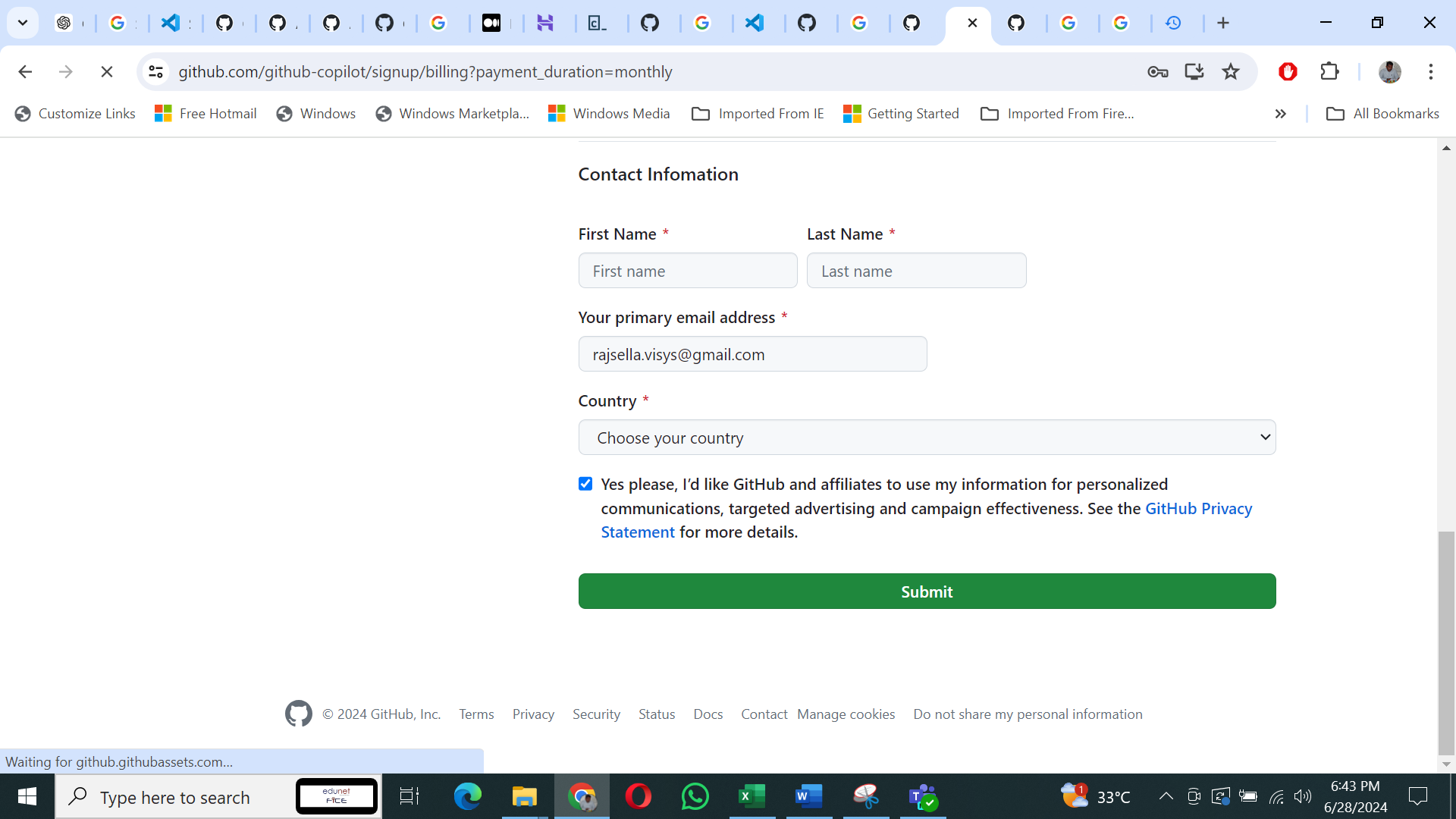


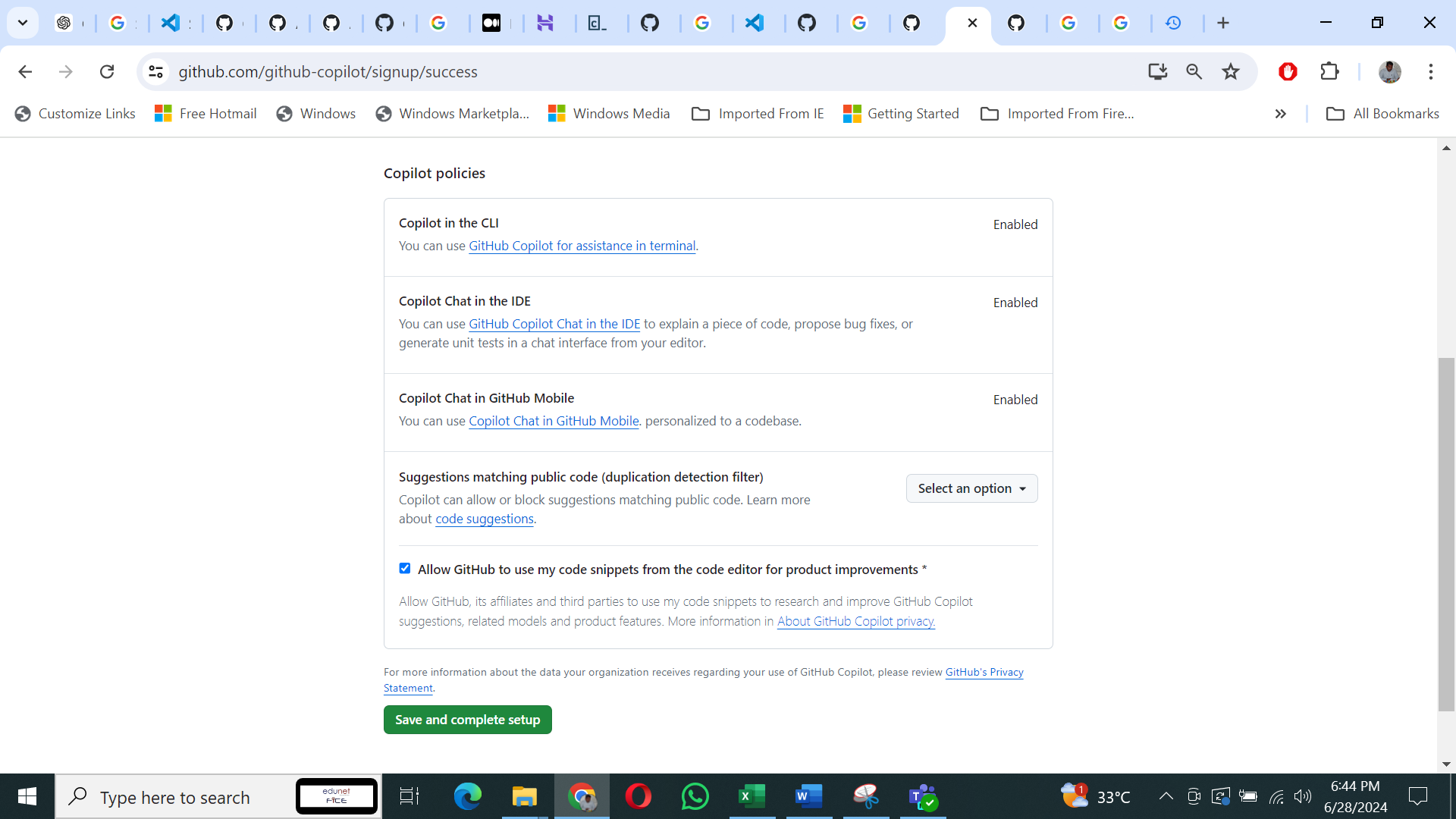
If you don't already have access to GitHub Copilot, you can set up a free trial or subscription for GitHub Copilot Individual on your personal GitHub account. For more information, see "[About GitHub Copilot Individual](https://docs.github.com/en/copilot/copilot-individual/about-github-copilot-individual)."

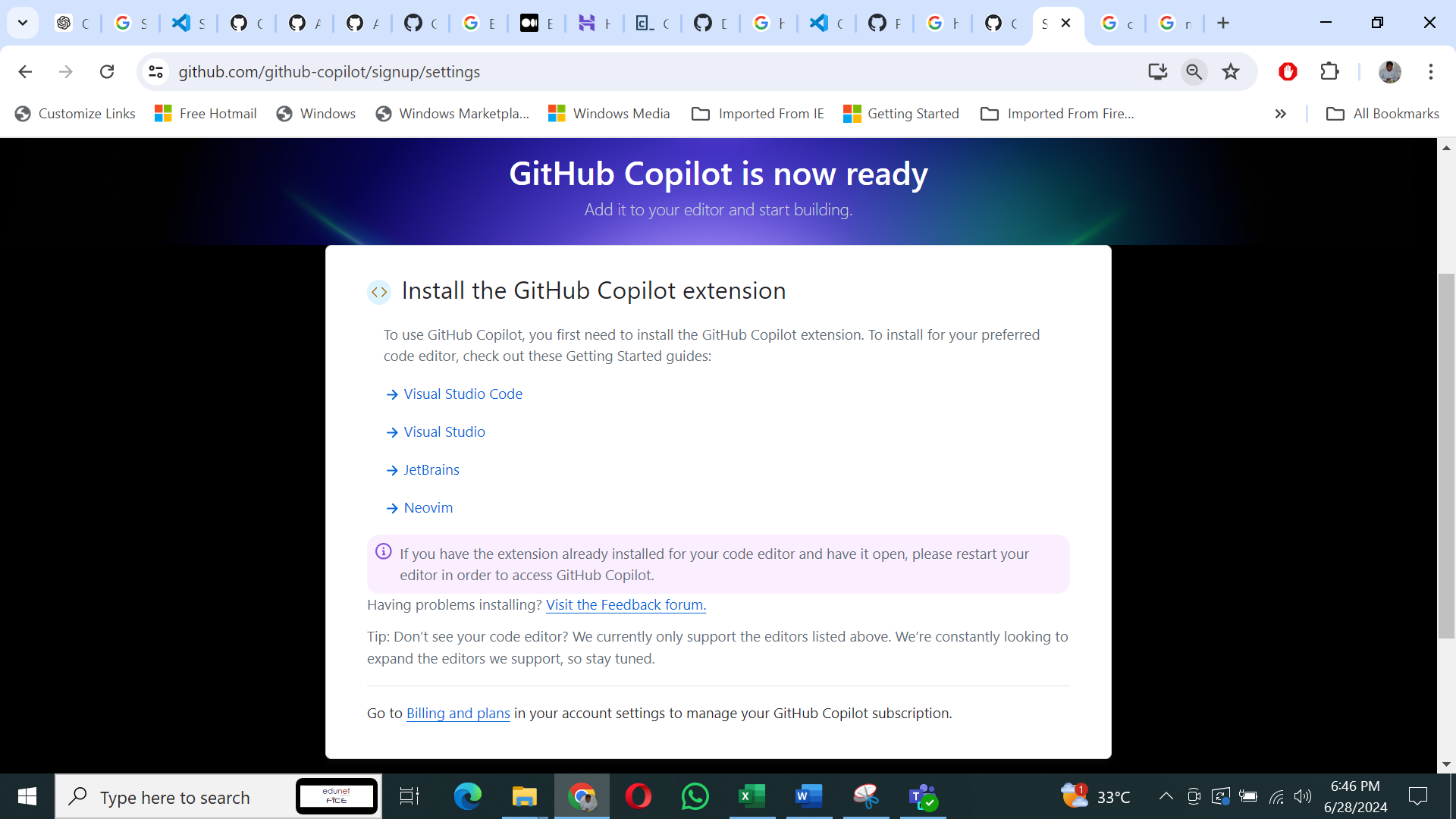












## [Prerequisites](https://docs.github.com/en/copilot/quickstart#prerequisites-2)

* **A compatible JetBrains IDE**. Copilot is supported in a large number of JetBrains IDEs. For a full list, see "[Asking GitHub Copilot questions in your IDE](https://docs.github.com/en/copilot/github-copilot-chat/copilot-chat-in-ides/using-github-copilot-chat-in-your-ide?tool=jetbrains)."
* **GitHub Copilot plugin**. See the [GitHub Copilot plugin](https://plugins.jetbrains.com/plugin/17718-github-copilot) in the JetBrains Marketplace. For installation instructions, see "[Installing the GitHub Copilot extension in your environment](https://docs.github.com/en/copilot/configuring-github-copilot/installing-the-github-copilot-extension-in-your-environment)."
* **Log in to GitHub in your JetBrains IDE**. For authentication instructions, see "[Installing the GitHub Copilot extension in your environment](https://docs.github.com/en/copilot/configuring-github-copilot/installing-the-github-copilot-extension-in-your-environment?tool=jetbrains#installing-the-github-copilot-plugin-in-your-jetbrains-ide)."

# Install a Visual Studio Code Extension

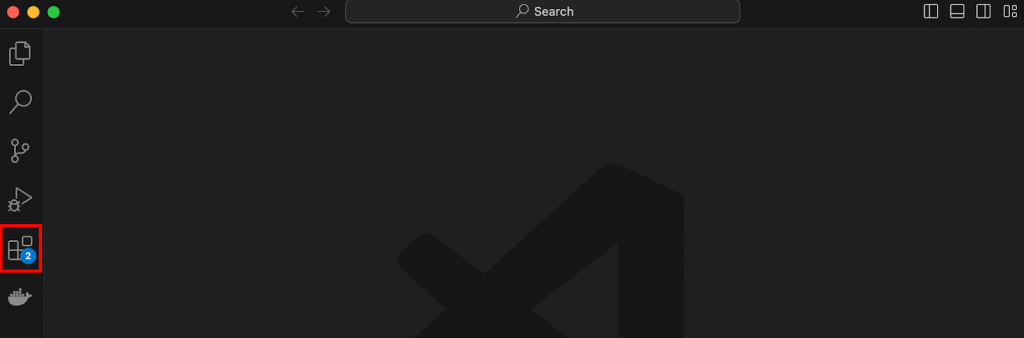
Installing GitHub Copilot is a straightforward process, regardless of whether you use Visual Studio Code, Visual Studio, Vim, Neovim, or JetBrains integrated development environment (IDE) suite. You also need a GitHub account. In this example, we will be installing the GitHub Copilot extension for Visual Studio Code:

1. **Open Visual Studio Code**

If you haven’t installed Visual Studio Code, you can [download it](https://code.visualstudio.com/download) from the official website.

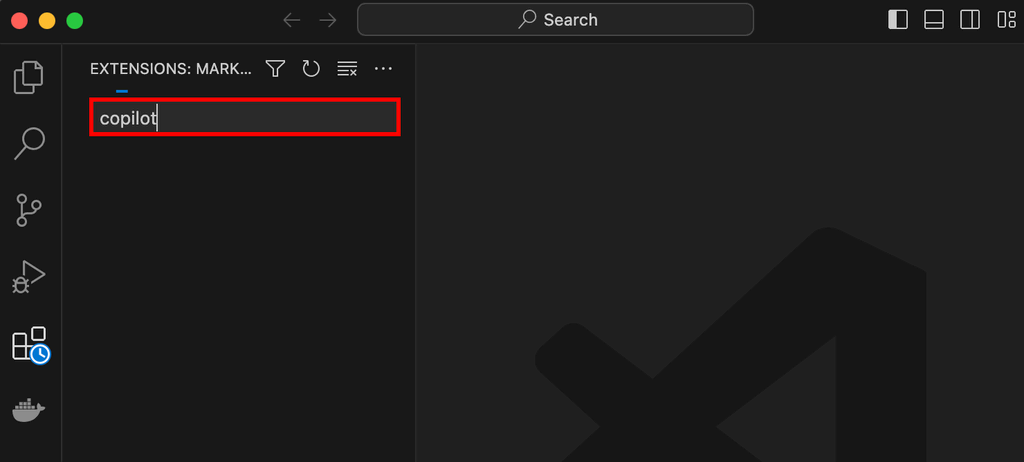
1. **Open the Extensions View**

Click on the Extensions icon in the sidebar on the left-hand side of the editor (or use the shortcut Ctrl+Shift+X on Windows/Linux or Cmd+Shift+X on macOS) to open the Extensions view.



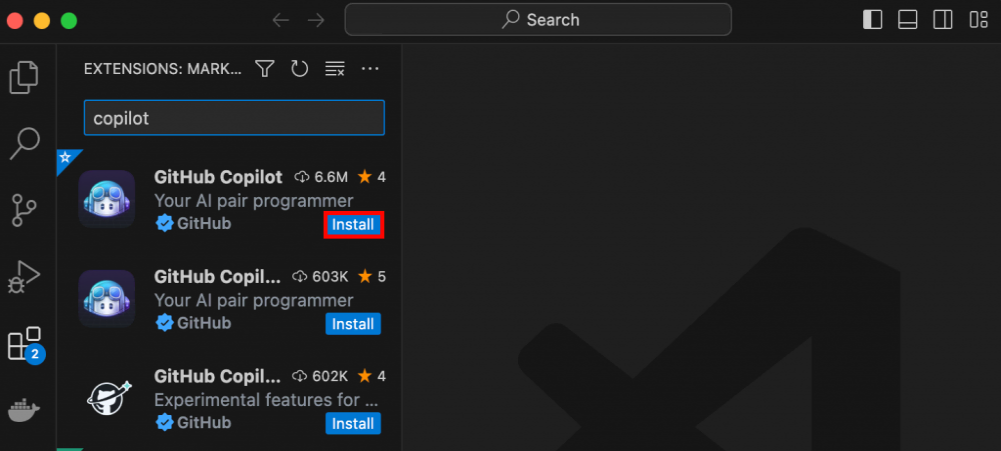
1. **Search for “GitHub Copilot”**

In the Extensions view, type “GitHub Copilot” into the search bar. From the search results, find the “GitHub Copilot” extension developed by GitHub, and click on it.



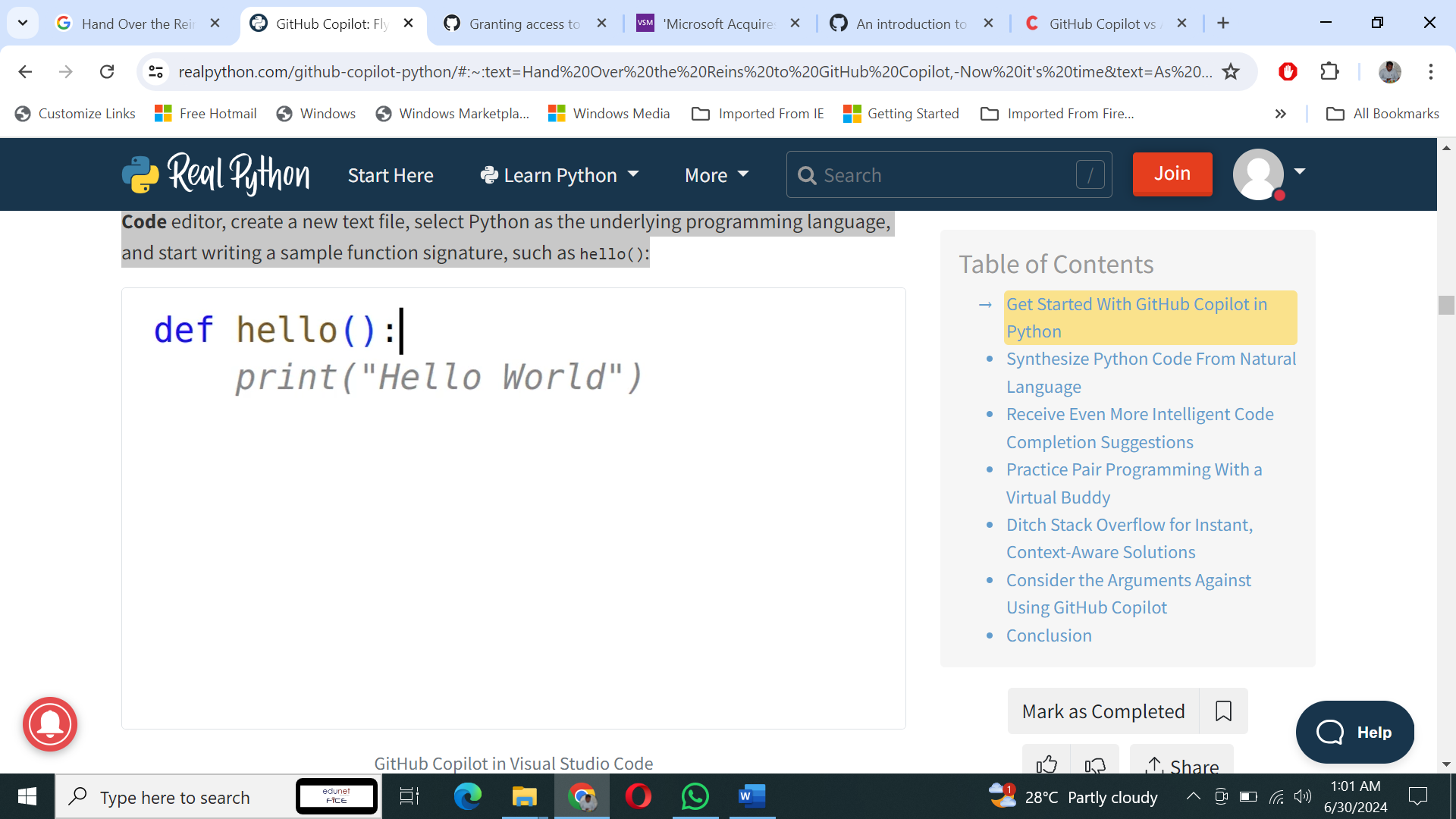
1. **Install GitHub Copilot**

On the extension page, click the “Install” button to start the installation process.



# Hand Over the Reins to GitHub Copilot

Now it’s time to make sure GitHub Copilot is working as expected in either Visual Studio Code or PyCharm. To check if GitHub Copilot is working as expected in your **Visual Studio Code** editor, create a new text file, select Python as the underlying programming language, and start writing a sample [function signature](https://en.wikipedia.org/wiki/Type_signature), such as hello():



As soon as you type the colon (:) at the end of the first line to introduce a new code block, GitHub Copilot fills in the suggested function body for you. Until you either accept it by hitting Tab or reject it with Esc, it’ll show up in gray font. The suggested code calls the [print()](https://realpython.com/python-print/) function to display the Hello World text on the screen in this case. While that wasn’t spectacular, it confirms that GitHub Copilot is indeed working correctly.

# Use a Python Comment to Describe the Problem

Although influential figures in the programming world like [Robert C. Martin](https://en.wikipedia.org/wiki/Robert_C._Martin) consider [code comments](https://realpython.com/python-comments-guide/) to be an anti-pattern, comments can sometimes be helpful in explaining *why* a certain piece of code looks the way it does. You typically write comments for your future self or your teammates working on the same codebase.

When you add GitHub Copilot into the mix, then it becomes yet another target audience who can read your code comments. Consider the following **single-line comment** in Python, which describes the classic [Hello, World!](https://en.wikipedia.org/wiki/%22Hello,_World!%22_program) program:

# Print "Hello, World!"

After typing that comment into your code editor, you’ll notice that GitHub Copilot doesn’t automatically pick it up. When you choose to communicate with it through comments, you must open the GitHub Copilot side panel or tab to see the suggestions. Alternatively, you may start typing a bit of code to have it auto-completed. Either way, writing the comment from above should present you with the following Python code:

print("Hello, World!")

That’s almost the same suggestion that you got when you verified the Visual Studio Code extension by writing the hello() function stub. However, this time, you get a slightly different output. GitHub Copilot understands that you wish to treat the quoted fragment of your comment as literal text rather than an instruction.