Python Level 3

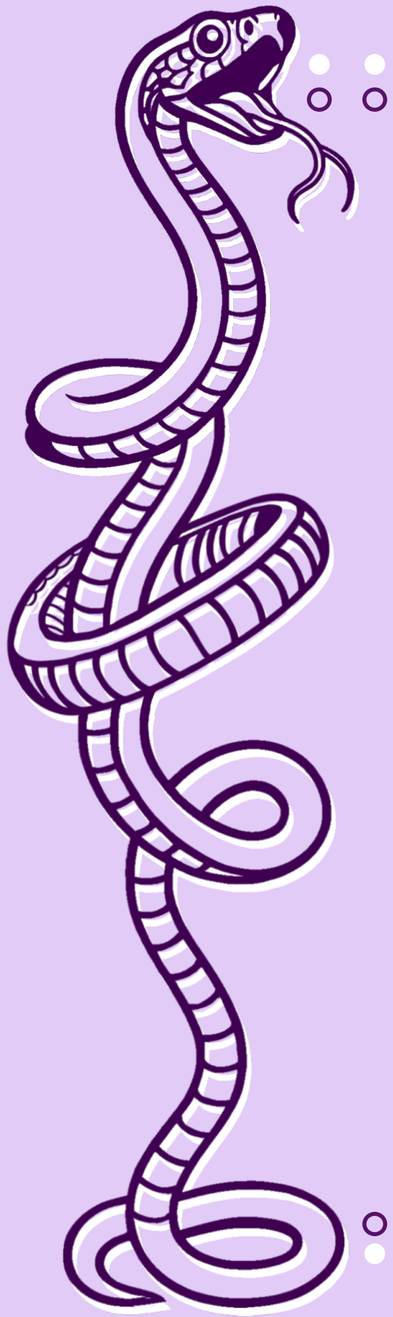
Advanced Course
Class 3

Catarina Sousa Santos

October 14th, 2025

AGENDA

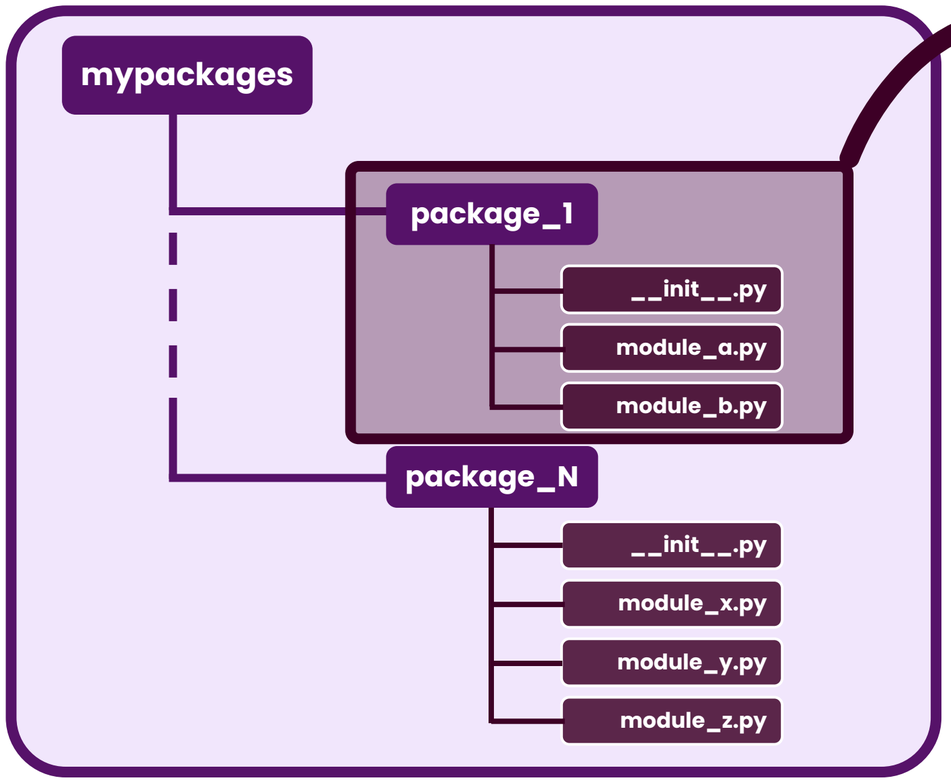
1. **Working with Python Modules**
    a. What is a Python Module?
    b. How to install Python Modules?
    c. Using Python Modules
    d. Building Python Modules
2. **Version Control with Git and GitHub**
    a. Why version control matters
    b. Core concepts
    c. GitHub basics
    d. Essential commands
    e. Creating your repository
3. **Introduction to Streamlit**
    a. What is Streamlit and why use it?
    b. Installing and launching a Streamlit app
    c. Creating basic componentes
    d. Connecting Streamlit with data
    e. Deploying Streamlit apps (intro)



PYTHON MODULES

PYTHON MODULES

A Python **module** is a **file** containing Python **definitions** and **statements**. A module can define **functions**, **classes**, and **variables**. A module can also include **runnable code**. Grouping related code into a module **makes the code easier to understand and use**. It also makes the code logically organized.



package_1

__init__.py

module_a.py

```
# Classes
class MyClassA:
    pass
class MyClassB:
    pass

# Functions
def myfunction_A():
    pass
def myfunction_B():
    pass
```

module_b.py

```
(...)
```

PYTHON MODULES

BUILT-IN MODULES

Python comes with a large **Standard Library**, that is, a collection of **ready-made modules** that you can use without installing anything.

module_examples.py

```
import datetime
import math
import random
import os


print(datetime.datetime.now()) # prints current date and time
print(math.pi, math.sqrt(16))  # 3.141592653589793 4.0
print(random.choice(['Python', 'R', 'SQL']))
print(os.getcwd())  # current directory
```

PYTHON MODULES

CREATING CUSTOM MODULES

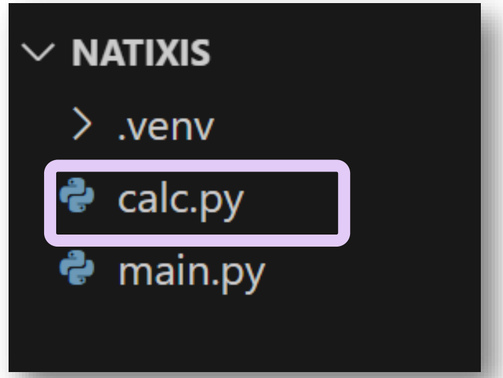
Let's create a simple **calc.py** in which we define two functions, one **add** and another **subtract**.

calc.py

```python
class Calc:
    """A simple calculator class."""

    def add(self, a, b):
        """Return the sum of two numbers."""
        return a + b

    def subtract(self, a, b):
        """Return the difference between two numbers."""
        return a - b
```



Good practice: include **docstrings** (triple quotes) to describe what each method does.

PYTHON MODULES

IMPORTING MODULES

We can **import** the functions and classes defined in a **module** to another module using the **import** statement in some other Python source file. When the interpreter encounters an import statement, it imports the module if the module is present in the **search path**.

A **search path** is a list of directories that the interpreter searches for importing a module.

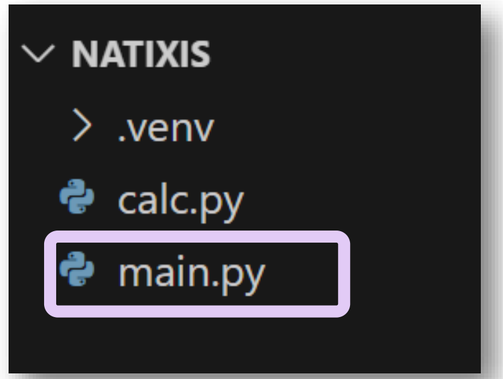
main.py

```
import calc

# Create an instance of Calc
calculator = Calc()

# Use its methods
result_add = calculator.add(10, 5)
result_sub = calculator.subtract(10, 5)

print(f"Addition: {result_add}")
Addition: 15
print(f"Subtraction: {result_sub}")
Subtraction: 5
```



IMPORTING CUSTOM MODULES

IMPORT FROM STATEMENT

Python's **from** statement lets you **import specific attributes** from a module **without importing the module as a whole.**

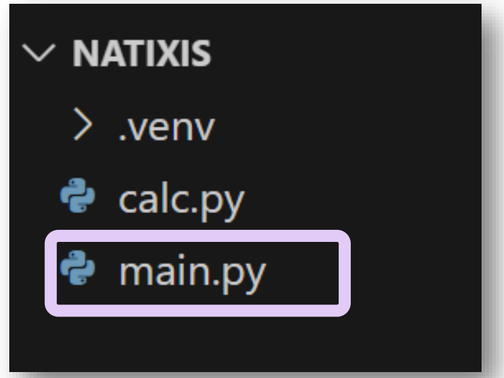
main.py

```python
from calc import Calc

# Create an instance of Calc
calculator = Calc()

# Use its methods
result_add = calculator.add(10, 5)
result_sub = calculator.subtract(10, 5)

print(f"Addition: {result_add}")
Addition: 15
print(f"Subtraction: {result_sub}")
Subtraction: 5
```



PYTHON MODULES

RENAMING MODULES

Module names can be long or long, used often, or might conflict with another name in your program.

The ***as*** keyword in Python allows providing an **alternative name** to the module, known as an **alias**.

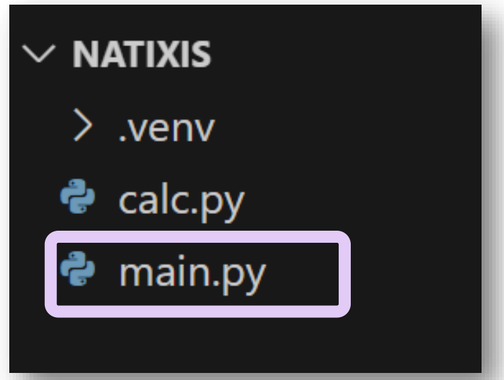
main.py

```python
import calc as clc

# Create an instance of Calc
calculator = clc.Calc()

# Use its methods
result_add = calculator.add(10, 5)
result_sub = calculator.subtract(10, 5)

print(f"Addition: {result_add}")
Addition: 15
print(f"Subtraction: {result_sub}")
Subtraction: 5
```



IMPORTING ALL NAMES

IMPORTING ALL NAMES

The * wildcard used with the **from import** statement is used **to import all the names** from a module to a current namespace.

While this works, it is **not considered good practice** in most situations.

main.py

```
from statistics import *
from numpy import *

print(mean([1,2,3])) # which mean is this? statistics or numpy?
```

When using from module import *, Python imports in order: **later imports silently overwrite earlier names** with the same identifier.

REASONS TO AVOID *

1. **Namespace pollution**: It imports **every** variable, function, and class from the module into your current namespace.
2. **Reduced readability**: it becomes unclear which module a function or variable came from. This makes **maintenance and debugging harder**, especially in large projects.
3. **Unexpected behavior**: If a module changes in future versions (e.g., adds new functions), **your code may break or behave differently** without you realizing it.
4. **No auto-completion or static analysis**: Modern **editors** and **IDEs cannot easily provide code hints** or autocomplete when everything is imported blindly.

PYTHON PACKAGES

PYTHON PACKAGES

A Python **package** is way to organize **related modules** into a **single directory structure**, making projects **easier to maintain**. A package is simply a **folder** containing:

- One or more **.py** files (modules), and
- A special file called **__init__.py**

ROLE OF __init__

The **__init__.py** file tells Python that the directory **should be treated as a package**. Without it, Python will not recognize the folder as importable*. It can:

- Be **empty**
- Contain **initialization code** or **imports** for easier access.

*in versions before 3.3, though keeping it is still best practice.

package_1

__init__.py

```
```

module_a.py

```
# Classes
class MyClassA:
    pass
class MyClassB:
    pass

# Functions
def myfunction_A():
    pass
def myfunction_B():
    pass
```

module_b.py

```
(...)
```

VERSION CONTROL

VERSION CONTROL

WHY VERSION CONTROL MATTERS

Version control (also known as **source control** or **revision control**) ensures **traceability**, **collaboration**, and **stability** in software projects, especially when working with others.

Version control is essential to:

- Keep track of **what changed**, **when**, and **by whom**.
- **Revert** (or **rollback**) a previous version if something breaks.
- Collaborate safely **without overwriting** each other's work.
- Maintain a **complete history** of the project's evolution.
- **Identify issues quickly**.

Git is the most widely used distributed version control system today, powering powers platforms like **GitHub**, **GitLab**, and **Bitbucket**.

VERSION CONTROL

CORE CONCEPTS

Version control is built around a few fundamental concepts that define how changes to code are tracked, managed, and shared:

REPOSITORY	The **main project folder** tracked by version control, containing all files, subfolders, and the complete history of changes. Repositories can be **local** (on a computer) or **remote** (e.g., on GitHub)
COMMIT	Represents a **snapshot** of the project at a given moment. Each commit includes, **what** changed, **who** made the change, and **when** the change was made.
BRANCH	An **independent** line of development. Allows experimenting or developing new features without affecting the main codebase. The **main** or **master** branch contains the **stable** version of the code.
MERGE	**Combines** the work from one branch to another. When a feature is ready, it's **merged back** into the main branch. **Conflicts may occur** if the same part of the code was changed differently.
PULL REQUEST	A formal way to **propose changes** before merging. It allows teammates to **review**, **comment**, and **approve** code, ensuring quality.

VERSION CONTROL

GIT

Git is a **distributed version control system** (**DVCS**) created by Linus Torvalds in 2005 to manage the development of the Linux kernel.

Git has since become the **industry standard for tracking and coordinating code changes** across teams and projects. It allows developers to:

- **Record snapshots** of a project's state through **commits**.
- **Work concurrently** on different **branches** without interfering with each other.
- **Merge** changes back into the main codebase once **tested** and **approved**.
- **Revert** to previous versions when errors occur.

Each developer's local repository is a **complete copy of the project** including its full history, enabling work even without an internet connection.

Git is the **engine** behind version control**.** It keeps **every version** of the code **safe**, **synchronized**, and **recoverable.**

VERSION CONTROL

GITHUB

GitHub is a **cloud-based platform** built around Git, designed to **host repositories**. It acts as the **remote counterpart** to a local Git repository, allowing teams to **synchronize their work**, **review code**, and **collaborate and manage projects** from anywhere.

GitHub delivers tools beyond basic version control, providing:

- **Remote hosting** of Git repositories.
- **Pull requests** (**PRs**) for reviewing and merging code changes.
- **Issues** and **discussions** to track bugs, tasks, and feedback.
- **Actions** for continuous integration and automation (CI/CD).
- **Wikis** and documentation to support project transparency.
- **Permission**, **issue**, and **workflow management**.
- Easy integration with tools like **Streamlit Cloud**, **Docker Hub**, and **CI/CD pipelines**.

GitHub is the workshop where developers collaborate, review, and share their work.

VERSION CONTROL

ESSENTIAL GIT COMMANDS

Git operationalizes version control through a **set of fundamental commands**. These include, but are not limited to:

Command	Purpose
`git init`	Initialize a new local repository and begin version tracking.
`git status`	Display the state of the working directory and staging area.
`git add`	Stage changes to be included in the next commit.
`git commit`	Record a snapshot of staged changes to the repository history.
`git diff`	Show differences between file versions, commits, or branches.
`git branch`	List, create, or delete branches within the repository.
`git checkout / git switch`	Move between branches or restore files to a previous state.
`git merge`*	Integrate changes from one branch into another.
`git rebase`*	Reapply commits from one branch onto another, creating a linear project history.
`git remote add origin <url>`	Link the local repository to a remote (commonly named origin).
`git push`	Upload local commits to a remote repository (e.g., GitHub).
`git pull`	Fetch and integrate changes from the remote repository into the local branch.

*__git merge__ and __git rebase__ achieve similar goals, but through different strategies: ***merge*** preserves the **full commit history**, while ***rebase*** rewrites it **linearly**.

VERSION CONTROL

INSTALLING GIT

Go to https://git-scm.com/downloads Choose the installer for your operating system:

- **Windows**: .exe installer (includes Git Bash)
- **MacOS**: .dmg installer or install via Homebrew
- **Linux**: use a package manager.

CHECKING INSTALLATION

TERMINAL

```
> git --version
git version 2.x.x
```

VERSION CONTROL

CONFIGURING GIT

Before creating your first repository, set your **user identity**.

TERMINAL

```
> git config --global user.name "Your Name"
> git config --global user.email "your@email.com"
```

Optional: set your **default branch**'s name to **main** instead of **master** (modern convention)

TERMINAL

```
> git config --global init.defaultBranch main
```

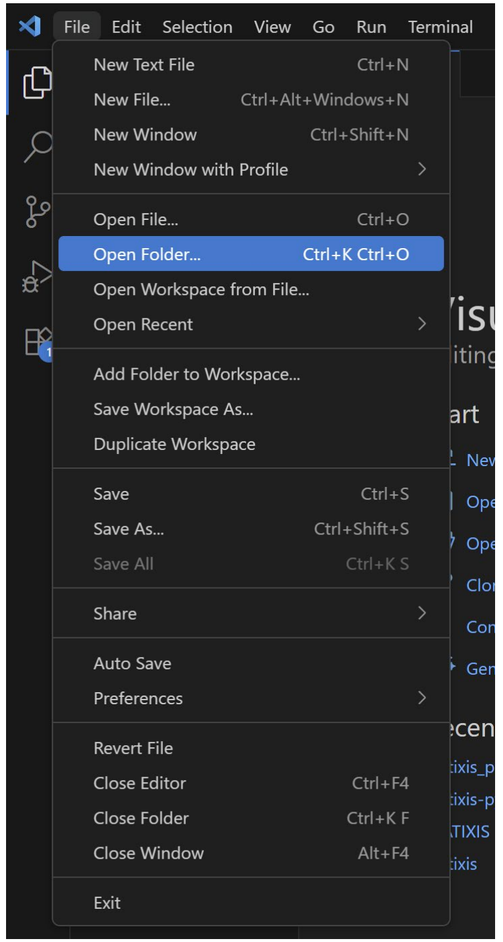
VERSION CONTROL

CREATING A GIT REPOSITORY

1 Open the Project Folder

In VS Code, go to File → Open Folder... and select your project directory, or create a new directory using **mkdir** and change directory (**cd**) to it.

```
TERMINAL
> mkdir my_first_repo
> cd my_first_repo
```



VERSION CONTROL

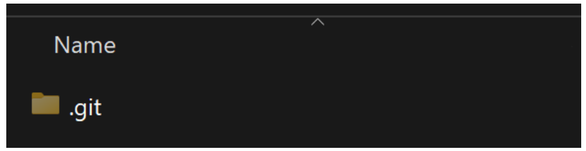
CREATING A GIT REPOSITORY

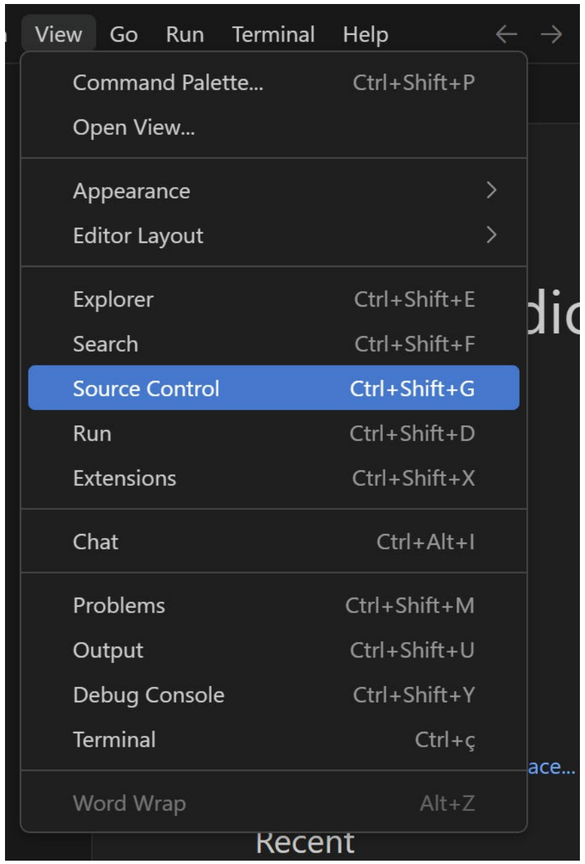
② Initialize Git

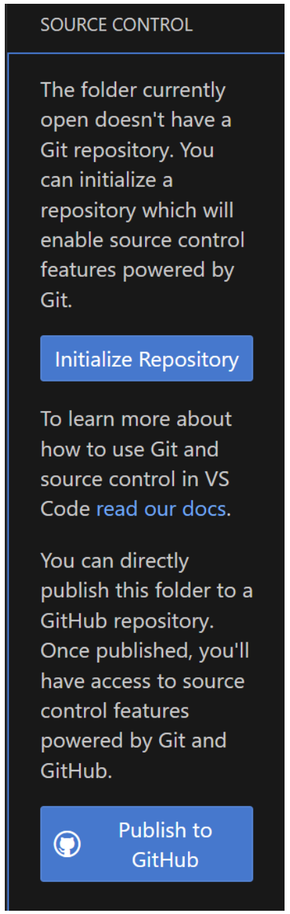
Click **Initialize Repository** in the **Source Control** panel or open the VS Code terminal and run:

```
TERMINAL
> git init
Initialized empty Git repository in C:/path_to_folder/.git
```

A hidden **.git** directory is created, making the project now **tracked by Git**.







VERSION CONTROL

CREATING A GIT REPOSITORY

3 Create a README file

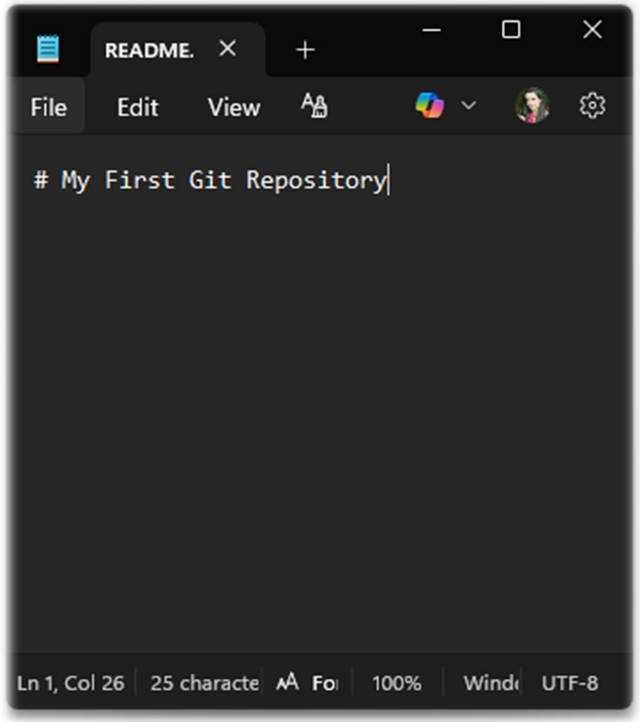
Before making your first commit, create at least one file in your project folder so Git has something to track.

You can do it via **VS Code**:

1. Click New File
2. Name it README.MD
3. Add a short line, such as: # My First Git Repository

Or using the terminal:

```
TERMINAL
> notepad README.md
[will create and open the file]
```



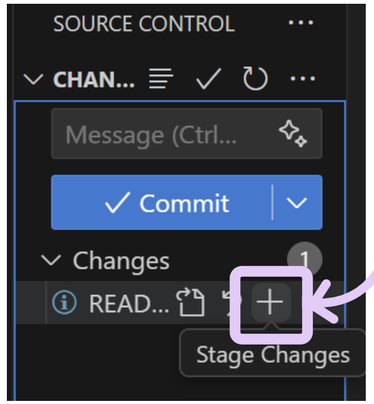
VERSION CONTROL

CREATING A GIT REPOSITORY

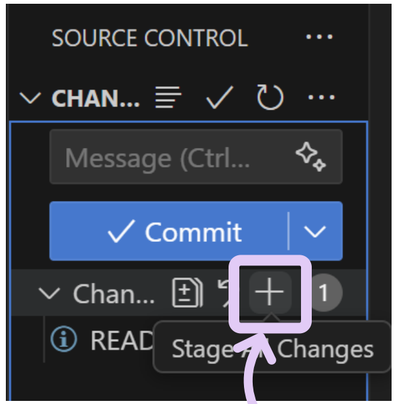
4 Stage your files

You can stage all your files at once or one by one.

In the **Source Control** panel, hover over your files and click **+** to **stage** them.

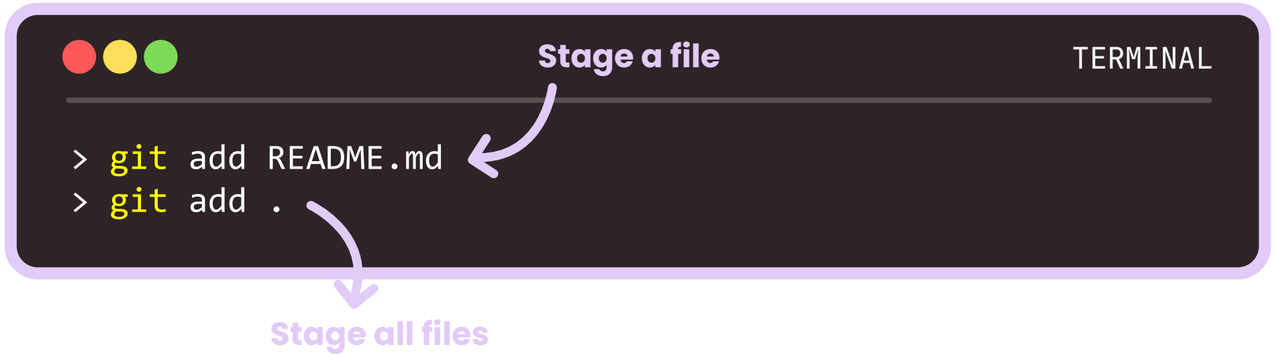


Stage a file



Stage all files

Or, using the terminal:



Stage all files

VERSION CONTROL

CREATING A GIT REPOSITORY

4 Exclude files from Git

The **.gitignore** file tells Git which files or folders **should not be tracked**. This keeps your repository clean and avoids pushing unnecessary or sensitive files to GitHub.

How to Create a .gitignore file

In VS Code:

1. In your project root, right-click → **New File** → name it **.gitignore.**
2. Add entries for files/folders you don't want tracked.
3. Save the file before running git add . or making your first commit

Or, using the terminal:

```
TERMINAL

> notepad .gitignore
```

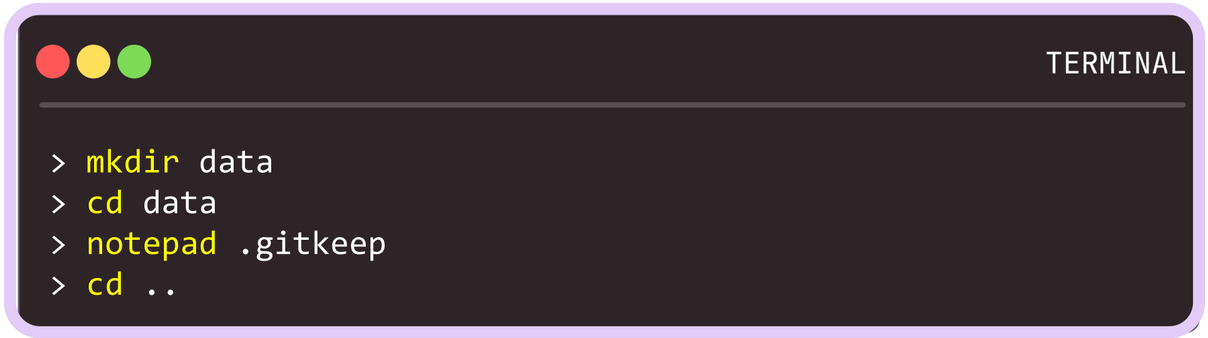
VERSION CONTROL

CREATING A GIT REPOSITORY

5 Stage empty directories

Git doesn't track **empty directories** by default. If you want to keep a folder structure in your repository, create an **empty file** inside it named **.gitkeep**.

Example (Terminal)

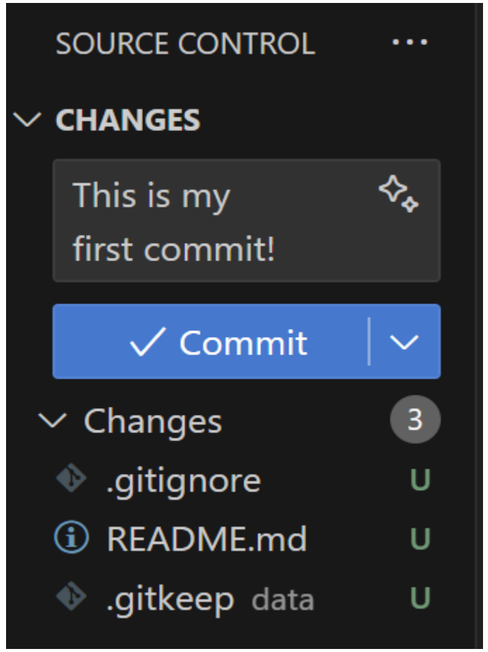


VERSION CONTROL

CREATING A GIT REPOSITORY

5 Commit your changes

In VS Code, enter a commit message (e.g. *This is my first commit!*) at the top of the panel and press **Ctrl + Enter.**



Or use the terminal:

```
TERMINAL

> git add .
> git commit -m "This is my first commit!"
```

The -m flag adds a short message describing what was done.

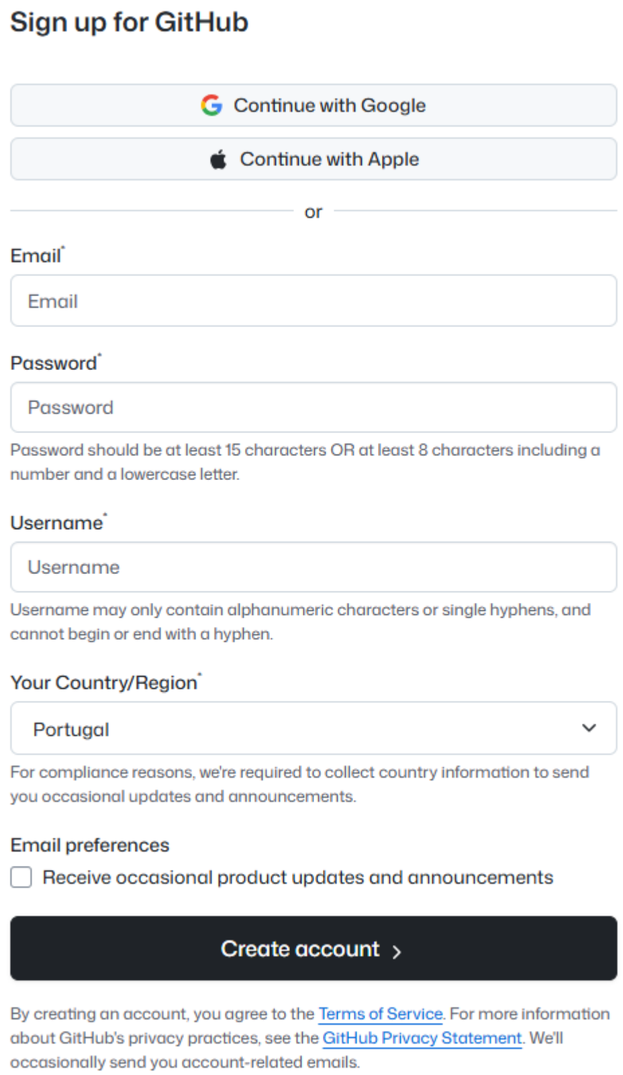
VERSION CONTROL

CREATING A GIT REPOSITORY

6 Create or Login to your GitHub Account

Go to https://github.com. If you don't have an account:

1. Click **Sign up** in the top-right corner.
2. Enter your details (email, password, and unique username)
3. Select the free plan (allows unlimited public and private repositories)
4. Verify your email to activate the account.

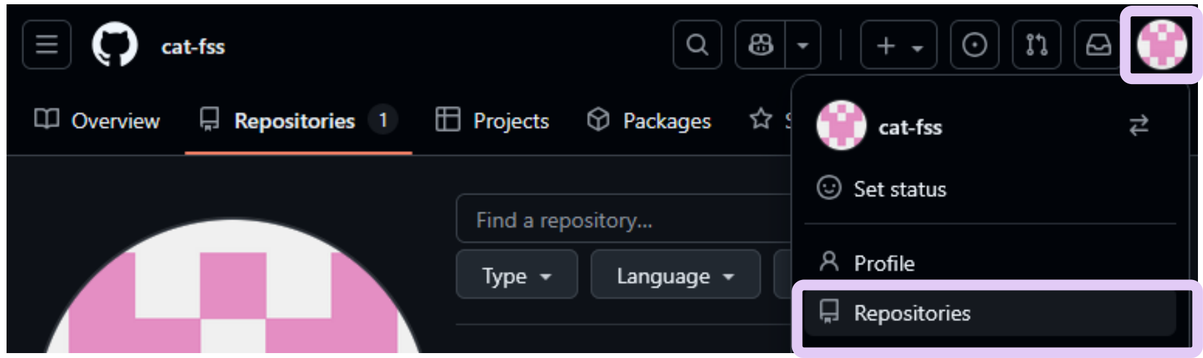


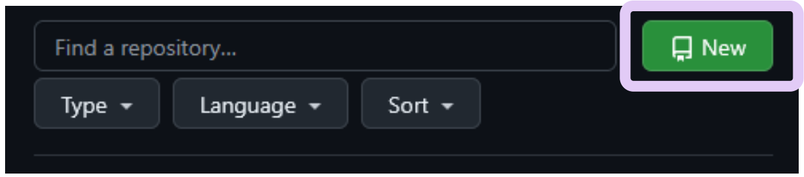
VERSION CONTROL

CREATING A GIT REPOSITORY

7 Create an empty repository on GitHub

Click on your **profile picture** in the top-right corner, then select **repositories**. Then, select **New** on the repositories page.

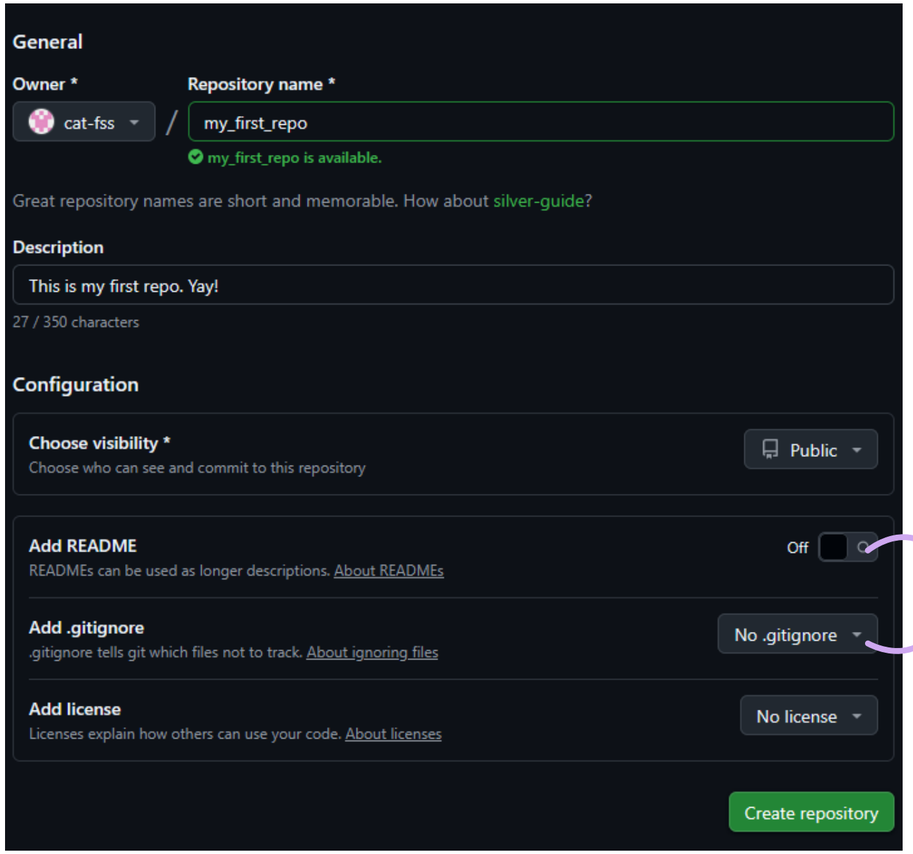




VERSION CONTROL

CREATING A GIT REPOSITORY

7 Create an empty repository on GitHub



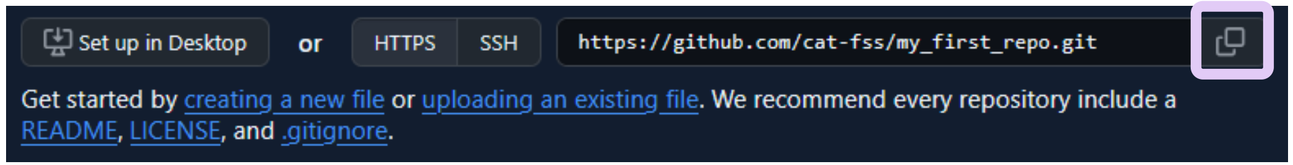
Keep these options off so you don't have conflicts with your local version!

VERSION CONTROL

CREATING A GIT REPOSITORY

8 Connect your local repository to GitHub

Copy your repository's URL.



In VS Code's terminal:

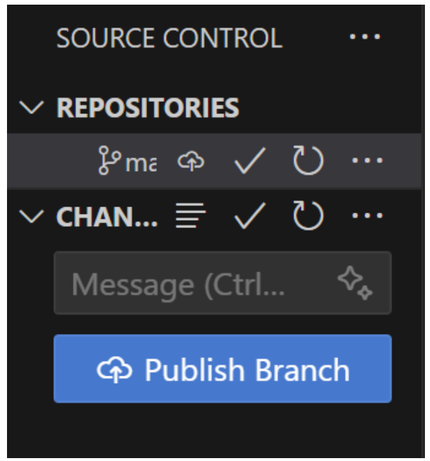
TERMINAL

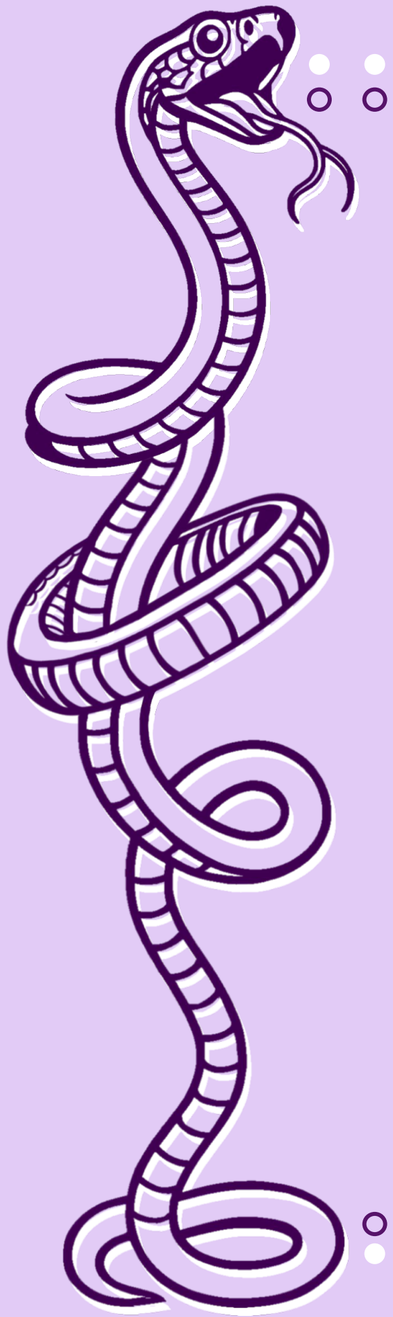
Move/rename the branch, even if the target name (main) already exists.

```
> git branch -M main
> git remote add origin https://github.com/<username>/<repo>.git
> git push -u origin main
```

Set upstream (remember that the **local branch main is linked to the remote branch main on origin)**

You can also connect directly through the **Source Control** → **Publish Branch** option





STREAMLIT INTRO

STREAMLIT

WHAT IS STREAMLIT?

Streamlit is an open-source Python framework that allows **building interactive web applications** directly from python scripts without needing any frontend development skills.

It is designed for **data scientists**, **researchers**, and **developers** who want to quickly transform data and models into interactive dashboards or tools.

The full documentation for Streamlit can be found at **https://docs.streamlit.io/develop/api-reference**.

Streamlit turns Python scripts into shareable web apps.

STREAMLIT

WHY USE STREAMLIT?

Streamlit

Feature	Benefit
Simplicity	Write apps using pure Python. No need for HTML, CSS, or JavaScript.
Integration	Works directly with Pandas, Matplotlib, Plotly, and machine learning libraries.
Real-time updates	Auto-refreshes when the script is modified, which is ideal for development.
Deployment	Easily deploy to Streamlit Cloud or other hosting services.

STREAMLIT

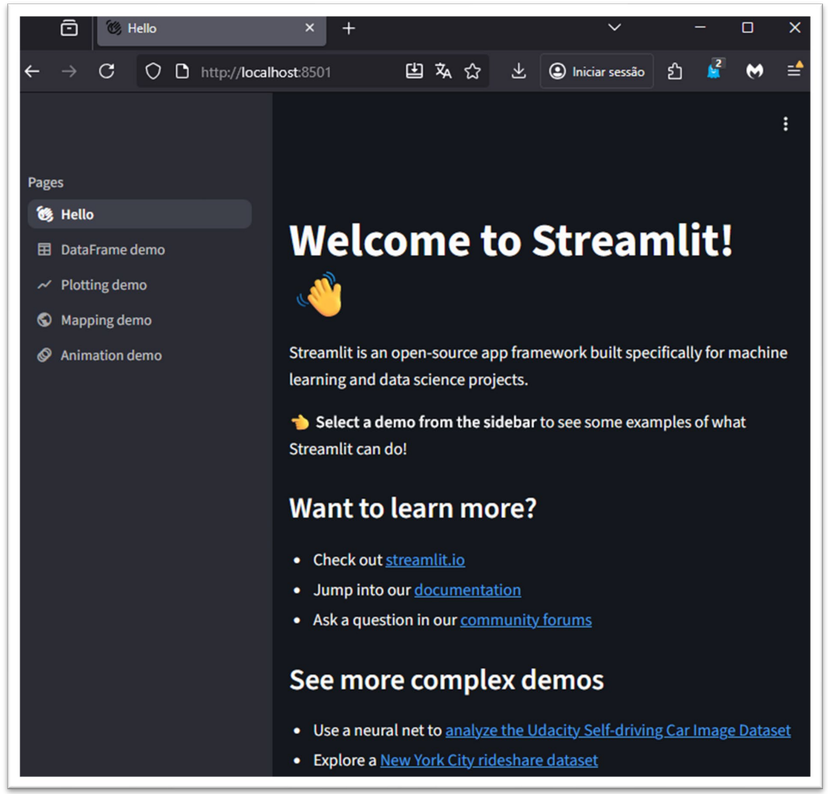
INSTALLING STREAMLIT

Streamlit is installed like any other Python library:

```
TERMINAL
> pip install streamlit
  [Successfully installed...]
```

Verify Installation:

```
main.py
> streamlit hello
```



STREAMLIT

CREATING AND RUNNING A STREAMLIT APP

1 Write a Simple Script

app.py

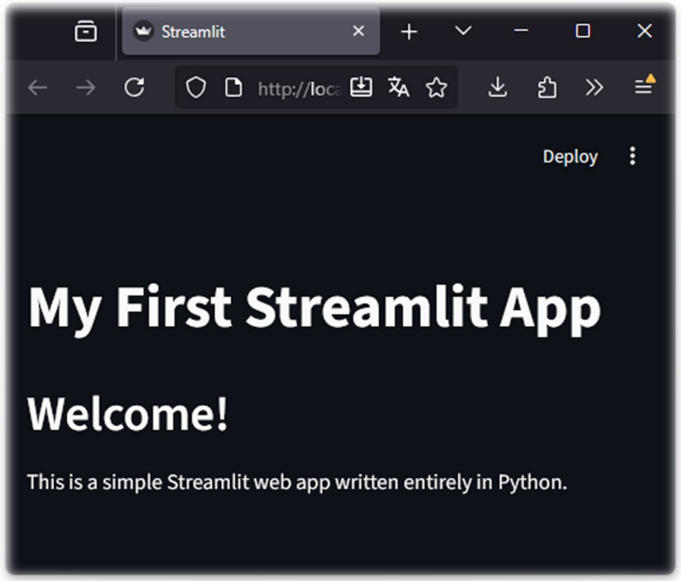
```
import streamlit as st

st.title("My First Streamlit App")
st.header("Welcome!")
st.write("This is a simple Streamlit web app written entirely in Python.")
```

2 In the terminal (not Python) execute:

app.py

```
> streamlit run app.py
```



STREAMLIT

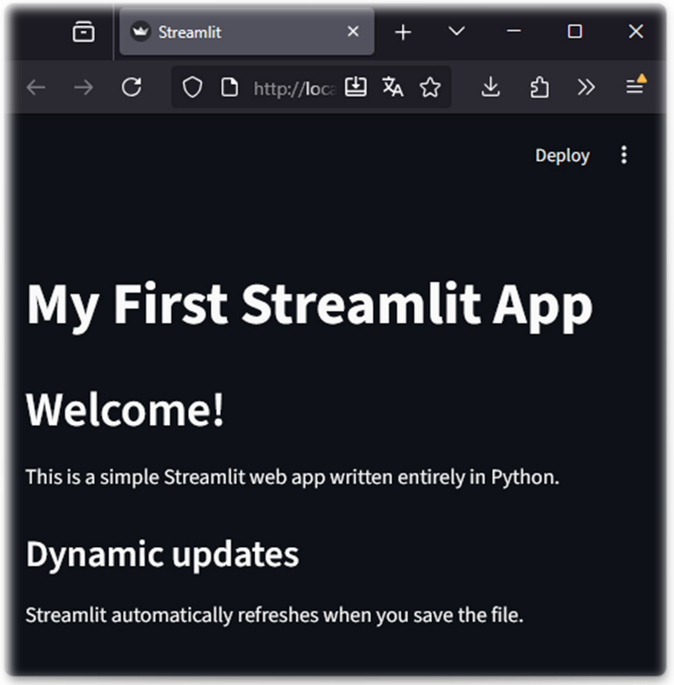
CREATING AND RUNNING A STREAMLIT APP

3 Edit and Observe

app.py

```
import streamlit as st

st.subheader("Dynamic updates")
st.write("Streamlit automatically refreshes when you save the file.")
```



4 Stop the Server

When finished, stop the app by pressing **Ctrl + C** in the terminal.

STREAMLIT

COMPONENTS OVERVIEW

Streamlit provides a wide variety of **built-in components** that allow developers to create interactive, data-driven web application. These components can be grouped by purpose:

① Text and Display

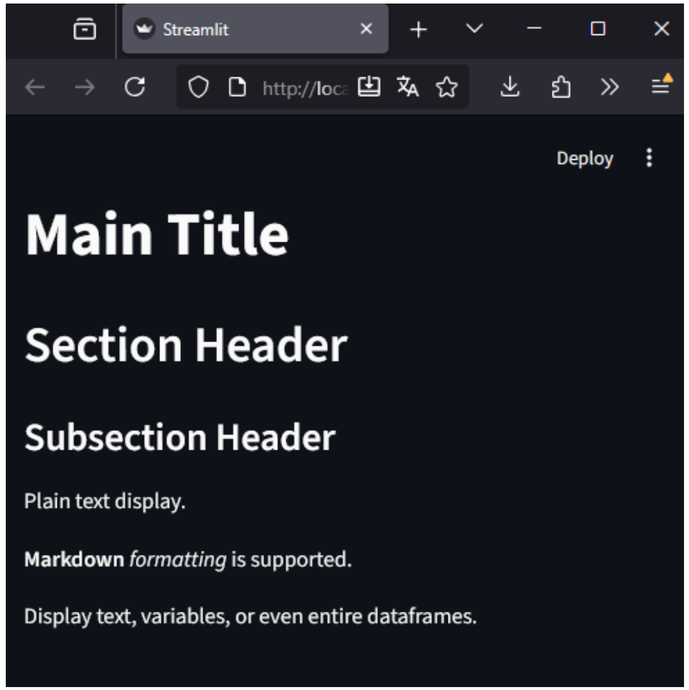
Used to display text, formatted output, or markdown.

app.py

```python
import streamlit as st

st.title("Main Title")
st.header("Section Header")
st.subheader("Subsection Header")
st.text("Plain text display.")
st.markdown("**Markdown** _formatting_ is supported.")
st.write("Display text, variables, or even entire dataframes.")
```

Automatically adapts to any type (text, numbers, dataframes, etc.)



STREAMLIT

COMPONENTS OVERVIEW

Streamlit provides a wide variety of **built-in components** that allow developers to create interactive, data-driven web applications. These components can be grouped by purpose:

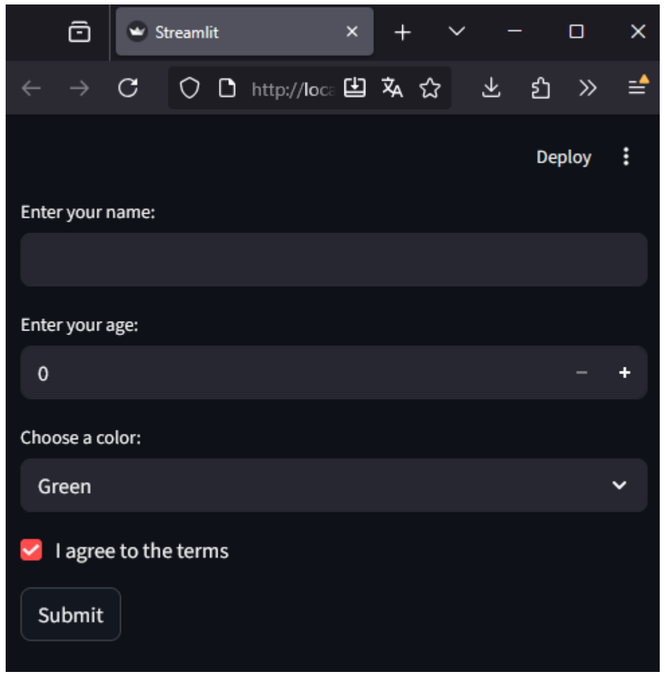
② Input

Enable interaction with the user. These widgets return values that can be stored in variables and used dynamically.

app.py

```python
import streamlit as st

name = st.text_input("Enter your name:")
age = st.number_input("Enter your age:", min_value=0, max_value=120)
color = st.selectbox("Choose a color:", ["Red", "Green", "Blue"])
agree = st.checkbox("I agree to the terms")
clicked = st.button("Submit")
```



STREAMLIT

COMPONENTS OVERVIEW

Streamlit provides a wide variety of **built-in components** that allow developers to create interactive, data-driven web applications. These components can be grouped by purpose:

② Input

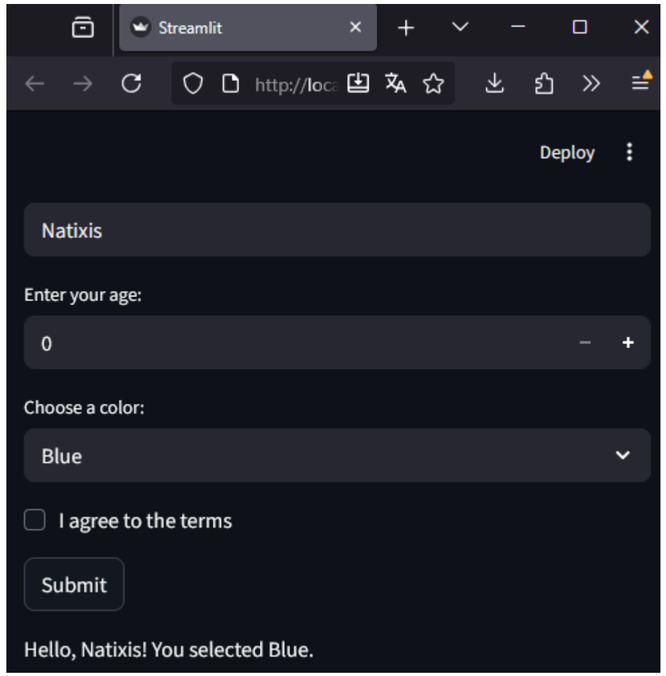
Enable interaction with the user. These widgets return values that can be stored in variables and used dynamically.

app.py

```python
import streamlit as st

name = st.text_input("Enter your name:")
age = st.number_input("Enter your age:", min_value=0, max_value=120)
color = st.selectbox("Choose a color:", ["Red", "Green", "Blue"])
agree = st.checkbox("I agree to the terms")
clicked = st.button("Submit")

if clicked:
    st.write(f"Hello, {name}! You selected {color}.")
```



STREAMLIT

COMPONENTS OVERVIEW

Streamlit provides a wide variety of **built-in components** that allow developers to create interactive, data-driven web applications. These components can be grouped by purpose:

3 Data Display

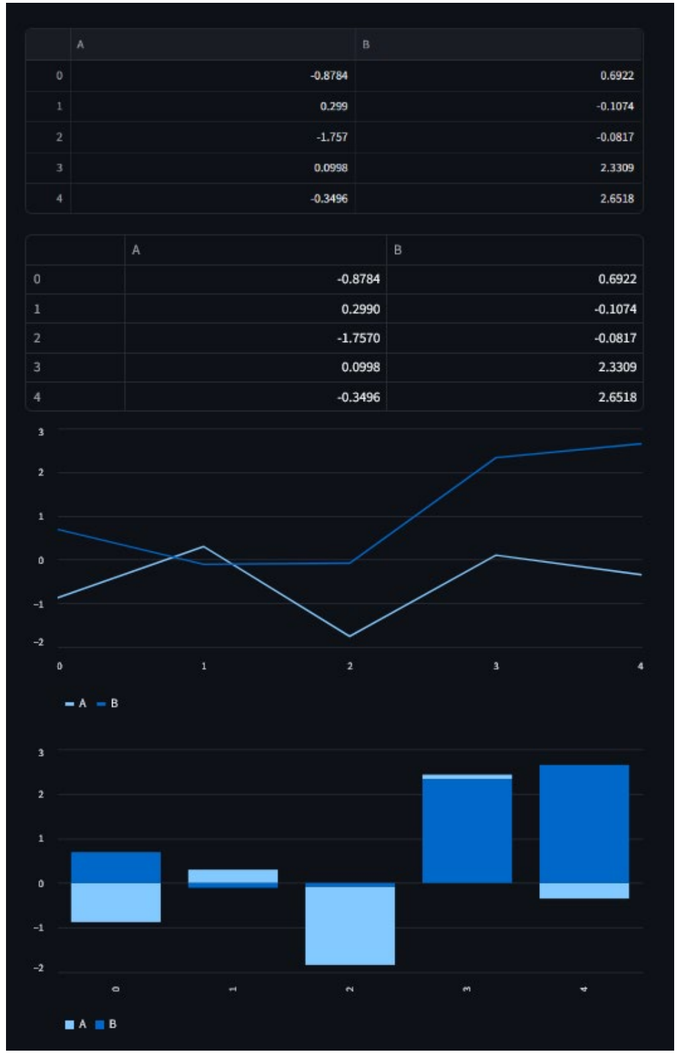
Used to integrate with Pandas, NumPy, and visualization libraries.

app.py

```python
import pandas as pd
import numpy as np
import streamlit as st

df = pd.DataFrame({
    "A": np.random.randn(5),
    "B": np.random.randn(5)
})

st.dataframe(df)          # Interactive table
st.table(df.head())       # Static table
st.line_chart(df)         # Quick visualization
st.bar_chart(df)
```



STREAMLIT

COMPONENTS OVERVIEW

Streamlit provides a wide variety of **built-in components** that allow developers to create interactive, data-driven web applications. These components can be grouped by purpose:

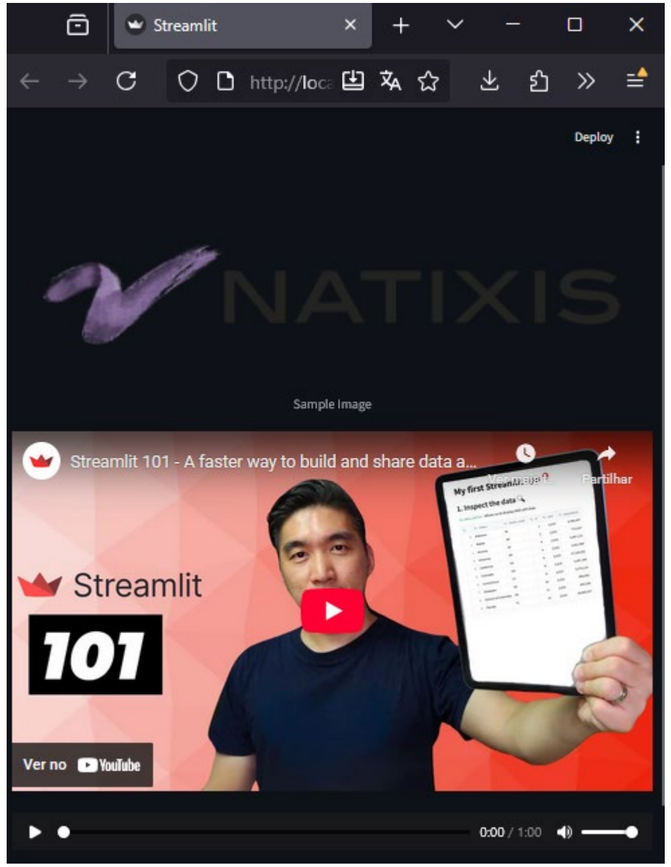
4 Media Components

Display images, videos, and audio files.

app.py

```python
import streamlit as st


st.image("example.png",
         caption="Sample Image",
         use_container_width=True)
st.video("example.mp4")
st.audio("example.mp3")
```



STREAMLIT

COMPONENTS OVERVIEW

Streamlit provides a wide variety of **built-in components** that allow developers to create interactive, data-driven web applications. These components can be grouped by purpose:

5 Layout and Organization

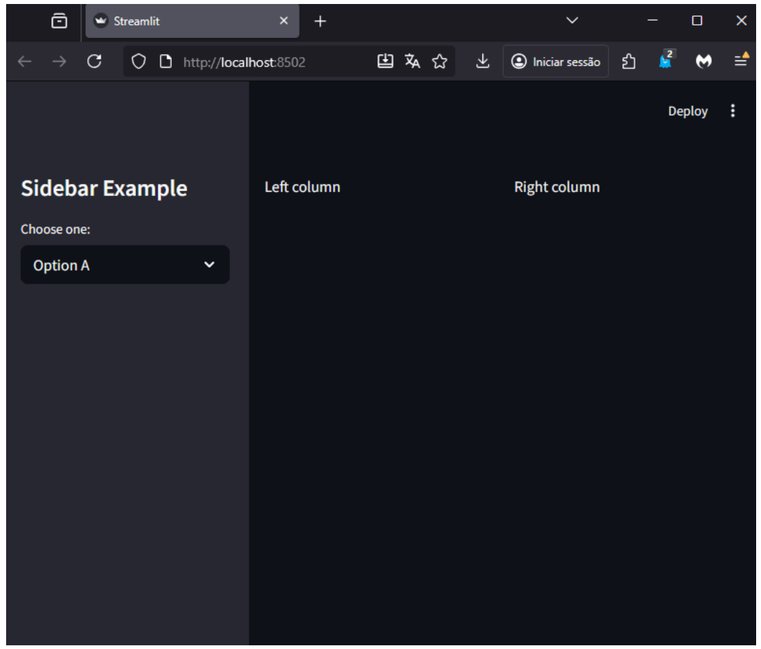
Control the structure of the app.

app.py

```python
import streamlit as st

st.sidebar.title("Sidebar Example")
option = st.sidebar.selectbox(
    "Choose one:", ["Option A", "Option B"])

col1, col2 = st.columns(2)
col1.write("Left column")
col2.write("Right column")
```



STREAMLIT

COMPONENTS OVERVIEW

Streamlit provides a wide variety of **built-in components** that allow developers to create interactive, data-driven web applications. These components can be grouped by purpose:

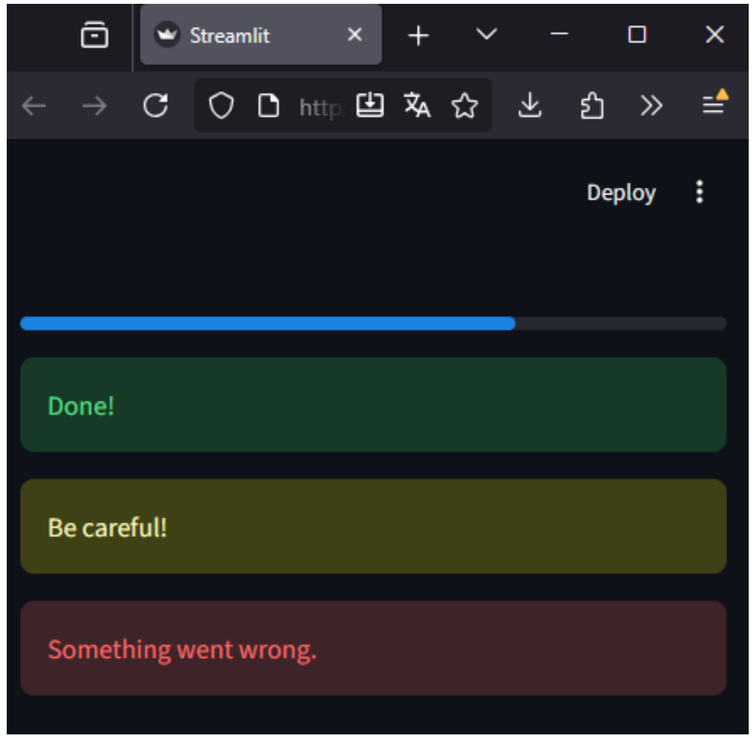
6 Status and Progress

Show progress or feedback during operations.

app.py

```python
import streamlit as st

st.progress(70)
st.spinner("Processing...")
st.success("Done!")
st.warning("Be careful!")
st.error("Something went wrong.")
```



STREAMLIT

LOAD A CSV FILE

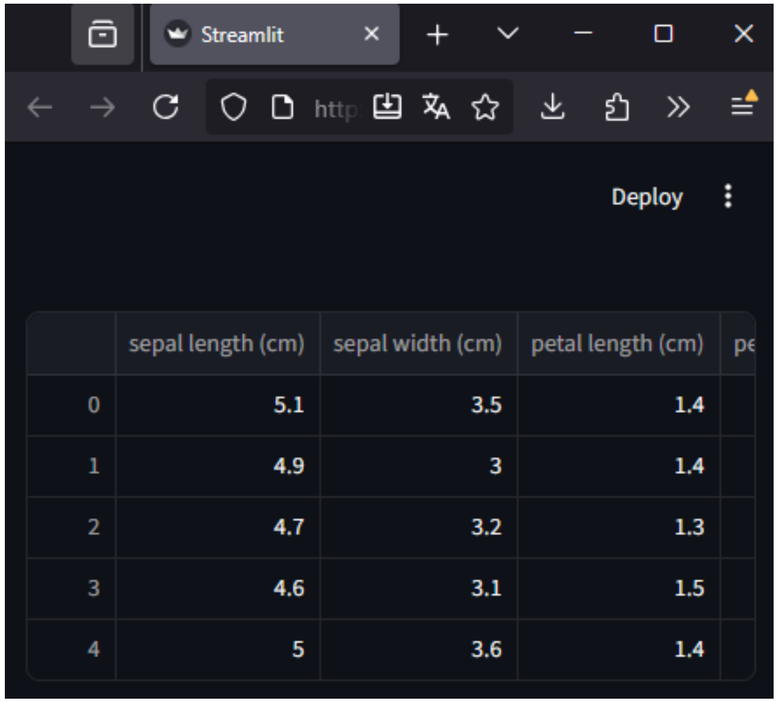
A dataset can be read directly from the project folder or from an URL.

app.py

```
import streamlit as st
from sklearn import datasets
import pandas as pd
import numpy as np


iris = datasets.load_iris()
df = pd.DataFrame(
    data= np.c_[iris['data'], iris['target']],
                 columns= iris['feature_names'] + ['target'])


st.write(df.head())
```



	sepal length (cm)	sepal width (cm)	petal length (cm)	pe
0	5.1	3.5	1.4	
1	4.9	3	1.4	
2	4.7	3.2	1.3	
3	4.6	3.1	1.5	
4	5	3.6	1.4	

STREAMLIT

ALLOW USER TO UPLOAD FILE

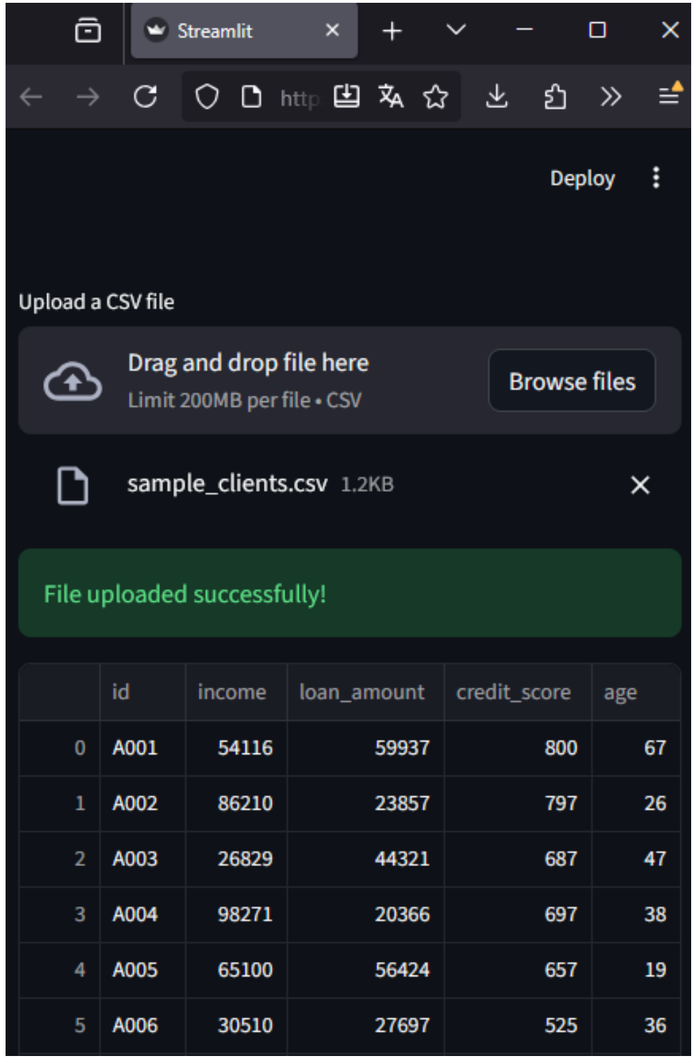
Streamlit provides a **file uploader widget** to load data dynamically.

app.py

```python
import streamlit as st
import pandas as pd

uploaded_file = st.file_uploader(
    "Upload a CSV file",
    type=["csv"])

if uploaded_file is not None:
    df = pd.read_csv(uploaded_file)
    st.success("File uploaded successfully!")
    st.dataframe(df)
```



	id	income	loan_amount	credit_score	age
0	A001	54116	59937	800	67
1	A002	86210	23857	797	26
2	A003	26829	44321	687	47
3	A004	98271	20366	697	38
4	A005	65100	56424	657	19
5	A006	30510	27697	525	36

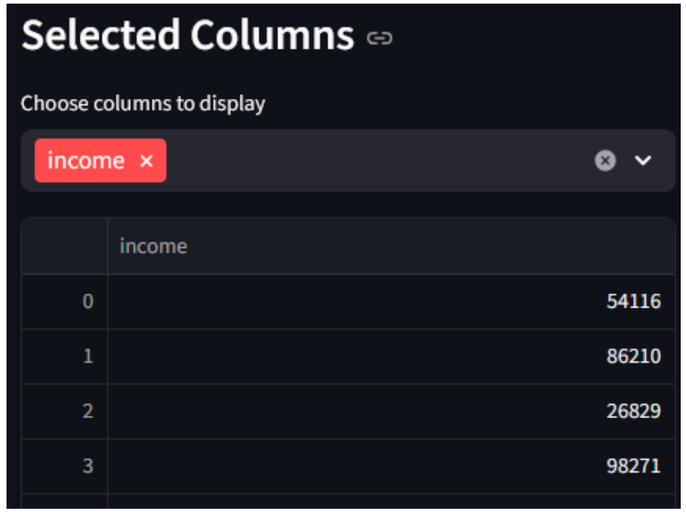
STREAMLIT

PERFORM BASIC OPERATIONS

The dataset can now be manipulated using **pandas**.

app.py

```
(...)
st.subheader("Selected Columns")
selected_columns = st.multiselect(
    "Choose columns to display",
    df.columns)
st.dataframe(df[selected_columns])
```



STREAMLIT

CUSTOMIZATION

A well-designed interface improves readability, usability, and engagement. Streamlit allows several levels of customization — from **themes** and **layout** to **page configuration**.

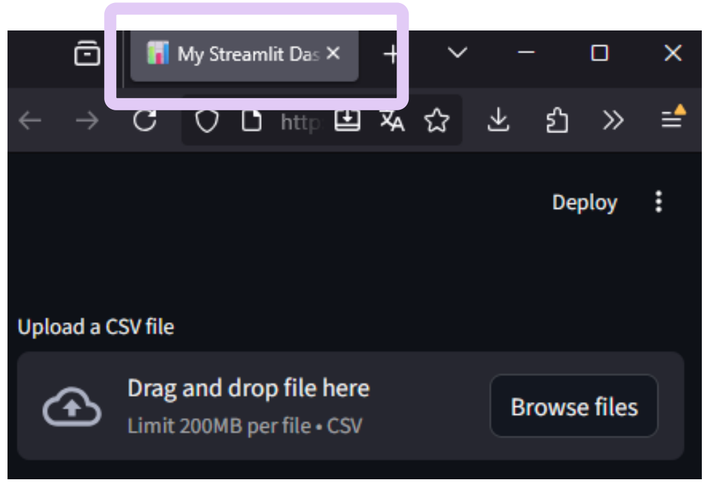
① Page Configuration

Define the app's title, icon, and layout.

app.py

```python
import streamlit as st
import pandas as pd

st.set_page_config(
    page_title="My Streamlit Dashboard",
    page_icon="📊",
    layout="wide",  # "centered" or "wide"
    initial_sidebar_state="expanded"
)
```



This line should appear before any other Streamlit commands!

STREAMLIT

CUSTOMIZATION

A well-designed interface improves readability, usability, and engagement. Streamlit allows several levels of customization — from **themes** and **layout** to **page configuration**.

② Themes

Streamlit supports **built-in themes** (Light/Dark) and **custom color palettes**.

Create a file named **.streamlit/config.toml** in the project root

.streamlit/config.toml

```
[theme]
primaryColor = "#1E88E5"
backgroundColor = "#F0F2F6"
secondaryBackgroundColor = "#FFFFFF"
textColor = "#262730"
font = "sans serif"
```

STREAMLIT

CUSTOMIZATION

A well-designed interface improves readability, usability, and engagement. Streamlit allows several levels of customization — from **themes** and **layout** to **page configuration**.

3 Layout

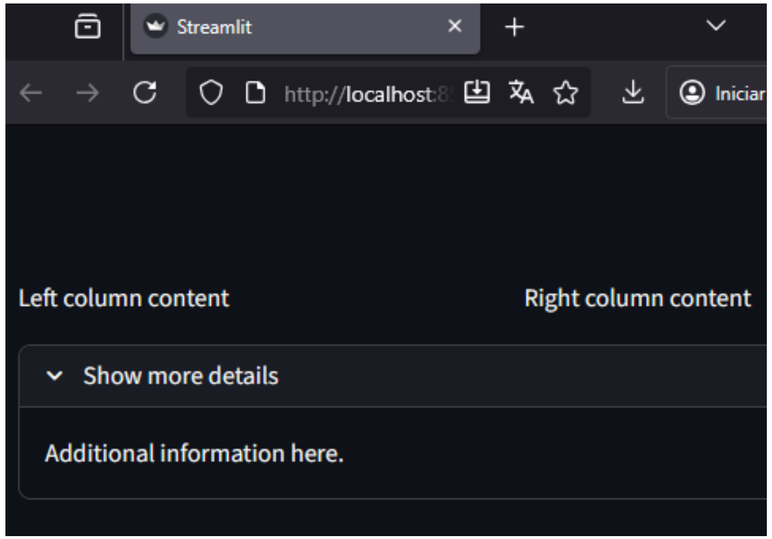
Use **columns**, **sidebars**, and **expanders** to organize the interface.

app.py

```python
import streamlit as st
import pandas as pd

col1, col2 = st.columns(2)
col1.write("Left column content")
col2.write("Right column content")

with st.expander("Show more details"):
    st.write("Additional information here.")
```



STREAMLIT

CUSTOMIZATION

A well-designed interface improves readability, usability, and engagement. Streamlit allows several levels of customization — from **themes** and **layout** to **page configuration**.

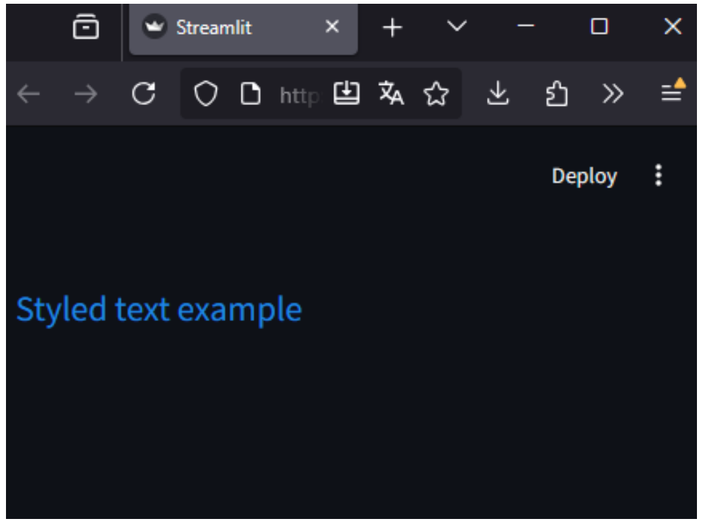
4 Custom HTML and CSS

HTML/CSS can be injected into Streamlit for fine-grained control using **st.markdown** with **unsafe_allow_html = True**

app.py

```python
import streamlit as st
import pandas as pd

st.markdown(
    """
    <style>
        .big-font {
            font-size:22px !important;
            color: #1E88E5;
        }
    </style>
    <p class="big-font">Styled text example</p>
    """,
    unsafe_allow_html=True
)
```



Use with care. This bypasses Streamlit's built-in style safety and **can break layout consistency**.

STREAMLIT

DEPLOYING STREAMLIT APPS

Once your app runs locally, the next step is to **share it with others**: teammates, students, or the public. **Deployment** allows anyone to access your Streamlit app directly through a web browser, without installing Python or dependencies.

1 Prepare the Project Directory

1. Create a **requirements.txt** file

```
TERMINAL
> pip freeze requirements.txt
```

Ensures the deployment environment matches the local setup

2. Make sure your app folder includes: **app.py** and **requirements.py**

STREAMLIT

DEPLOYING STREAMLIT APPS

Once your app runs locally, the next step is to **share it with others**: teammates, students, or the public. **Deployment** allows anyone to access your Streamlit app directly through a web browser, without installing Python or dependencies.

2 Push the app to Git

Commit and **push** your files

TERMINAL

```
> git add .
> git commit –m "Initial Streamlit app"
> git push -u origin main
```

Verify that **app.py** and **requirements.txt** appear in your GitHub repo.

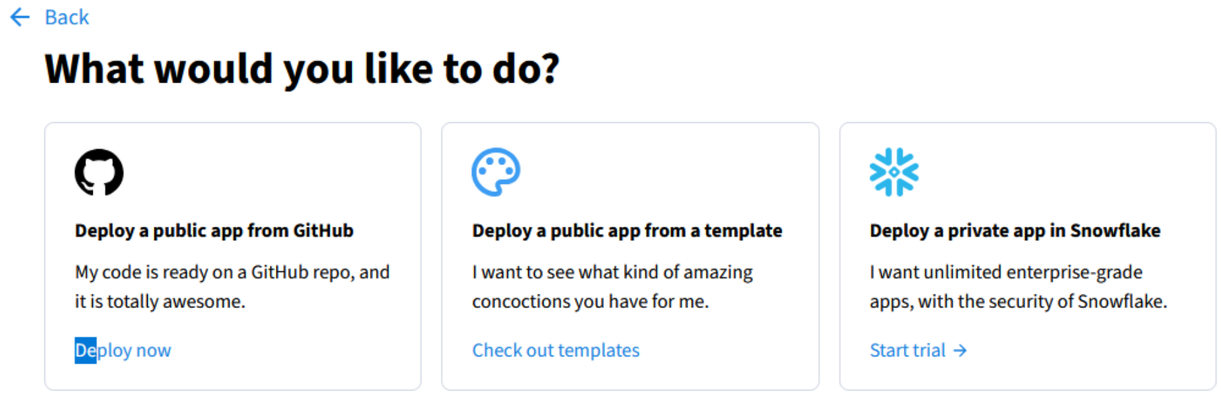
STREAMLIT

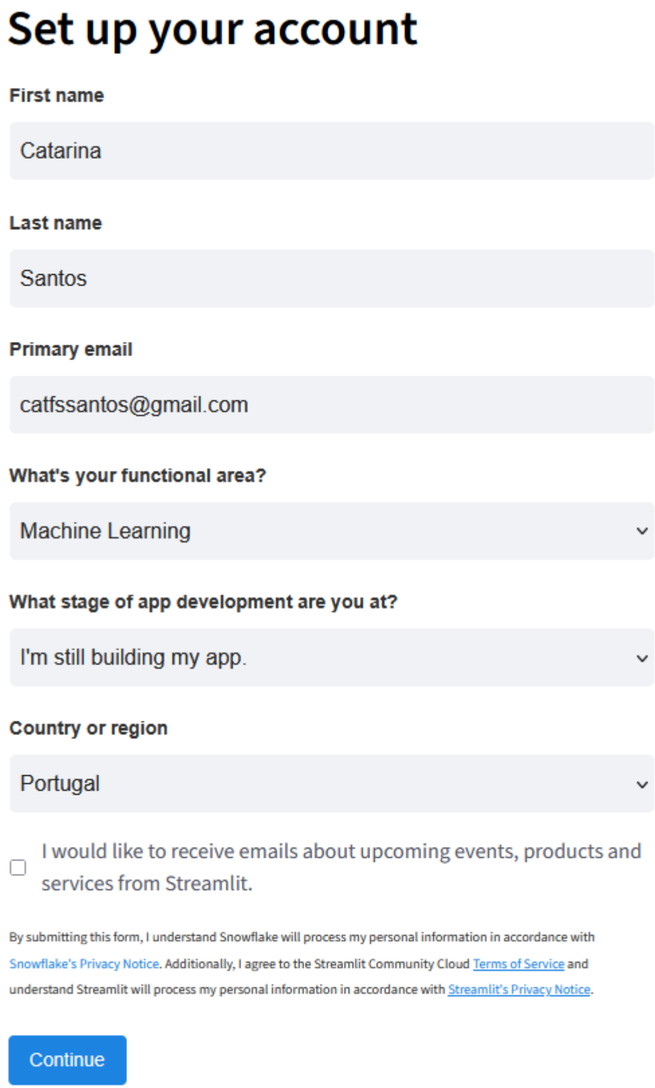
DEPLOYING STREAMLIT APPS

Once your app runs locally, the next step is to **share it with others**: teammates, students, or the public. **Deployment** allows anyone to access your Streamlit app directly through a web browser, without installing Python or dependencies.

3 Deploy on Streamlit Cloud

1. Go to https://share.streamlit.io
2. Log in with your GitHub Account and set up your Streamlit account.
3. Click **Create App**.
4. Select **Deploy a public app from GitHub**.





STREAMLIT

DEPLOYING STREAMLIT APPS

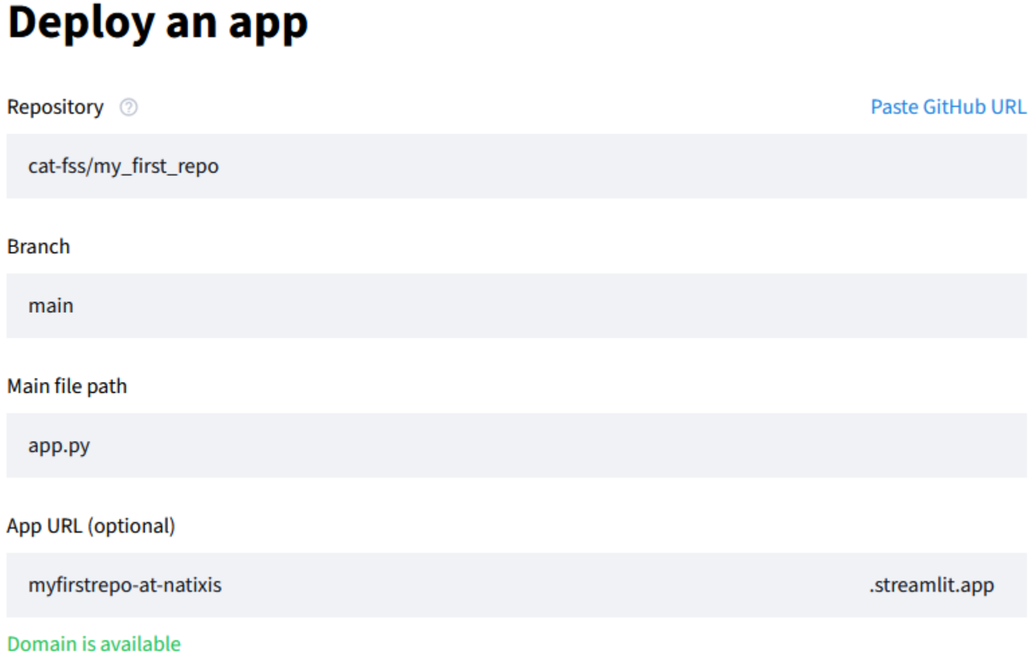
Once your app runs locally, the next step is to **share it with others**: teammates, students, or the public. **Deployment** allows anyone to access your Streamlit app directly through a web browser, without installing Python or dependencies.

3 Deploy on Streamlit Cloud

5. Select repository **name**, **branch** (main), and **file path** (app.py)
6. Click **Deploy**.

Streamlit will:

- Install dependencies from requirements.txt
- Launch the app on a public URL like https://myfirstrepo-at-natixis.streamlit.app/
- Automatically **detect changes** on GitHub and **redeploy** the app.



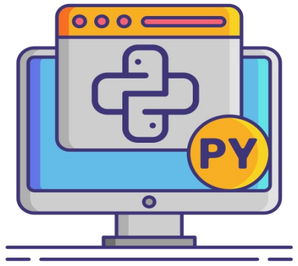
PRACTICE
PRACTICE
PRACTICE

YOU WON'T MASTER A SKILL IF YOU DON'T PRACTICE!

EXERCISES - LEARN BY DOING

In order to facilitate the learning process of Python **we have prepared for each session a python file** where you can find **exercises** that will help you to grasp the introduced Python concepts.

Visual Studio Code



We will use **VS CODE** as our Python program IDE

EXERCISES FOR TODAY

README.md added initial course structure and .README~ 17 hours ago

README

🐍 Natixis Python Level 3

📚 Course Overview

This advanced Python course is designed for students who have already mastered intermediate programming concepts and want to expand their skills toward object-oriented programming (OOP), modular design, and interactive application development. Students will learn to structure Python code professionally, understand reusability and abstraction, manage projects using Git and GitHub, and create Streamlit applications for data-driven interfaces.

📋 Course Structure (Intermediate Python - Level 3)

0 forks

Report repository

Releases

No releases published

Create a new release

Packages

No packages published

Publish your first package

Link to exercises: https://github.com/cat-fss/natixis_python_level_3/blob/master/exercises/Class3_exercises.py

WHY SHOULD YOU DEACTIVATE CO-PILOT?

As **beginners in Python programming**, it's crucial to focus on truly understanding how code works, rather than just seeing it appear. Tools like GitHub Copilot can be tempting, but they **often offer solutions without explanation**, making it easy to skip the learning process. While these tools are designed to assist, **not replace your thinking**, they can encourage you to rely on solutions you don't fully grasp—and they're not always correct. To truly learn, you need to write, debug, and explore code on your own. **By turning off Copilot** during the early stages of your learning, you give yourself the opportunity to develop real problem-solving skills, build confidence, and create a strong foundation. Later, when you have a solid grasp of the basics, Copilot can serve as a useful support tool, but always approach its suggestions with a critical mindset, not blind trust.

Steps to turn-off GitHub Copilot:

1. Go to Settings (File > Preferences > Settings or press Ctrl+,).
2. In the search bar, type: Copilot.
3. Find the setting GitHub Copilot: Enable.
4. Uncheck it to disable Copilot globally.

THANK YOU

Questions?

Catarina Santos catfssantos@gmail.com