gradio

Build & Share Delightful Machine Learning Apps

Team Members

Vahid Ebrahimian

Heading an liar of the group

Mahboobe Askarian

Teacher and Data Senior

Mohammad Javadpur

AI Senior & Data Scientist

Table of Content 1

01 Introduction to GUI libraries in PYTHON
- ❑ Tkinter
- ❑ PyQt
- ❑ PySide
- ❑ wxPython
- ❑ Kivy

02 Introduction to Gradio library
- ❑ What is Gradio?
- ❑ Why use Gradio?
- ❑ Features of Gradio

03 Installation and Setup
- ❑ Installing Gradio
- ❑ Setting up a development environment

04 Building Interactive Interfaces
- ❑ Creating input/output components
- ❑ Using custom components
- ❑ Adding styling and layout

Table of Content 2

05 Connecting Models to Gradio
- Integrating machine learning models
- Working with different model types
- Handling model inputs and outputs

06 Deploying Gradio Applications
- Hosting on local and remote servers
- Using cloud deployment platforms
- Best practices for deployment

07 Advanced Topics
- Handling errors and debugging
- Scaling Gradio applications
- Working with Gradio API

08 Conclusion and Resources
- Recap of Gradio features and benefits
- Additional resources and documentation

Welcome Message From Kaizen

“ **Gradio is a great library for quickly creating interactive web applications for machine learning models.** ”

What is a GUI?

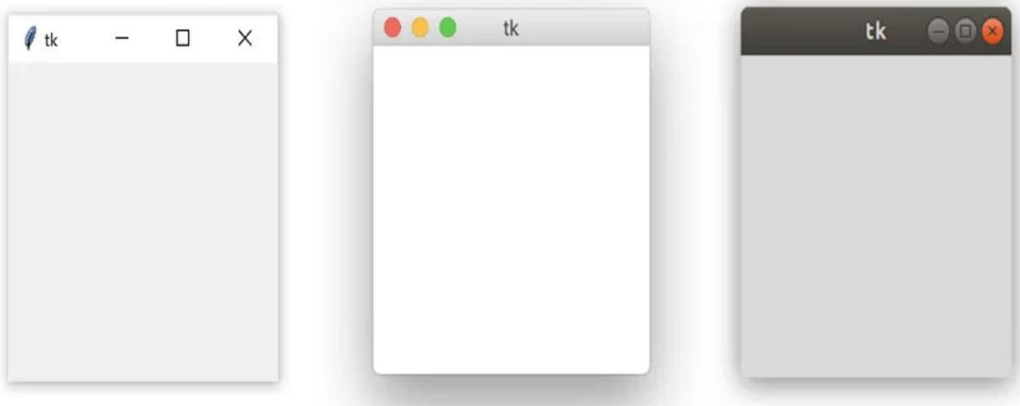
A GUI or a graphical user interface is an interactive environment to take responses from users on various situations such as forms, documents, tests, etc. It provides the user with a good interactive screen than a traditional Command Line Interface (CLI).

Tkinter

What is Tkinter used for ?

Another GUI framework is called Tkinter. Tkinter is one of the most popular Python GUI libraries for developing desktop applications. It’s a combination of the TK and python standard GUI framework.
Tkinter provides diverse widgets such as labels, buttons, textboxes, checkboxes that are used in a graphical user interface application.

```
import tkinter as tk

window = tk.Tk()
```



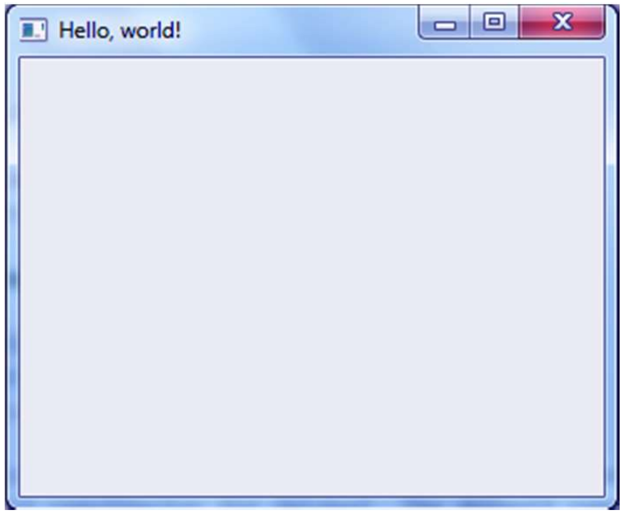
(a) Windows (b) macOS (c) Ubuntu

PyQT5

What is QT used for ?

PyQT5 is a graphical user interface (GUI) framework for Python. It is very popular among developers and the GUI can be created by coding or a QT designer. A QT Development framework is a visual framework that allows drag and drop of widgets to build user interfaces.

```
pip install pyqt5

import PyQt5


>>> import sys
>>> from PyQt5.QtWidgets import QApplication, QWidget
>>> app=QApplication(sys.argv)
>>> root=QWidget()
>>> root.resize(320,240)
>>> root.setWindowTitle('Hello, world!')
>>> root.show()
```

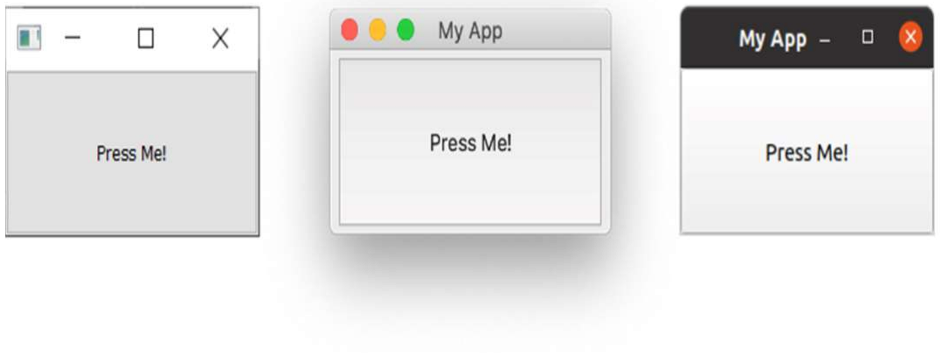


PySide

What is PySide used for ?

PySide, also known as Qt for Python, is a Python library for creating GUI applications using the Qt toolkit. PySide is the official binding for Qt on Python and is now developed by The Qt Company itself.

```
import sys

from PySide6.QtCore import QSize, Qt
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        self.setCentralWidget(button)

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec_()
```

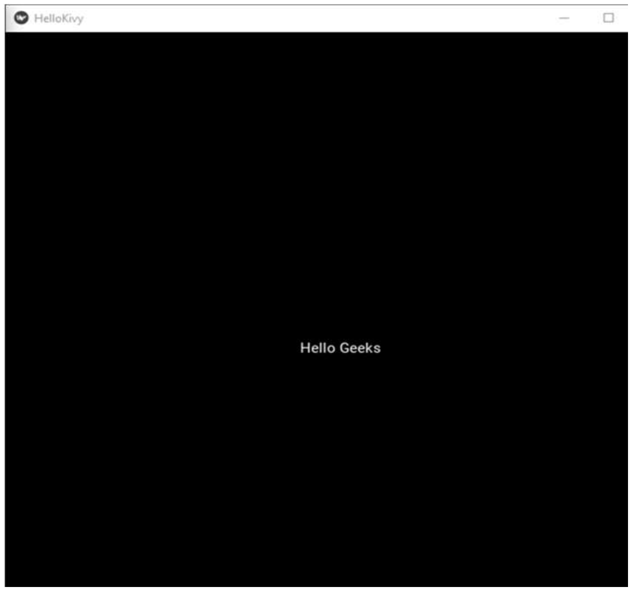


Kivy

What is Kivy used for ?

Kivy is an open source, **multi-platform application development framework** for Python. It allows us to develop multi-platform applications on various platforms such as **Windows**, **Linux**, **Android**, **macOS**, **iOS**, **and Raspberry Pi**.

```
import kivy
kivy.require('1.10.0')

from kivy.app import App
from kivy.uix.button Label
import

class HelloKivy(App):

    def build(self):

        return Label(text ="Hello Geeks")

helloKivy = HelloKivy()
helloKivy.run()
```

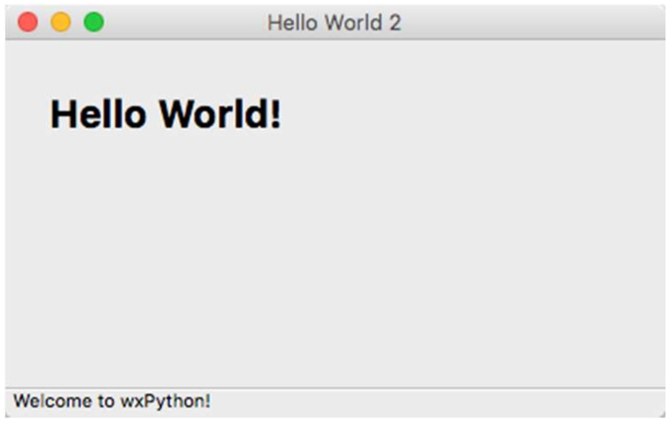


wxPython

What is wxPython used for?

wxPython is a cross-platform **GUI toolkit** for the Python programming language. It allows Python programmers to create programs with a robust, highly functional graphical user interface, simply and easily.

```
import wx

app = wx.App()

frm = wx.Frame(None, title="Hello World")

frm.Show()


app.MainLoop() torial/
```



Hello World!

Welcome to wxPython!

Django

Django is a Python web framework designed for building web applications and websites. It follows the Model-View-Controller (MVC) design pattern and provides tools and structures to facilitate the development process. It's used to create robust and feature-rich web applications by providing components like models, views, templates, and URL routing.

While Django and Gradio serve different purposes, they can be used together in certain scenarios. For example, you could develop a machine learning model using Python, integrate it into a Django web application, and use Gradio to create an interactive UI for users to interact with the model within the Django app. This would combine the web development capabilities of Django with the interactive UI-building capabilities of Gradio to create a comprehensive application.

Install Django:

•With the virtual environment activated, run the following command to install Django:

```
pip install django
```

Streamlit

Streamlit is an open-source Python library used for creating web applications for data science and machine learning projects. It allows developers to create interactive and data-driven web applications with minimal effort and coding. With Streamlit, you can turn data scripts into shareable web apps quickly.

Streamlit simplifies the process of turning data visualizations, plots, and models into interactive web applications. You can build dashboards, demos, and tools without requiring extensive web development expertise. The library is particularly popular among data scientists, engineers, and researchers who want to showcase their work or create user-friendly interfaces for their data-driven applications.

Installation:

Install Streamlit using pip, a Python package manager. Open your terminal or command prompt and run:

```
pip install streamlit
```

```
import streamlit as st
```

•While Streamlit and Gradio serve different primary purposes, they can complement each other in certain cases.
•You might use Streamlit to create an overarching application that includes data visualizations, insights, and explanations, and then embed a Gradio interface for users to directly interact with machine learning models within that application.
•For instance, you could build a Streamlit app to display stock market trends and insights and include a Gradio component that allows users to input a tweet and see the sentiment analysis prediction.

What is Gradio?

Gradio

Gradio is an open-source Python library that is used to build machine learning and data science demos and web applications.

With Gradio, you can quickly create a beautiful user interface around your machine learning models or data science workflow and let people "try it out" by dragging-and-dropping in their own images, pasting text, recording their own voice, and interacting with your demo, all through the browser.

Why use Gradio?

Gradio is useful for:

Demoing

your machine learning models for clients/collaborators/users/students.

Deploying

your models quickly with automatic shareable links and getting feedback on model performance.

Debugging

your model interactively during development using built-in manipulation and interpretation tools.

Features of Gradio:

There are several reasons why you might want to use Gradio:

1. Shareability: Gradio provides an easy way to share your machine learning models and other functions with others, without requiring them to install any additional software or dependencies. This can be especially useful if you want to share your models with non-technical users or people who don't have the same development environment as you.

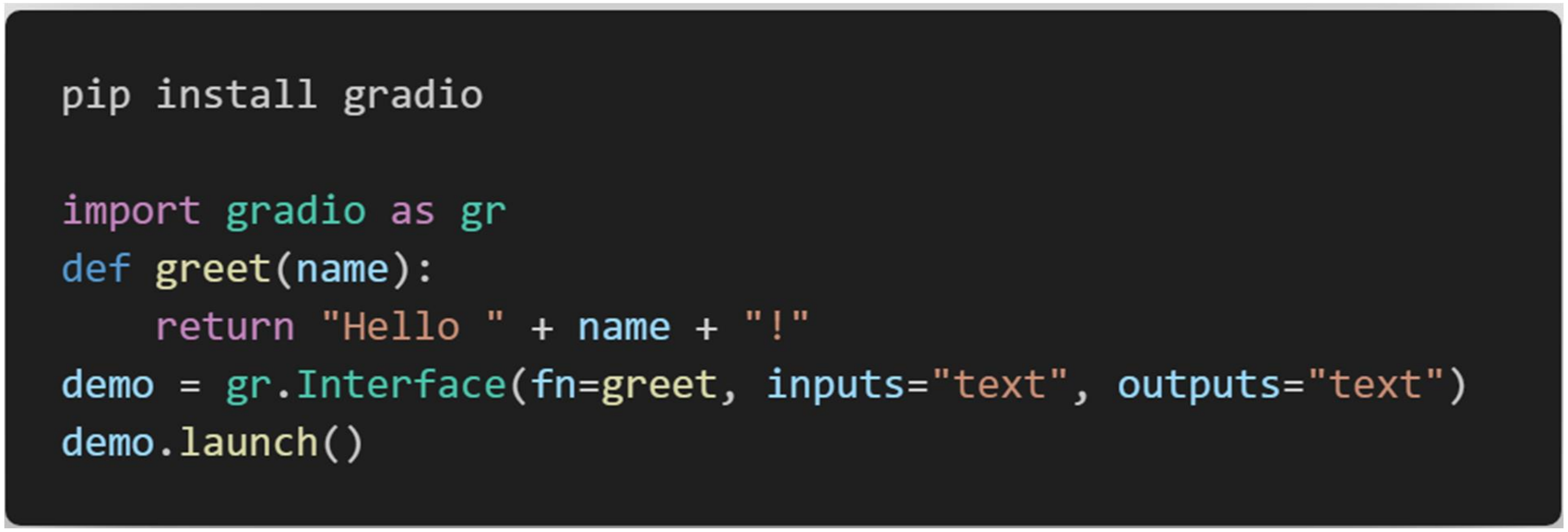
2. Customizable Interface: Gradio allows you to create a custom interface for your model that matches the style and branding of your application or website. This can help to improve the user experience and make your model more accessible to a wider audience.

3. Real-time Feedback: Gradio provides real-time feedback on the output of your model, allowing users to see the results of their input immediately. This can be especially useful for debugging and testing your model.

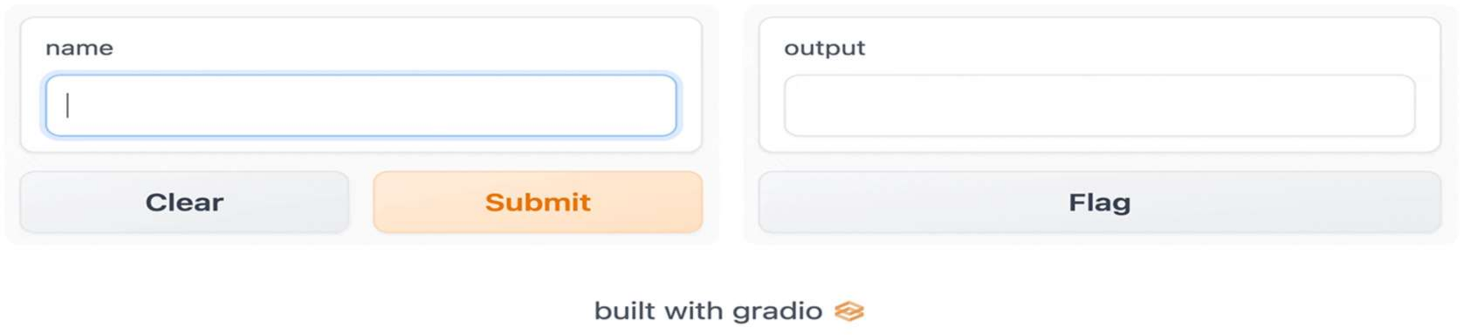
4. Multi-Framework Support: Gradio supports a wide range of popular machine learning frameworks, including TensorFlow, PyTorch, and scikit-learn. This means that you can use Gradio with the framework of your choice, without having to rewrite your code.

5. Multi-Language Support: Gradio supports multiple programming languages, including Python, R, and Julia. This means that you can use Gradio with the language of your choice, without having to learn a new language or switch to a different language.

Installing Gradio:

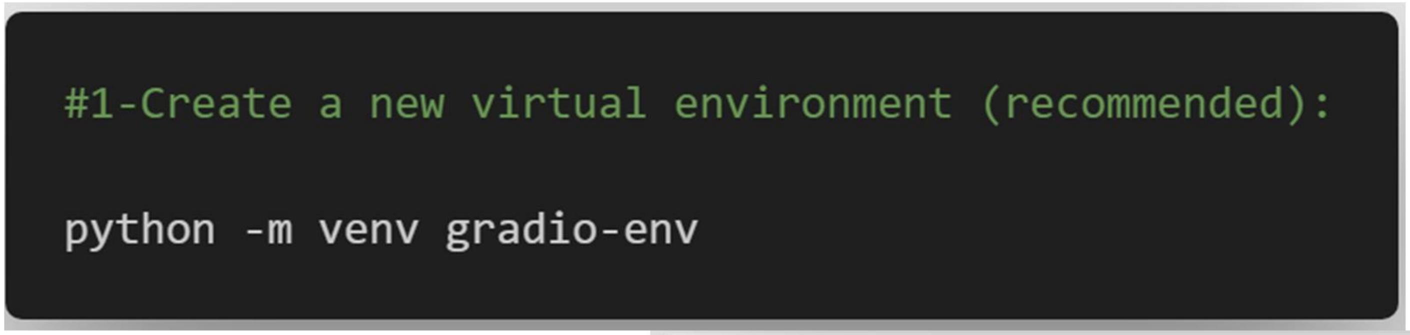


```
pip install gradio

import gradio as gr
def greet(name):
    return "Hello " + name + "!"
demo = gr.Interface(fn=greet, inputs="text", outputs="text")
demo.launch()
```

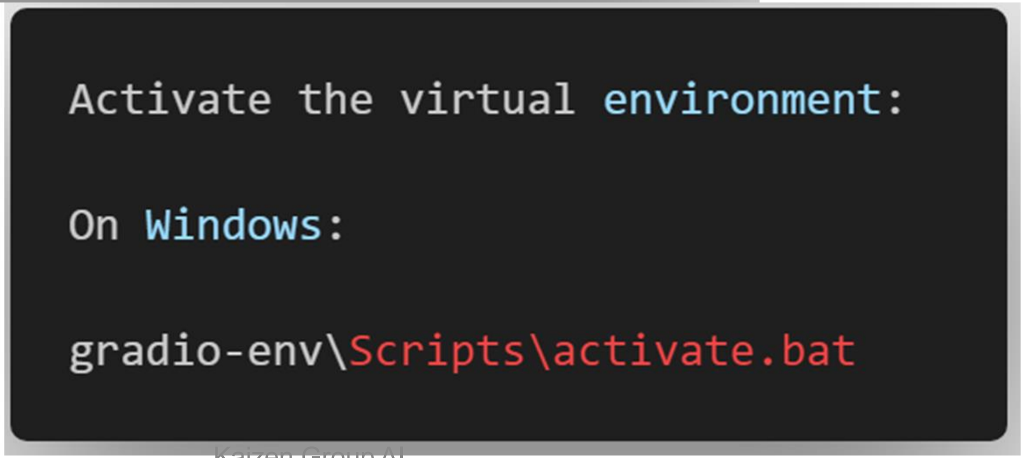


Setting up a development environment Gradio

- To set up a development environment for Gradio, you will need to have Python 3.8 or later installed on your computer. Here are the steps to set up a development environment for Gradio:



```
#1-Create a new virtual environment (recommended):

python -m venv gradio-env
```



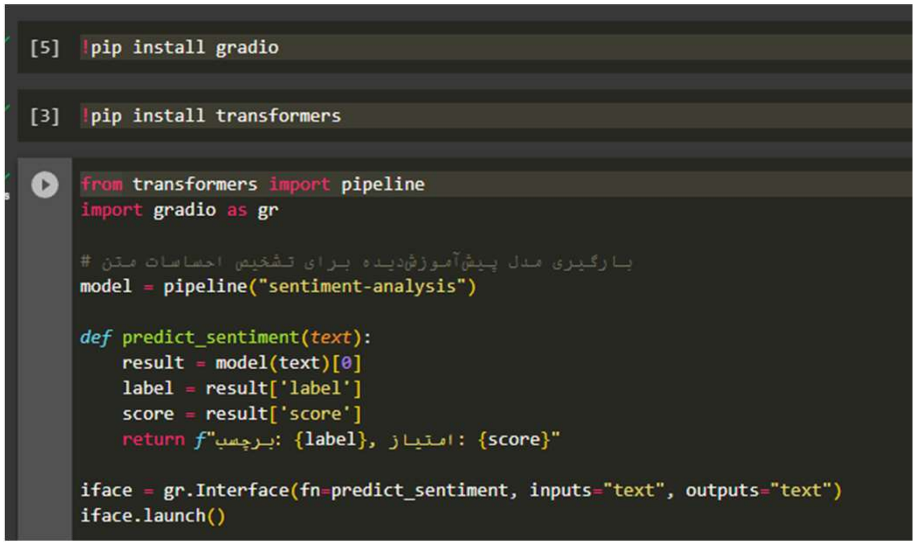
```
Activate the virtual environment:

On Windows:

gradio-env\Scripts\activate.bat
```

Hugging Face And Gradio

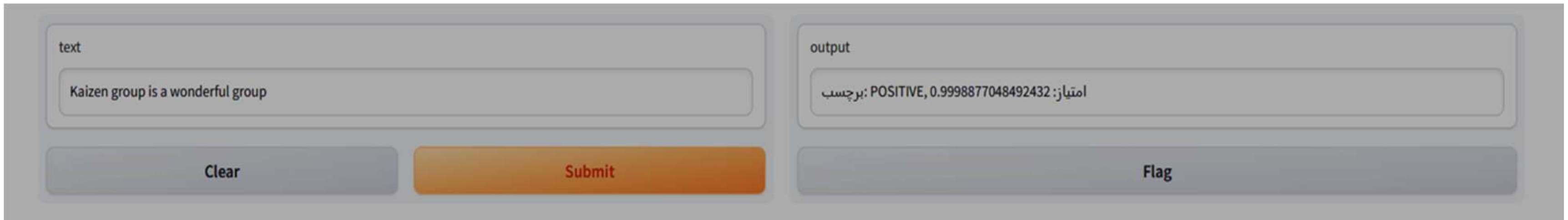
Hugging Face provides a wide range of pre-trained models for various NLP tasks, including sentiment analysis, text classification, machine translation, and more. These models are available in popular architectures such as BERT, GPT, RoBERTa, and others. Hugging Face also offers a convenient Python library called "transformers" that allows you to easily load, use, and fine-tune these pre-trained models.

On the other hand, Gradio is a user interface library for ML models. It enables you to quickly create and deploy web-based interfaces for your ML models, including NLP models. Gradio simplifies the process of building interactive interfaces, allowing users to interact with your models by entering text, making selections, and visualizing the results.

By combining Hugging Face and Gradio, you can build powerful NLP applications with user-friendly interfaces. For example, you can use a pre-trained sentiment analysis model from Hugging Face and create a Gradio interface where users can input text and instantly see the sentiment analysis results.



```
[5] !pip install gradio

[3] !pip install transformers

from transformers import pipeline
import gradio as gr

# بارگیری مدل پیش‌آموزش‌دیده برای تشخیص احساسات متن
model = pipeline("sentiment-analysis")

def predict_sentiment(text):
    result = model(text)[0]
    label = result['label']
    score = result['score']
    return f"برچسب: {label}, امتیاز: {score}"

iface = gr.Interface(fn=predict_sentiment, inputs="text", outputs="text")
iface.launch()
```



text: Kaizen group is a wonderful group

output: برچسب: POSITIVE, امتیاز: 0.9998877048492432

Recap of Gradio features and benefits

Features:

Customizable input and output forms:
Users can create custom forms for their models, allowing them to accept a wide range of inputs, including images, text, and other data types.
Real-time feedback:
Gradio provides real-time feedback on the output of the model, allowing users to see the results of their input immediately.
Customizable UI:
The Gradio interface can be customized to match the style and branding of the user's application or website.
Shareable:
Gradio provides a simple way to share models with others, without requiring any additional software or dependencies.
Multi-Framework Support:
Gradio supports a wide range of popular machine learning frameworks, including TensorFlow, PyTorch, and scikit-learn.
Multi-Language Support:
Gradio supports multiple programming languages, including Python, R, and Julia.
Collaboration:
Gradio allows multiple users to collaborate on a shared model, allowing them to work together to improve the model's performance or create new models from scratch.

Recap of Gradio features and benefits

Benefits:

Ease of use:
Gradio provides an easy-to-use interface for creating web-based interfaces for machine learning models and other functions.
Customizability:
Gradio allows users to create custom interfaces that match the specific needs of their application or website.
Shareability:
Gradio provides a simple way to share models with others, without requiring any additional software or dependencies.
Real-time feedback:
Gradio provides real-time feedback on the output of the model, allowing users to see the results of their input immediately.
Multi-Framework Support:
Gradio supports a wide range of popular machine learning frameworks, allowing users to use the framework of their choice.
Multi-Language Support:
Gradio supports multiple programming languages, allowing users to use the language of their choice.
Collaboration:
Gradio allows multiple users to collaborate on a shared model, allowing them to work together to improve the model's performance or create new models from scratch.

Resources and Documentation for Gradio:

- https://www.askpython.com/python-modules/top-best-python-gui-libraries
- https://realpython.com/python-gui-tkinter/
- https://data-flair.training/blogs/python-pyqt5-tutorial/
- https://www.pythonguis.com/pyside6-tutorial/
- https://www.geeksforgeeks.org/introduction-to-kivy/
- https://www.analyticsvidhya.com/blog/2023/02/streamlit-vs-gradio-a-guide-to-building-dashboards-in-python/

- Gradio Documentation: The official Gradio documentation provides a comprehensive guide for getting started with Gradio, including tutorials, API reference, and examples. You can visit the documentation at **https://gradio.app/docs/. ↗**

- Gradio GitHub Repository: The Gradio GitHub repository contains the source code for Gradio, including examples, documentation, and issue tracking. You can visit the repository at **https://github.com/gradio-app/gradio. ↗**

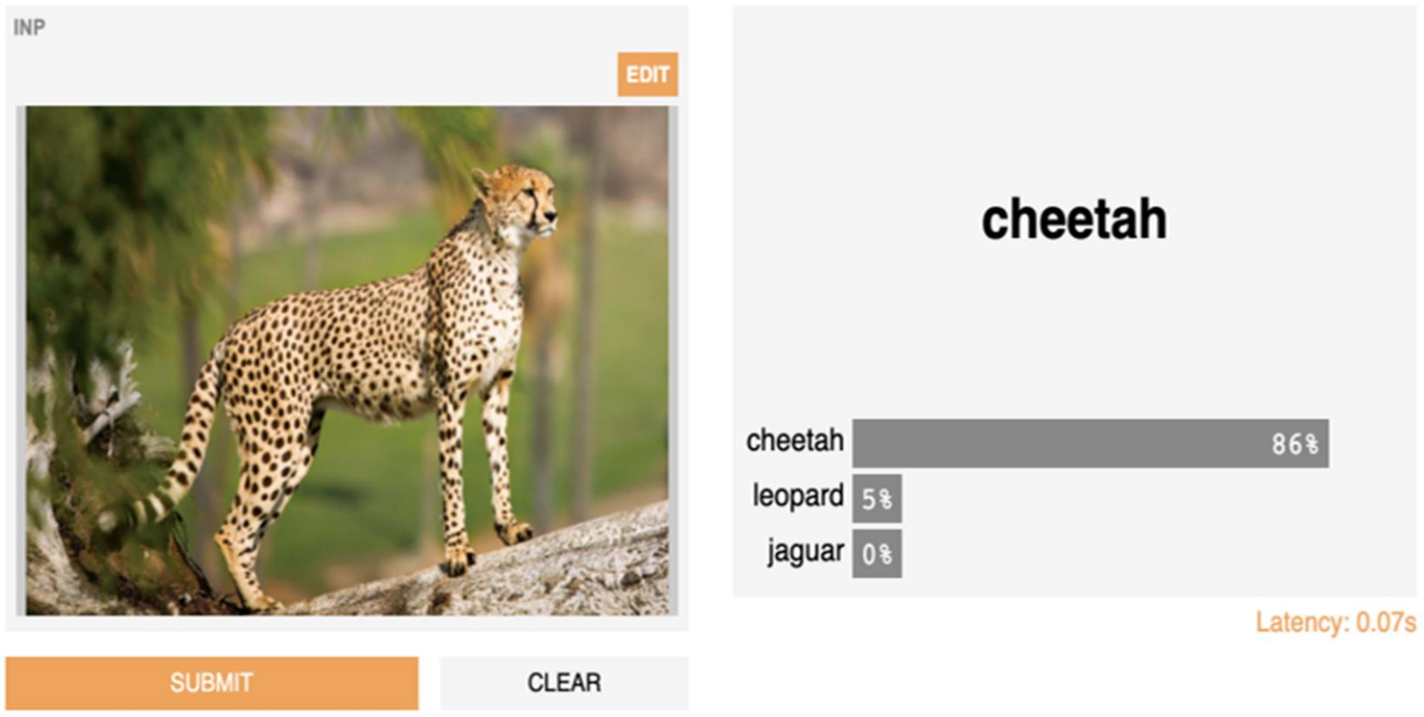
- Gradio YouTube Channel: The Gradio YouTube channel contains a series of video tutorials and demos of Gradio features and functionality. You can visit the channel at **https://www.youtube.com/channel/UCdyjiMAZMqyChLxXrSPk7iQ. ↗**

- Gradio Gitter Community: The Gradio Gitter community provides a forum for developers to ask questions, share ideas, and discuss issues related to Gradio. You can join the community at **https://gitter.im/gradio-app/community. ↗**
- Gradio Blog: The Gradio blog contains articles and tutorials on various topics related to Gradio, including tips and tricks, use cases, and best practices. You can visit the blog at **https://gradio.app/blog/. ↗**

Building Interactive Interfaces

Building Interactive Interfaces

Image Classification



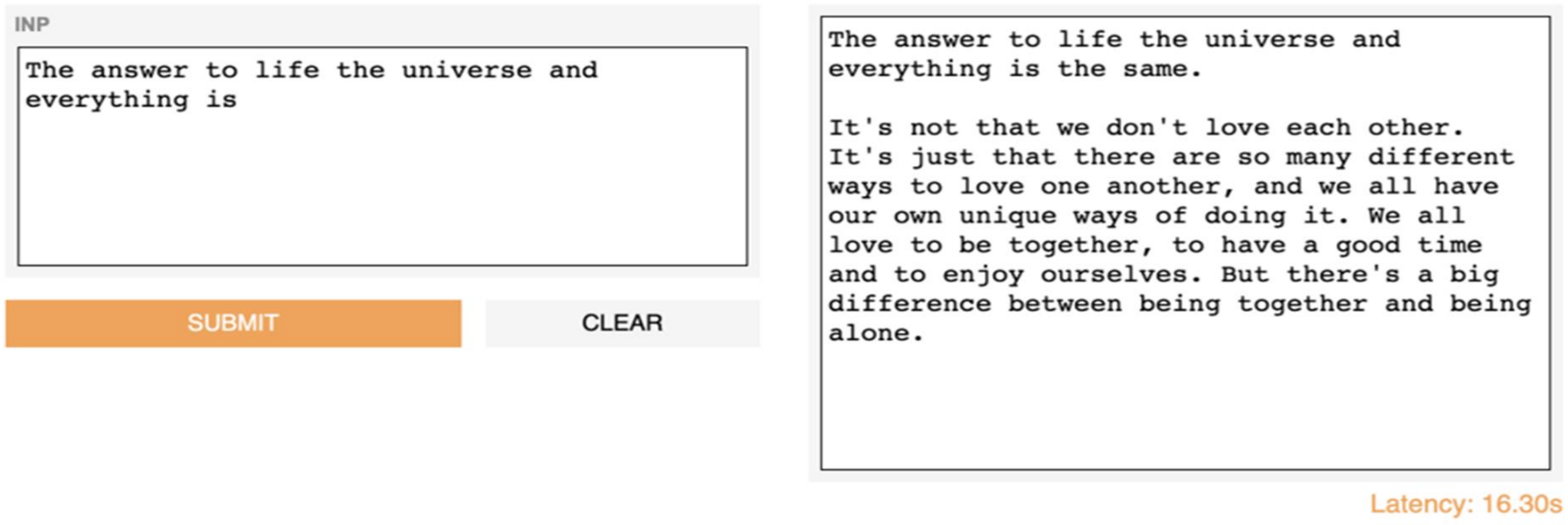
cheetah

```
cheetah  86%
leopard  5%
jaguar   0%
```

Latency: 0.07s

Building Interactive Interfaces

Text Generation with Transformers (GPT-2)

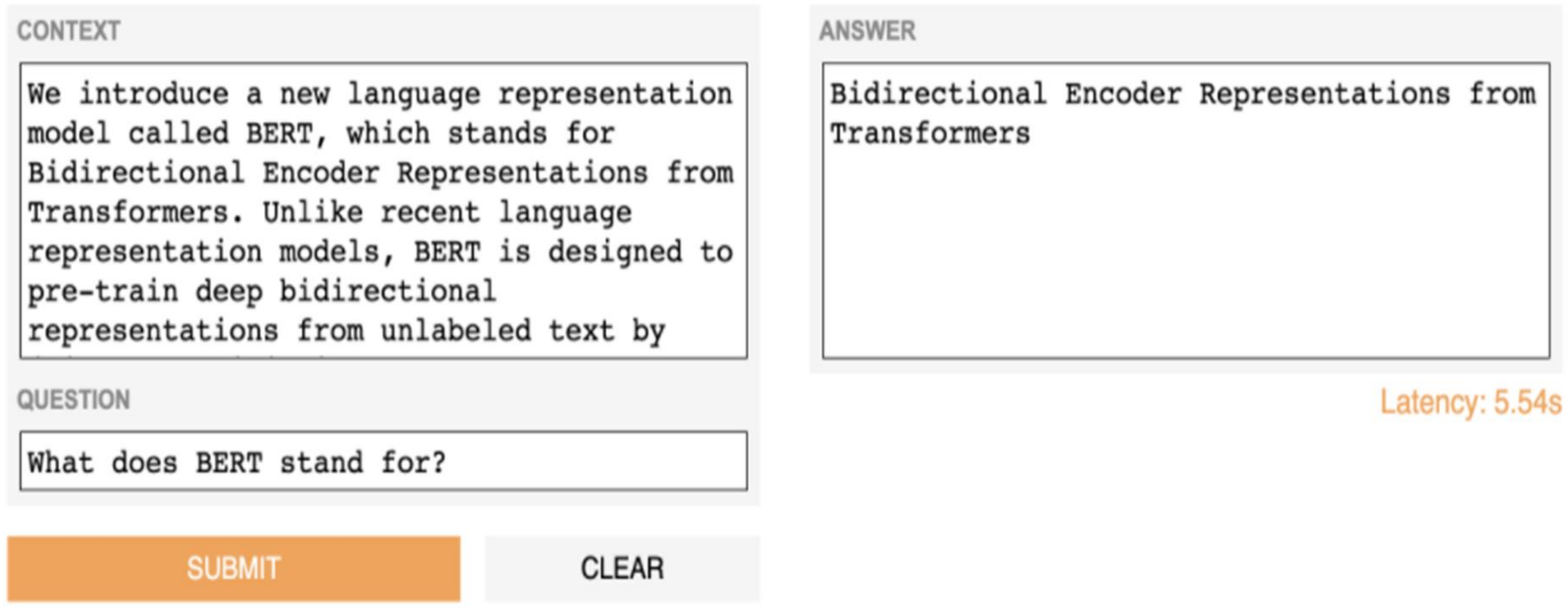


```
The answer to life the universe and
everything is
```

```
The answer to life the universe and
everything is the same.

It's not that we don't love each other.
It's just that there are so many different
ways to love one another, and we all have
our own unique ways of doing it. We all
love to be together, to have a good time
and to enjoy ourselves. But there's a big
difference between being together and being
alone.
```

Building Interactive Interfaces

- Answering Questions With BERT-QA



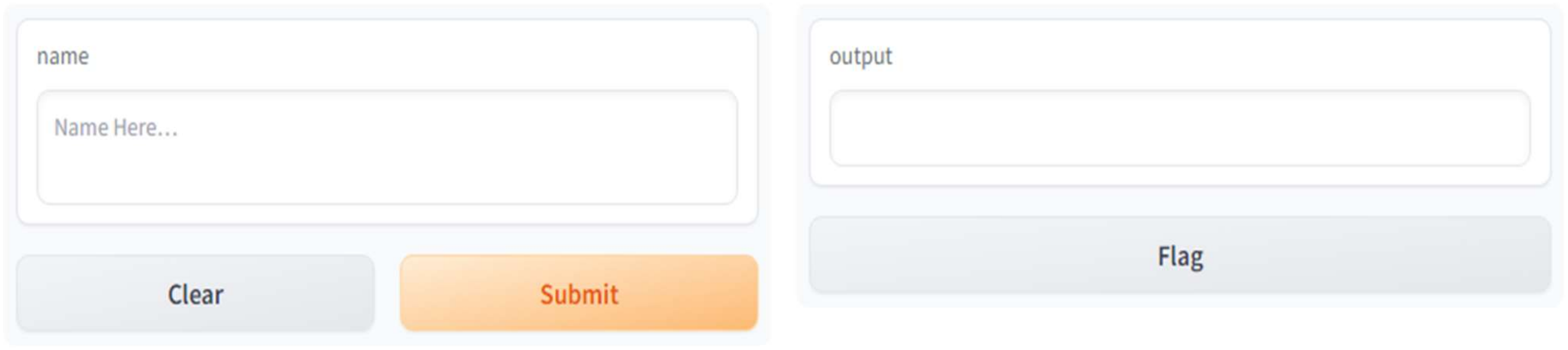
We introduce a new language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. Unlike recent language representation models, BERT is designed to pre-train deep bidirectional representations from unlabeled text by

What does BERT stand for?

Bidirectional Encoder Representations from Transformers

Latency: 5.54s

The Interface

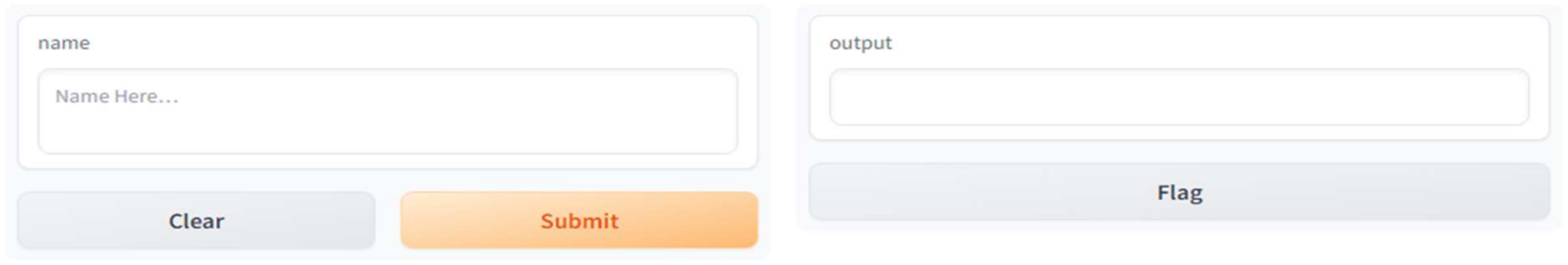


The Interface : The interface class has three parameters

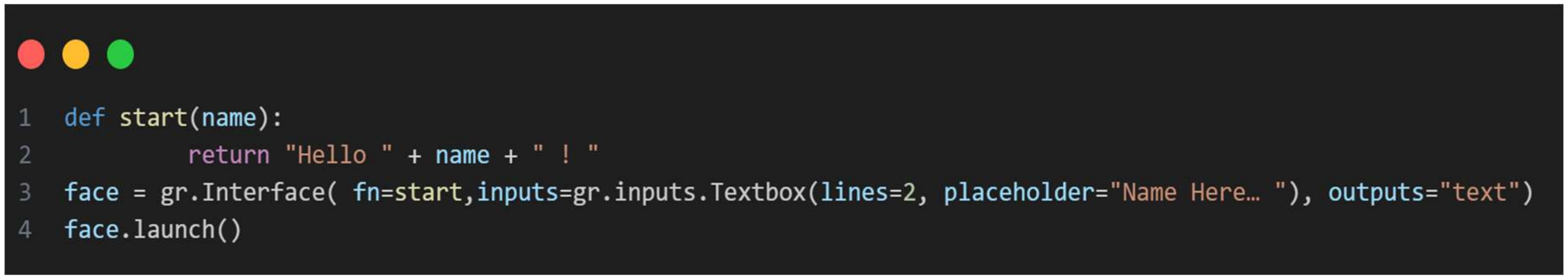
Gradio.interface(fn, input, output)

- **fn**: (Callable)the function to wrap an interface.
- **inputs**: (Union[str, List[Union[str, AbstractInput]]]) a single Gradio input component, or list of Gradio input components.
- **outputs**: a single Gradio output component, or list of Gradio output components.
- **live**: (bool) whether the interface should automatically reload on change.
- **capture session:** (bool) if True, captures the default graph and session
- **title:** (str) a title for the interface; if provided, appears above the input and output components.
- **description**: (str) a description for the interface; if provided, appears above the input and output components.
- **examples**: sample inputs for the function

The Interface

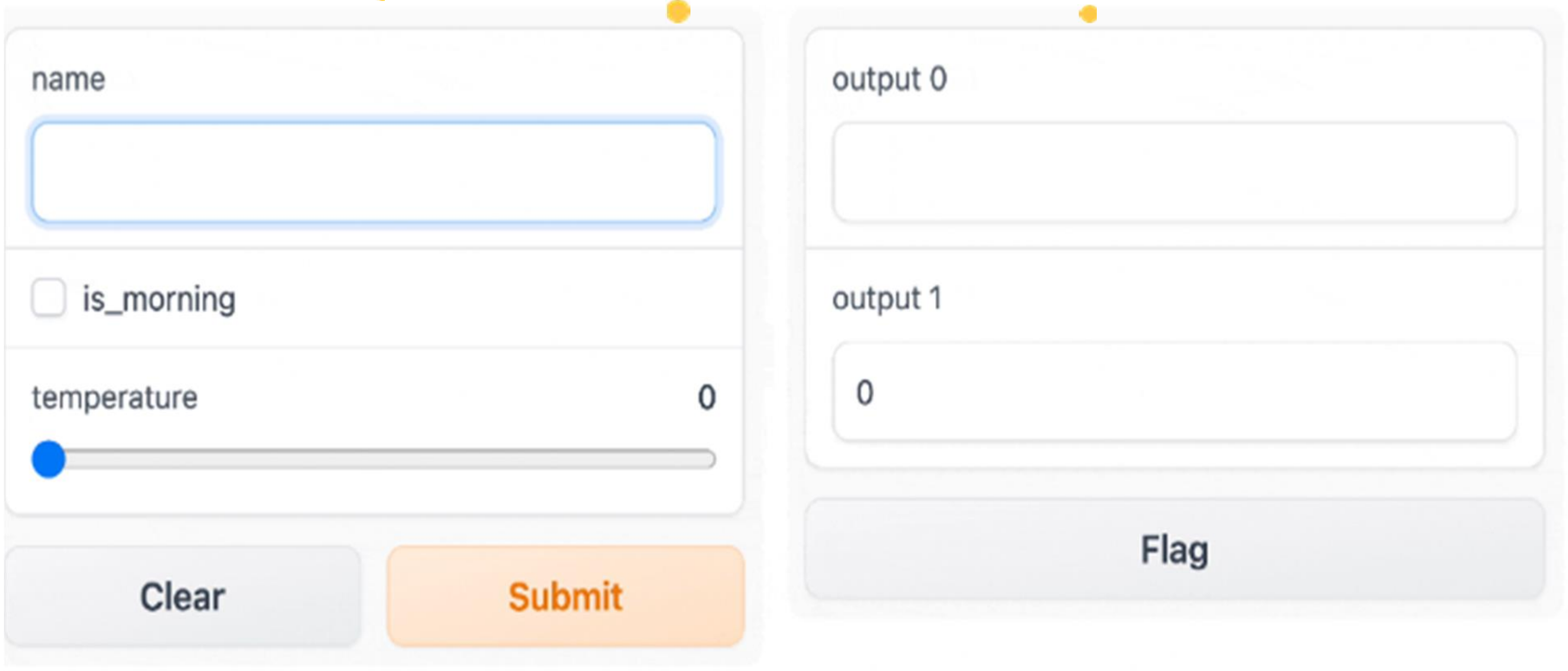


Activate Windows



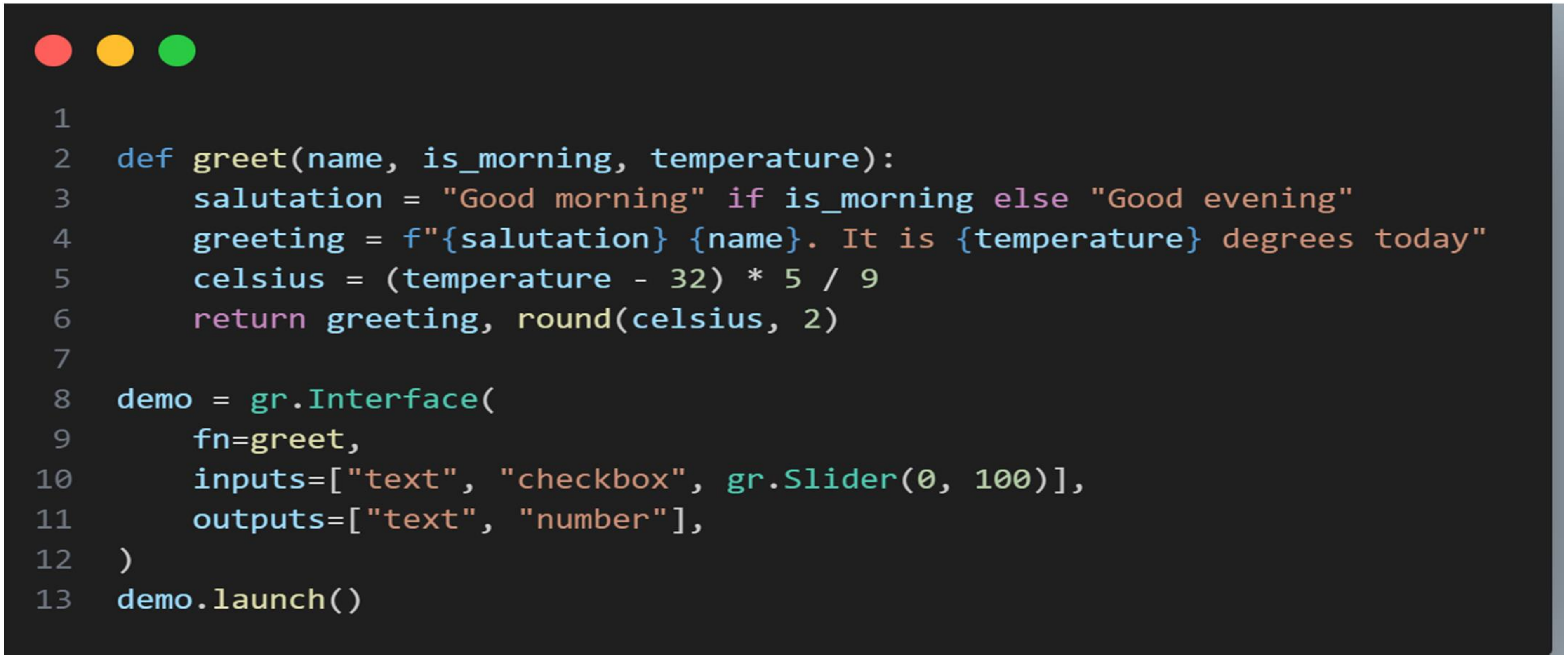
```python
def start(name):
        return "Hello " + name + " ! "
face = gr.Interface( fn=start,inputs=gr.inputs.Textbox(lines=2, placeholder="Name Here… "), outputs="text")
face.launch()
```

Multiple Input and Output Components



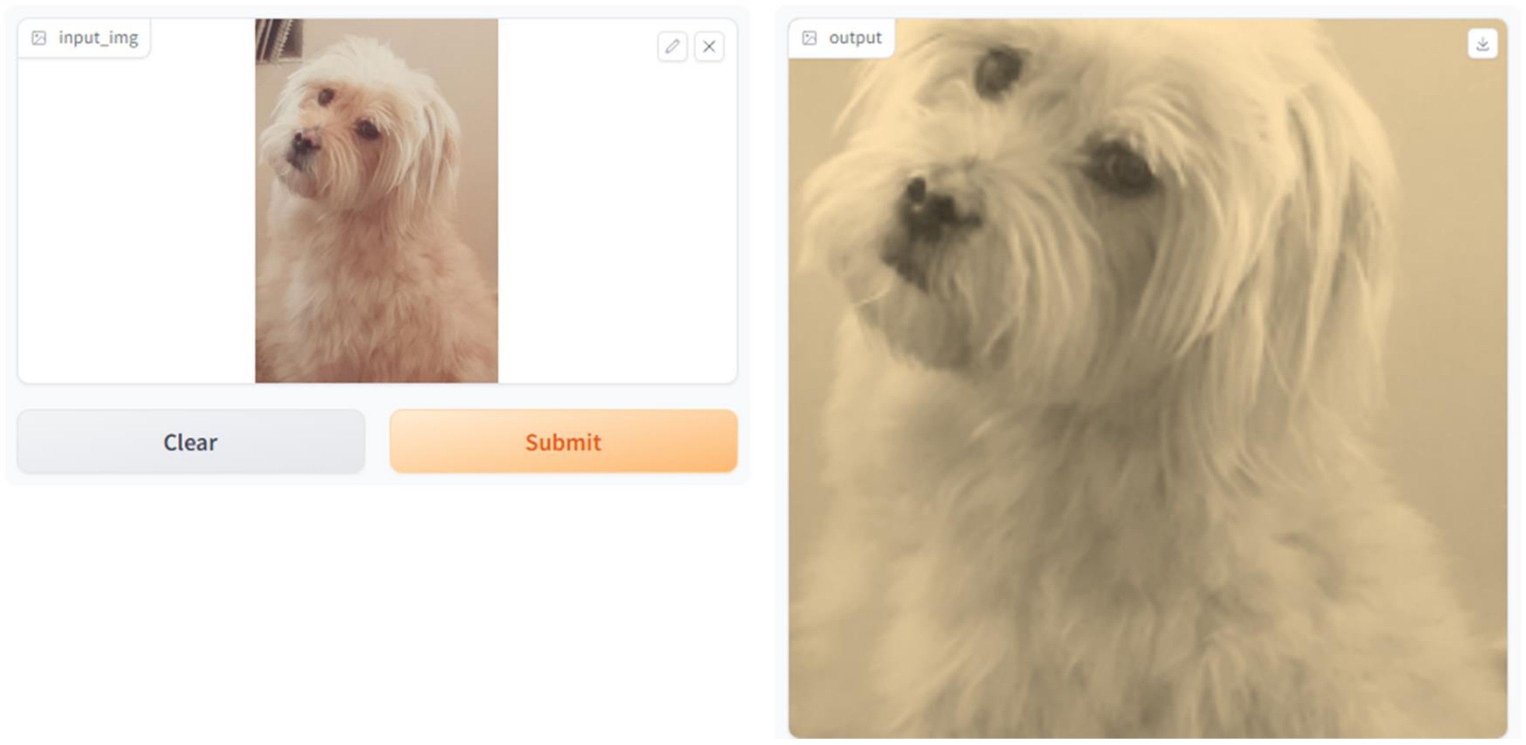
Multiple Input and Output Components

Each component in the inputs list corresponds to one of the parameters of the function, in order. Each component in the outputs list corresponds to one of the values returned by the function, again in order.



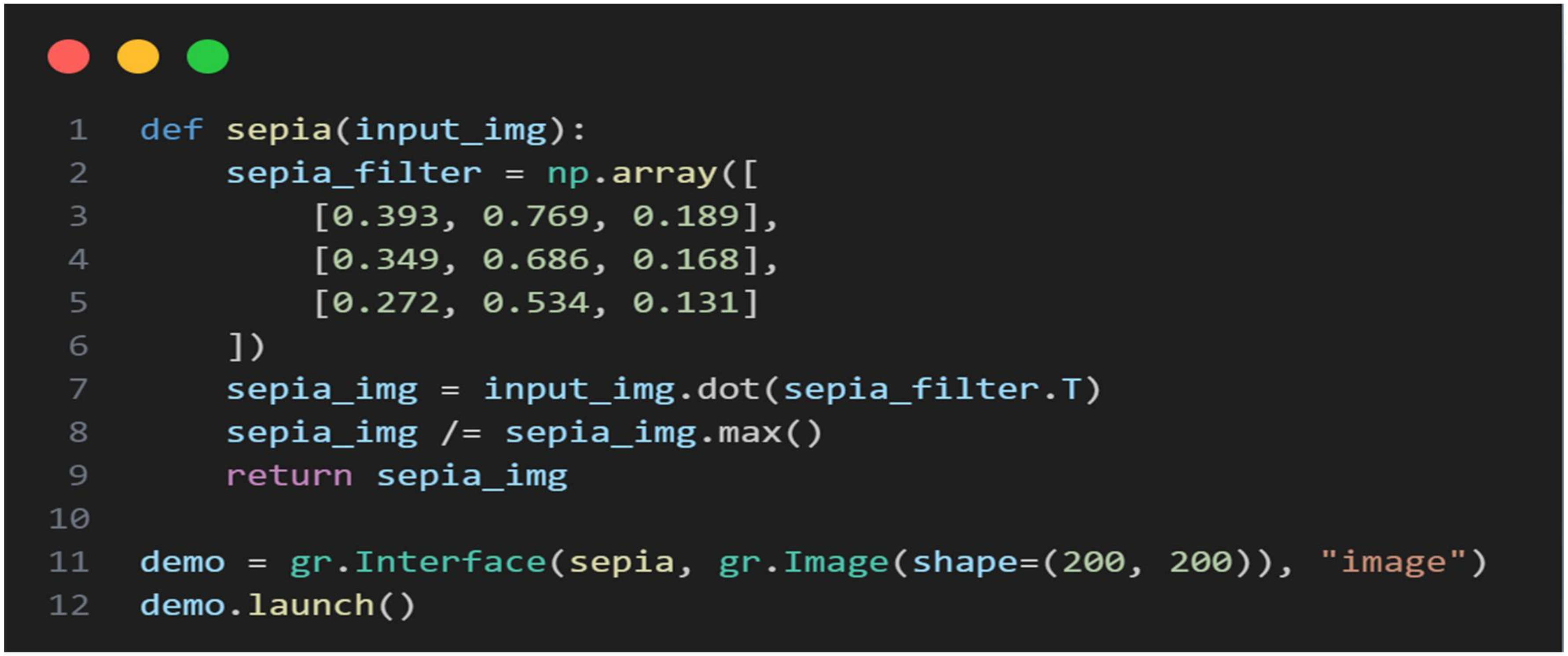
```python

def greet(name, is_morning, temperature):
    salutation = "Good morning" if is_morning else "Good evening"
    greeting = f"{salutation} {name}. It is {temperature} degrees today"
    celsius = (temperature - 32) * 5 / 9
    return greeting, round(celsius, 2)

demo = gr.Interface(
    fn=greet,
    inputs=["text", "checkbox", gr.Slider(0, 100)],
    outputs=["text", "number"],
)
demo.launch()
```

Multiple Input and Output Components



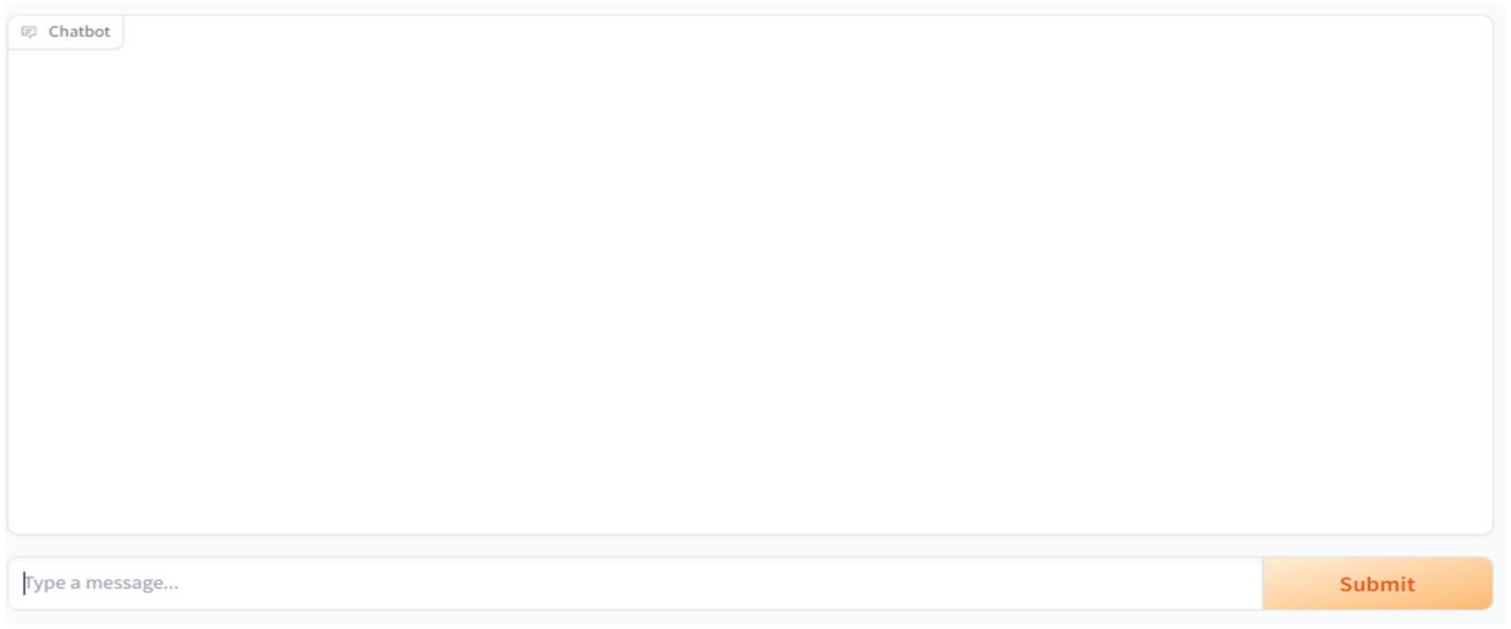
Multiple Input and Output Components

gr.Image(type="filepath", shape=...)



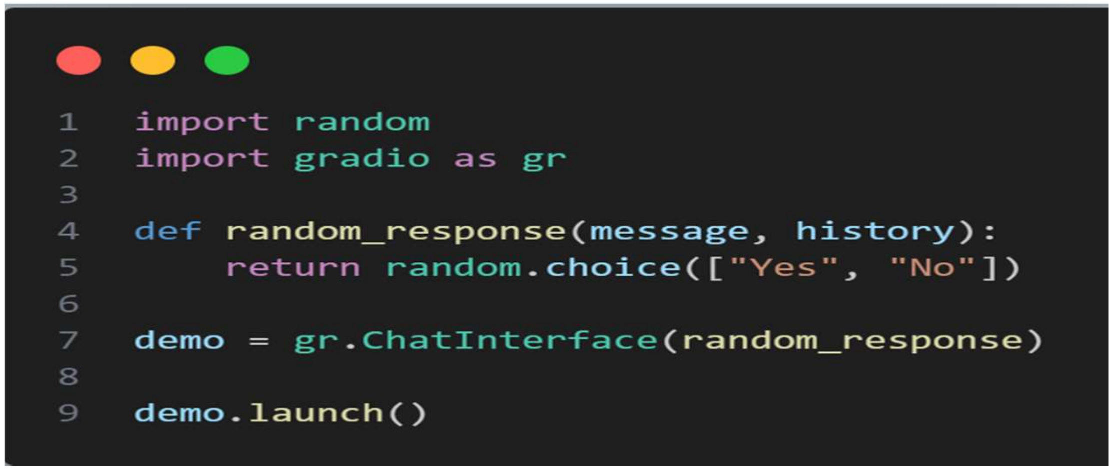
```python
def sepia(input_img):
    sepia_filter = np.array([
        [0.393, 0.769, 0.189],
        [0.349, 0.686, 0.168],
        [0.272, 0.534, 0.131]
    ])
    sepia_img = input_img.dot(sepia_filter.T)
    sepia_img /= sepia_img.max()
    return sepia_img

demo = gr.Interface(sepia, gr.Image(shape=(200, 200)), "image")
demo.launch()
```

Multiple Input and Output Components



Multiple Input and Output Components: Chatbots

- gr.ChatInterface is specifically designed for Chabot Uis



```python
import random
import gradio as gr

def random_response(message, history):
    return random.choice(["Yes", "No"])

demo = gr.ChatInterface(random_response)

demo.launch()
```

The function should take two arguments:

- Message: a str representing the user's input
- History: a list of list representing the conversations up until that point. Each inner list consists of two str representing a pair: [user input, bot response].

Blocks: More Flexibility and Control

Gradio approaches to build apps:

1. Interface and ChatInterface.
2. Blocks

Block:

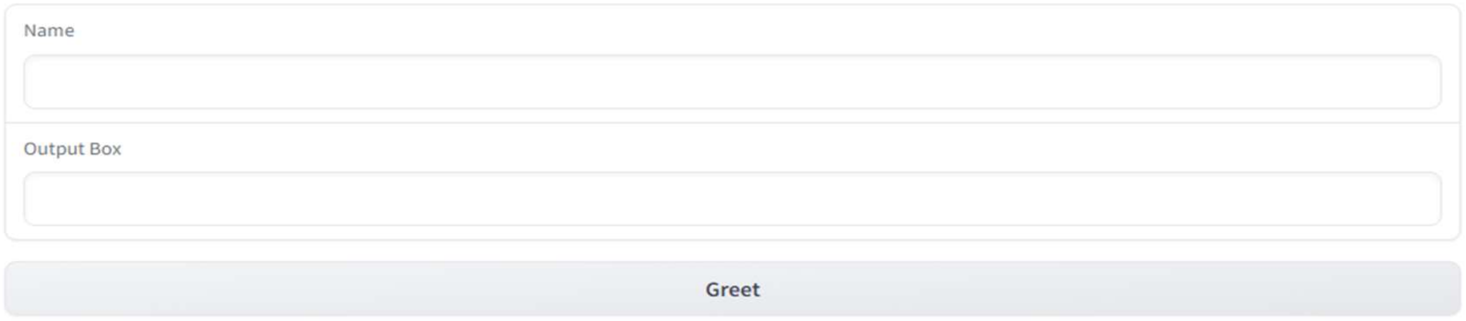
a low-level API for designing web apps with more flexible layouts and data flows.

Block:

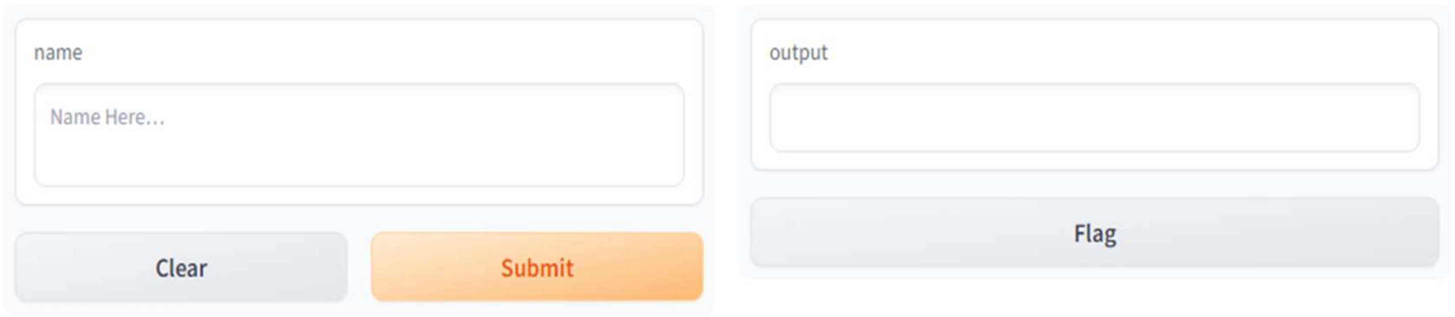
- allows you to do things like feature multiple data flows and demos, control where components appear on the page,
- handle complex data flows (e.g. outputs can serve as inputs to other functions)
- update properties/visibility of components based on user interaction — still all in Python.
- If this customizability is what you need, try Blocks instead!

Blocks: More Flexibility and Control

- Greeting by block:

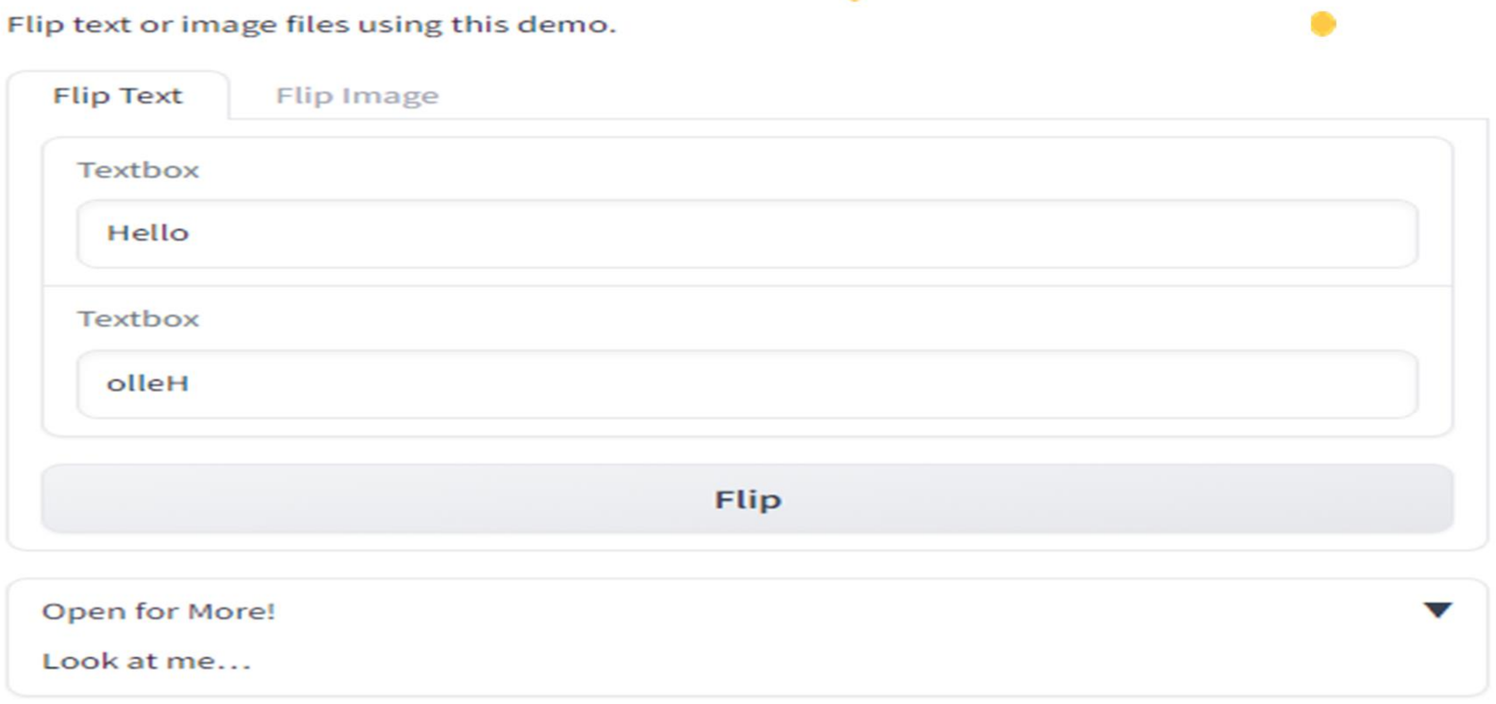


- Greeting by interface



Activate Windows

Blocks: More Flexibility and Control

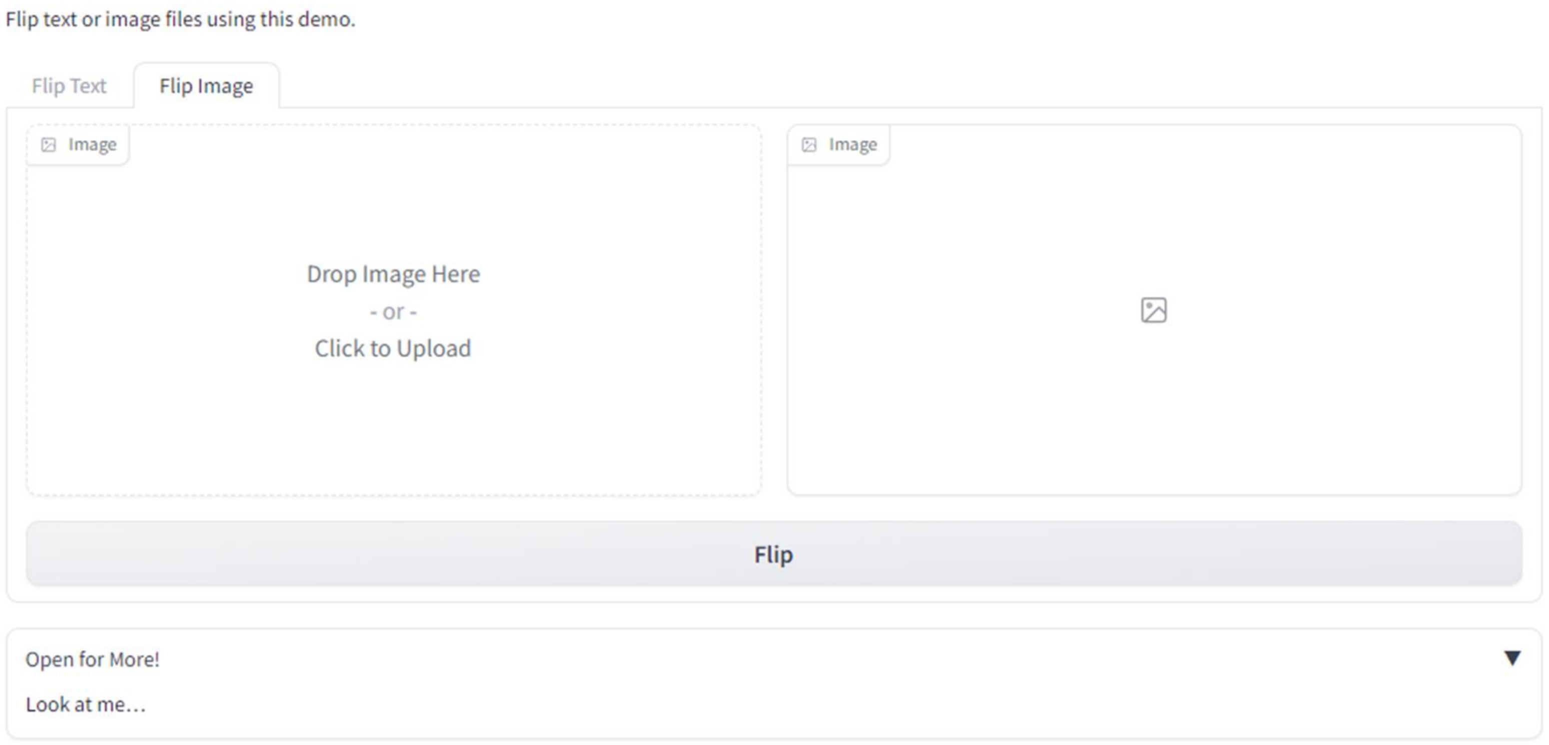

Flip text or image files using this demo.

Hello

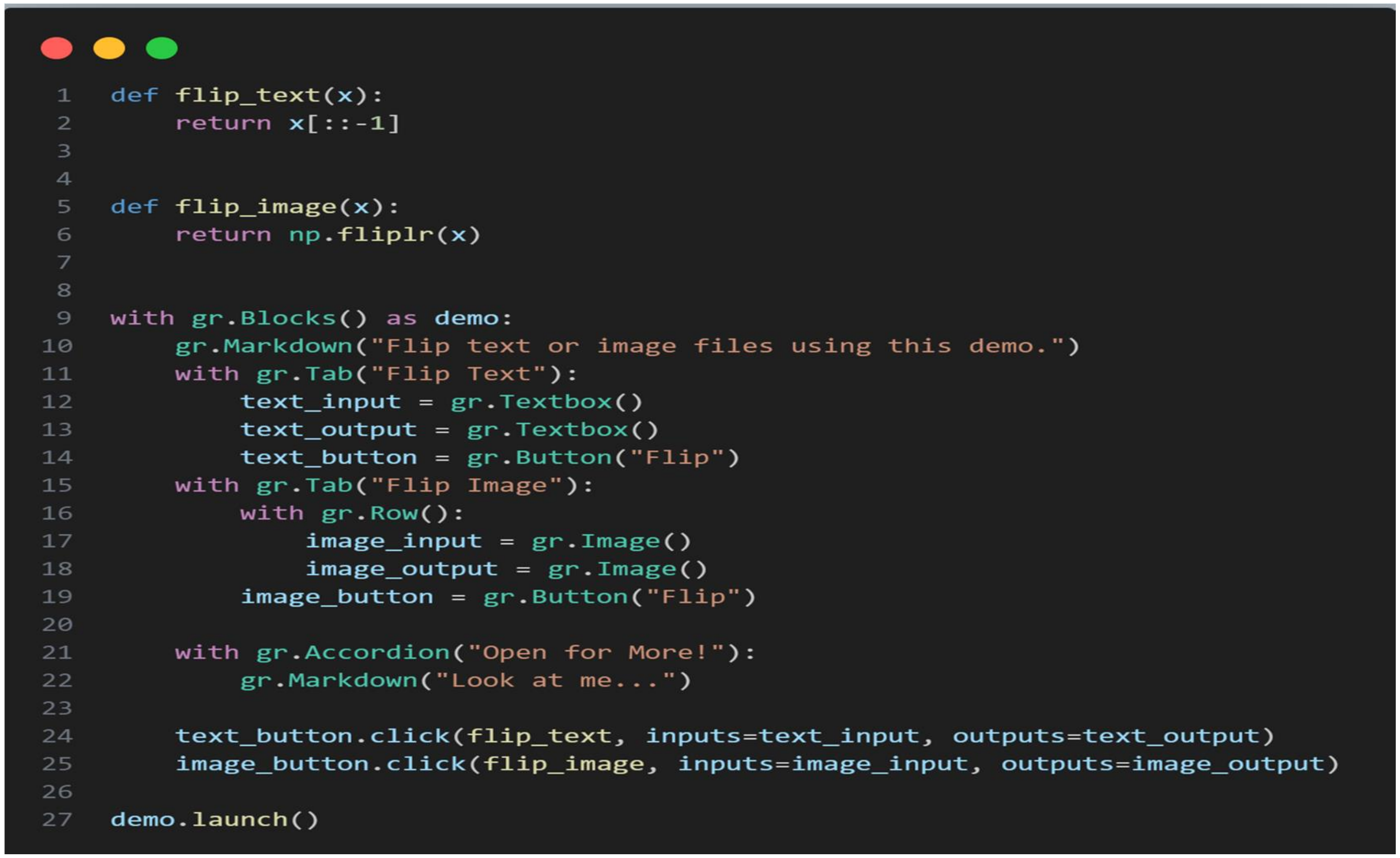
olleH

Look at me...

Blocks: More Flexibility and Control


Flip text or image files using this demo.

Blocks: More Flexibility and Control

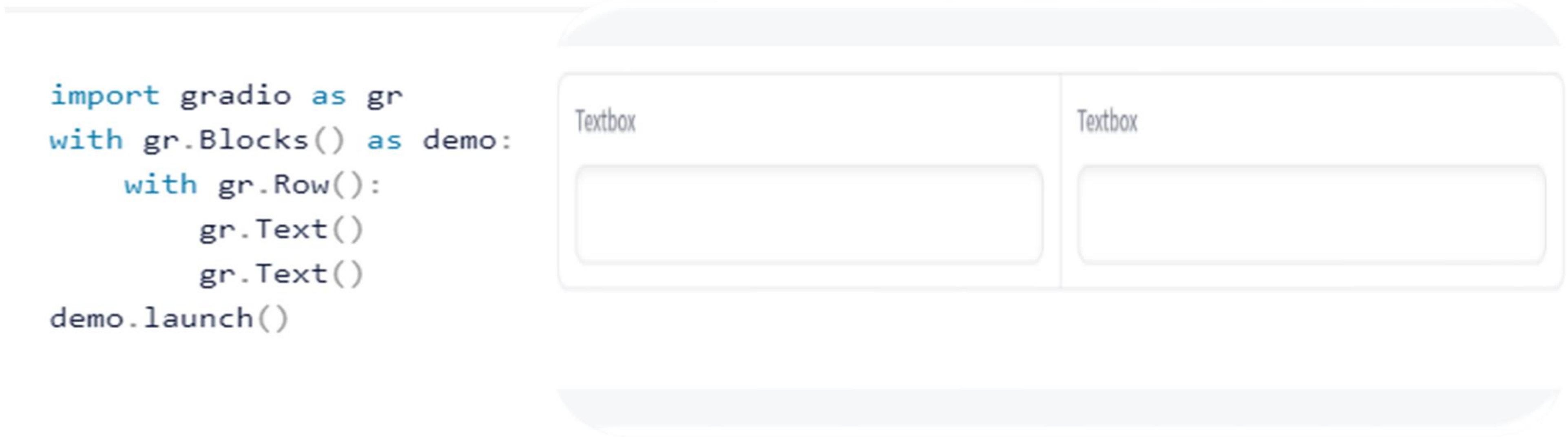


```python
def flip_text(x):
    return x[::-1]


def flip_image(x):
    return np.fliplr(x)


with gr.Blocks() as demo:
    gr.Markdown("Flip text or image files using this demo.")
    with gr.Tab("Flip Text"):
        text_input = gr.Textbox()
        text_output = gr.Textbox()
        text_button = gr.Button("Flip")
    with gr.Tab("Flip Image"):
        with gr.Row():
            image_input = gr.Image()
            image_output = gr.Image()
        image_button = gr.Button("Flip")

    with gr.Accordion("Open for More!"):
        gr.Markdown("Look at me...")

    text_button.click(flip_text, inputs=text_input, outputs=text_output)
    image_button.click(flip_image, inputs=image_input, outputs=image_output)

demo.launch()
```

Laying out your Gradio application

The layout of the application's blocks can be customized using layout classes like :

- **gradio.Row(),**

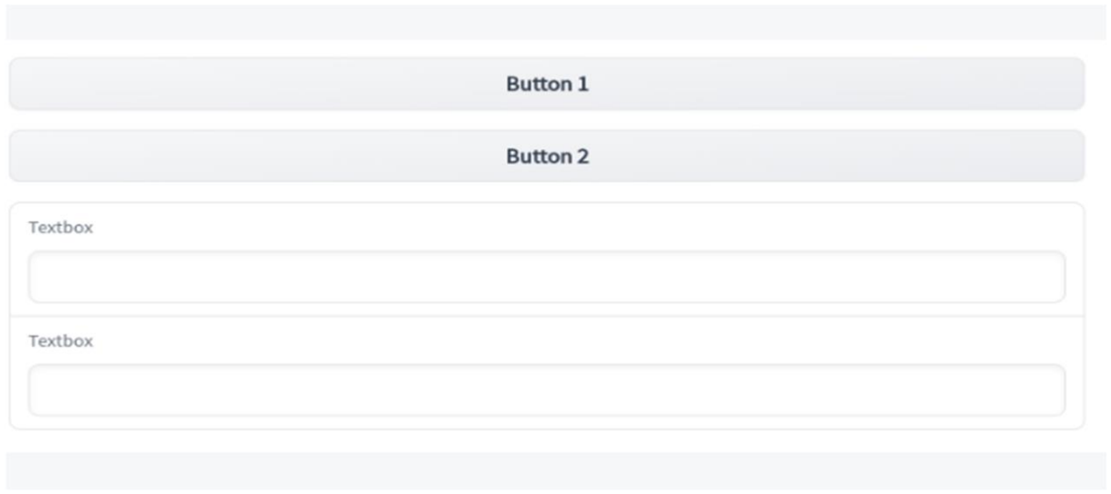


```
import gradio as gr
with gr.Blocks() as demo:
    with gr.Row():
        gr.Text()
        gr.Text()
demo.launch()
```

Laying out your Gradio application

- gradio.Columns(),

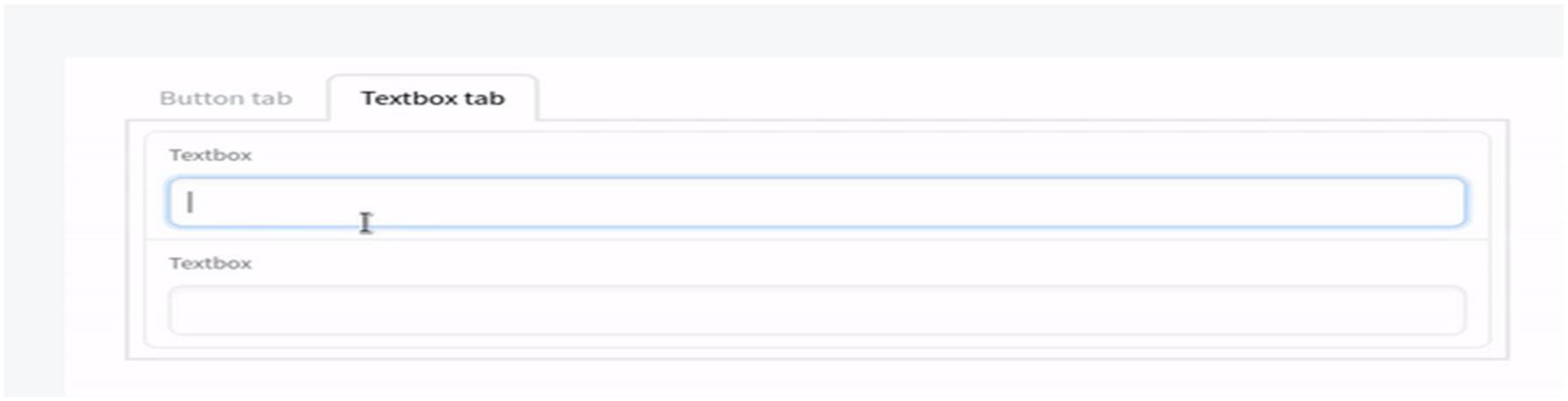
```
import gradio as gr
with gr.Blocks() as demo:
    with gr.Column(scale=2):
        btn1 = gr.Button("Button 1")
        btn2 = gr.Button("Button 2")
    with gr.Column(scale=1):
        text1 = gr.Textbox()
        text2 = gr.Textbox()
demo.launch()
```



Laying out your Gradio application

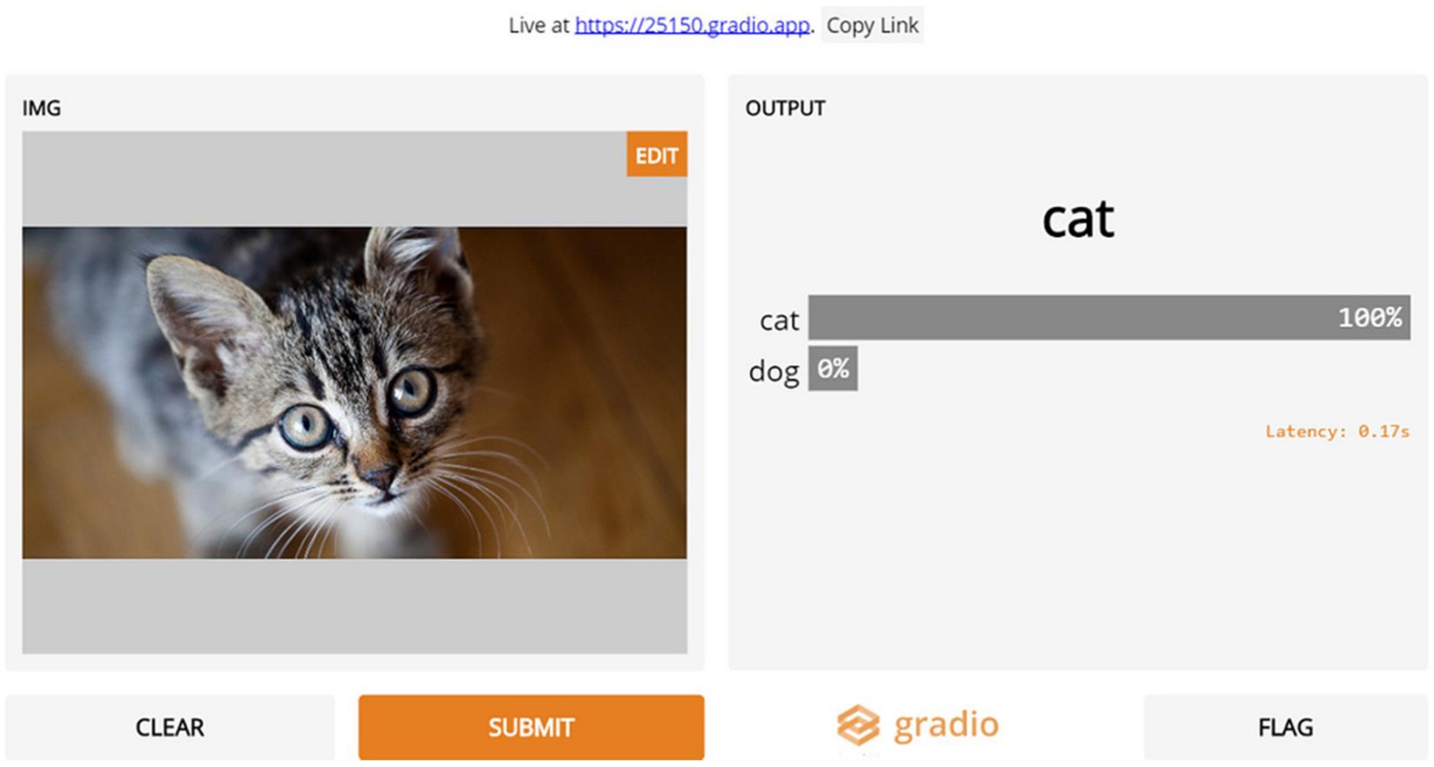
- gradio.Tab(),

```
import gradio as gr
with gr.Blocks() as demo:
    with gr.Tab(label = "Button tab"):
        btn1 = gr.Button("Button 1")
        btn2 = gr.Button("Button 2")
    with gr.Tab(label = "Textbox tab"):
        text1 = gr.Textbox()
        text2 = gr.Textbox()
demo.launch()
```



Connecting Model to Gradio

Cat-Dog Image classification

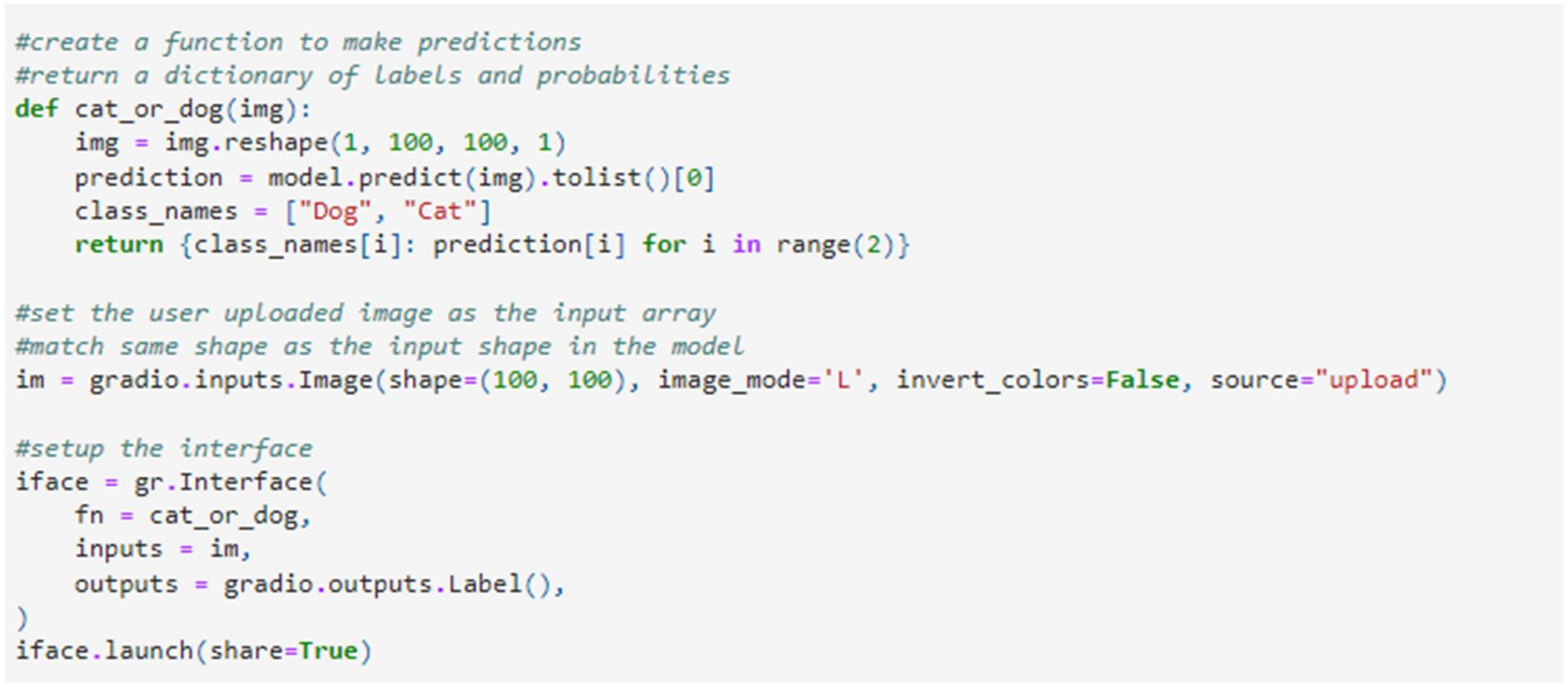


cat

```
cat  100%
dog  0%
```

Latency: 0.17s

Cat-Dog Image classification



```python
#create a function to make predictions
#return a dictionary of labels and probabilities
def cat_or_dog(img):
    img = img.reshape(1, 100, 100, 1)
    prediction = model.predict(img).tolist()[0]
    class_names = ["Dog", "Cat"]
    return {class_names[i]: prediction[i] for i in range(2)}

#set the user uploaded image as the input array
#match same shape as the input shape in the model
im = gradio.inputs.Image(shape=(100, 100), image_mode='L', invert_colors=False, source="upload")

#setup the interface
iface = gr.Interface(
    fn = cat_or_dog,
    inputs = im,
    outputs = gradio.outputs.Label(),
)
iface.launch(share=True)
```

Deploy a Machine Learning Model on Gradio

Images
&
Computer Vision Demo

- Image segmentation
- Image generation

Audio
&
Speech

- Text to Speech
- Speech to text
- Speaker verification

Hosting on a local server

1. **Install Gradio and the dependencies for your application**

2. **Create a file called \`app.py\` that contains your Gradio application**

3. **Run the following command to start a Gradio server on your local machine:**

   gradio run app.py

   \`http://localhost:8000\`

Hosting on a remote server

1. Rent a server from a cloud provider such as AWS, Azure, or Google Cloud Platform.

2. Install Gradio and the dependencies for your application on the remote server.

3. Create a `Dockerfile` that describes how to build and deploy your application.

4. Build the Docker image by running the following command:

```
docker build -t my-app
```

5. Deploy the Docker image to your remote server by running the following command:

```
docker run -p 8000:8000 my-app
```

`http://<your_server_ip>:8000`

Best practices for deployment

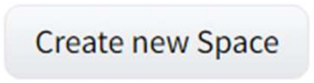
1. Use a cloud deployment platform.
2. Use a containerization tool. Like Docker
3. Use a continuous integration and continuous delivery (CI/CD) pipeline.
4. Test your application thoroughly.
5. Monitor your application
6. Keep your application up to date

Deployment Sample 1/5

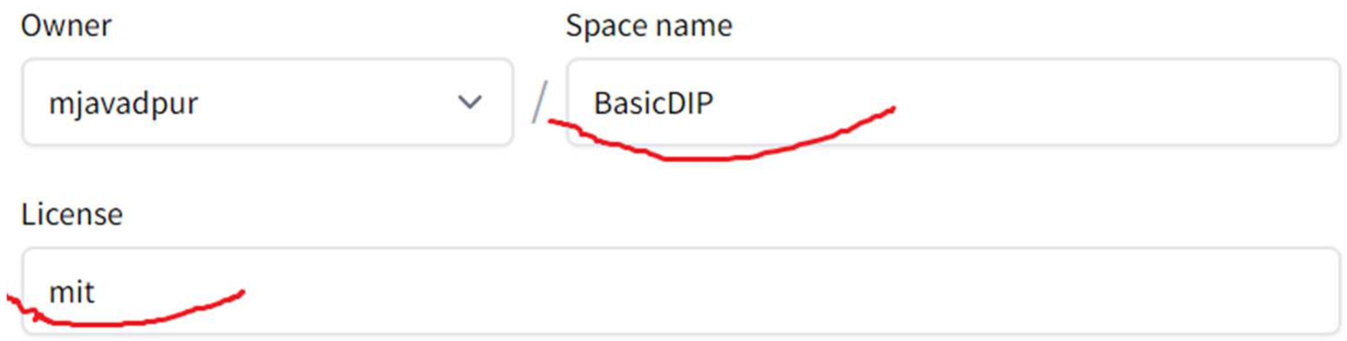
1 Login on https://huggingface.co/

2 Go to: https://huggingface.co/spaces

3 Click on “Create new Space”



4 Write Space name and Select License



Owner: mjavadpur

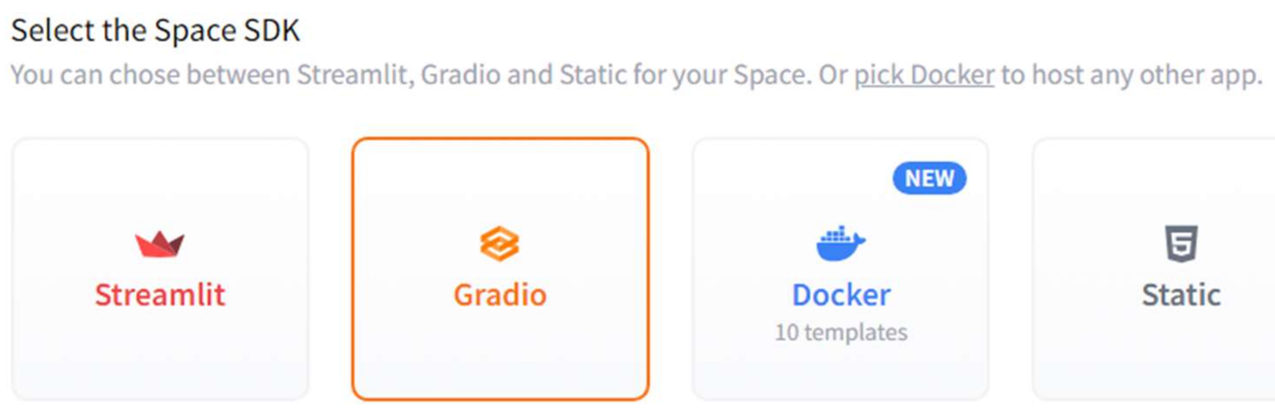
Space name: BasicDIP

License: mit

Deployment Sample 2/5

5

Select Gradio as SDK

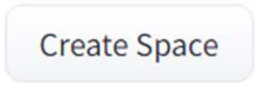


Select the Space SDK

You can chose between Streamlit, Gradio and Static for your Space. Or pick Docker to host any other app.

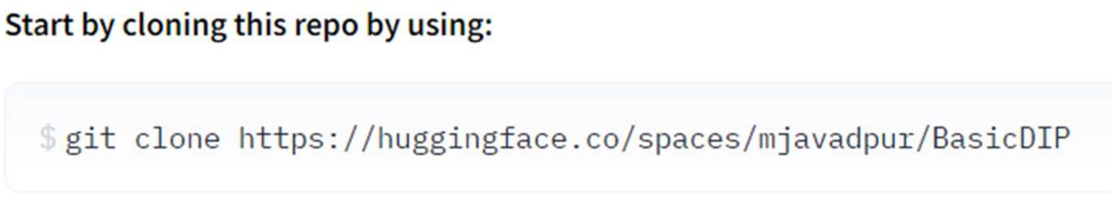
6

Then click on Create Space button



Deployment Sample 3/5

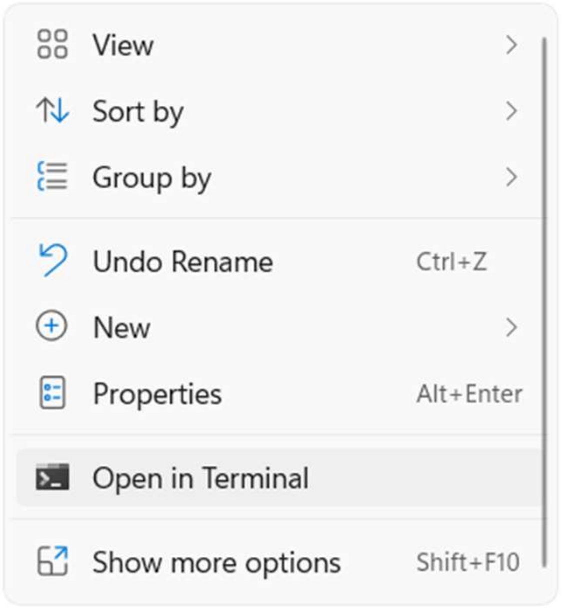
7 Copy the git code



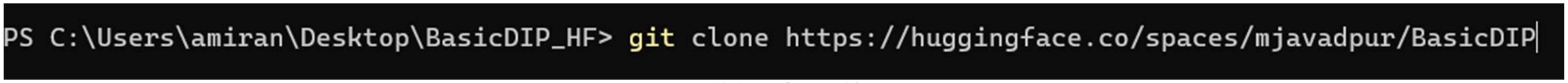
Start by cloning this repo by using:

```
$ git clone https://huggingface.co/spaces/mjavadpur/BasicDIP
```

8 Write click on project folder in your computer and click on "Open in Terminal"



9 Paste git code and enter

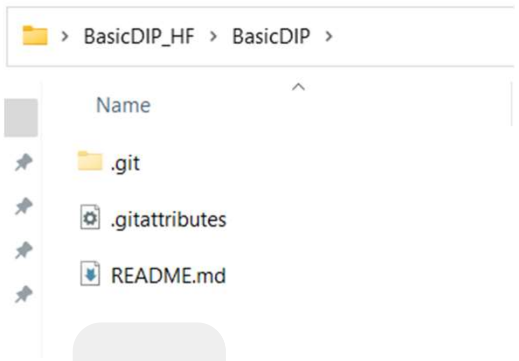


```
PS C:\Users\amiran\Desktop\BasicDIP_HF> git clone https://huggingface.co/spaces/mjavadpur/BasicDIP
```

Deployment Sample 4/5

10

Folder and files cloned:



BasicDIP_HF › BasicDIP ›

Name

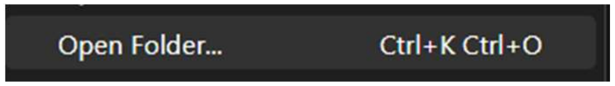
.git

.gitattributes

README.md

11

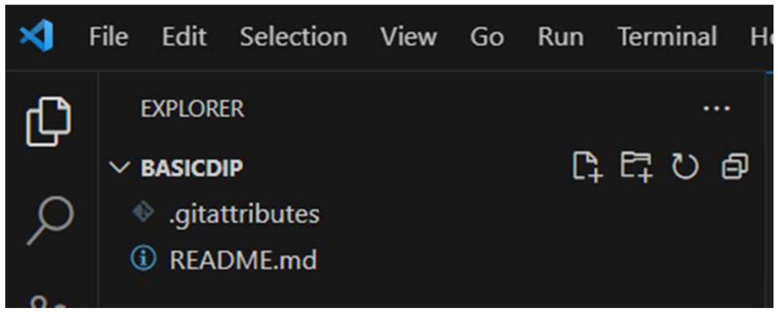
In VSCode select File->Open Folder



Open Folder... Ctrl+K Ctrl+O

12

Select project folder on local to open like this

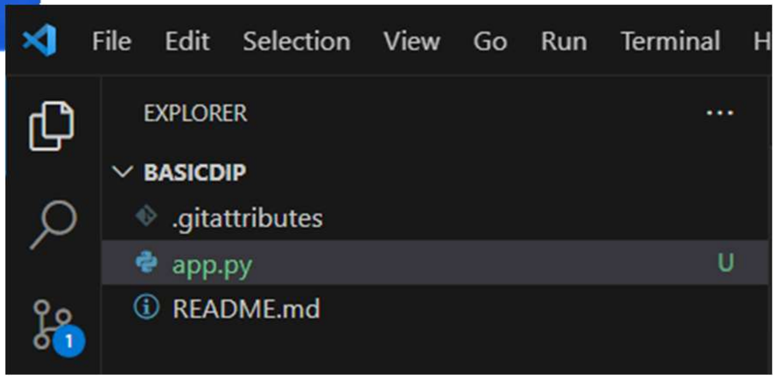


.gitattributes

README.md

13

Project entry point is app.py



.gitattributes

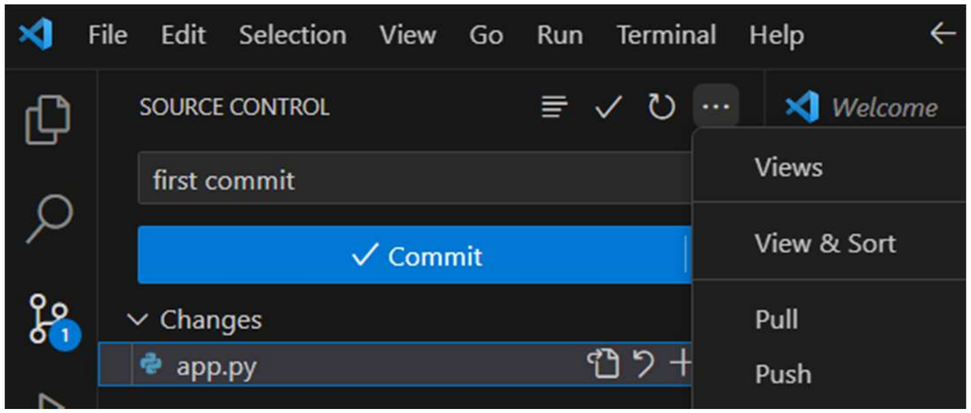
app.py

README.md

Deployment Sample 5/5

14 After debug, Run and Test your codes

15 Commit and Push codes
In first time, Request
HF username and
password to push

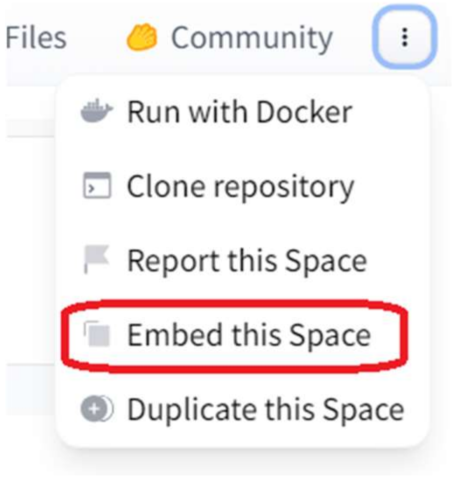


16 Now go to your project address on HF and use your app.

Embedding Hosted Spaces 1/4

Once you have hosted your app on Hugging Face Spaces (or on your own server), you may want to embed the demo on a different website, such as your blog or your portfolio.

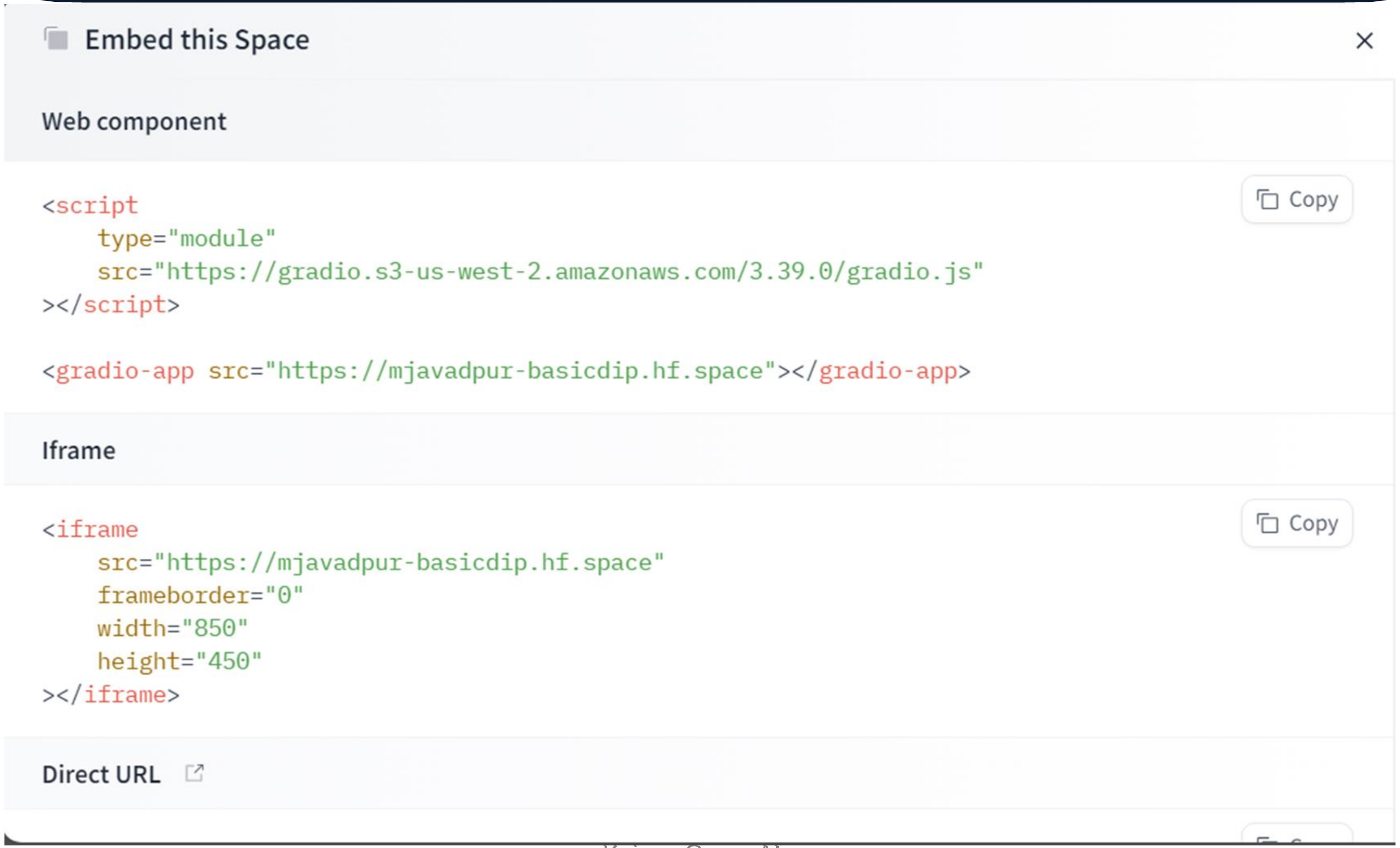
1 You can find quick links to both options directly on the Hugging Face Space page, in the "Embed this Space" dropdown option



2 There are two ways to embed your Gradio demos.

1. Embedding with Web Components
2. Embedding with IFrames

Embedding Hosted Spaces 2/4



Embed this Space

Web component

```html
<script
    type="module"
    src="https://gradio.s3-us-west-2.amazonaws.com/3.39.0/gradio.js"
></script>

<gradio-app src="https://mjavadpur-basicdip.hf.space"></gradio-app>
```

Iframe

```html
<iframe
    src="https://mjavadpur-basicdip.hf.space"
    frameborder="0"
    width="850"
    height="450"
></iframe>
```

Direct URL

1. Embedding with Web Components 3/4

1 Import the gradio JS library into into your site by adding the script below in your site (replace 3.39.0 in the URL with the library version of Gradio you are using).

```
<script type="module"
src="https://gradio.s3-us-west-2.amazonaws.com/3.39.0/gradio.js">
</script>
```

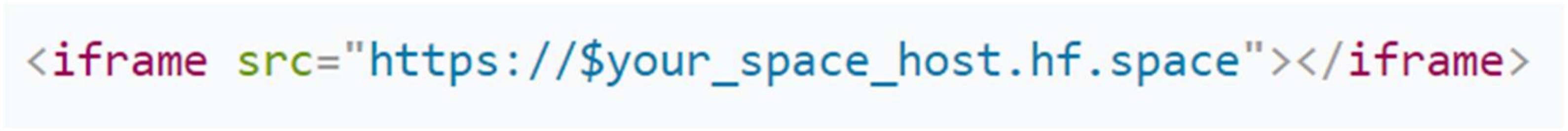
2 Add below code into element where you want to place the app :

```
<gradio-app src="https://$your_space_host.hf.space"></gradio-app>
```

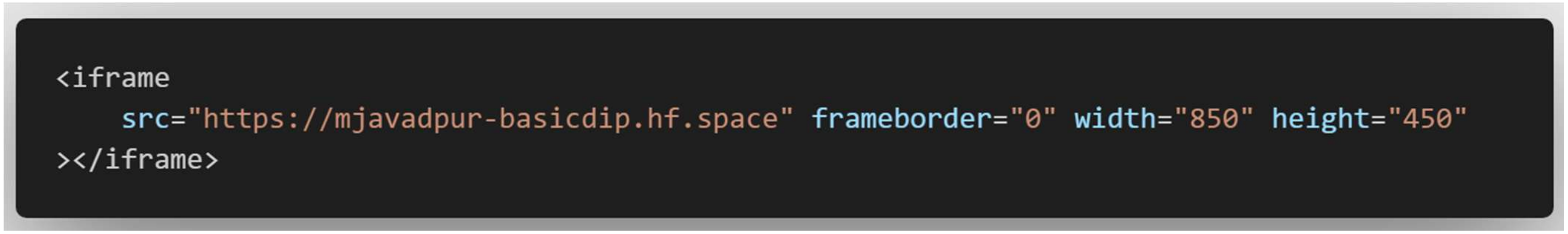
```
<iframe
    src="https://mjavadpur-basicdip.hf.space"
    frameborder="0"
    width="850"
    height="450"
></iframe>
```

2. Embedding with IFrames 4/4

To embed with IFrames instead (if you cannot add javascript to your website, for example), add this element



```
<iframe src="https://$your_space_host.hf.space"></iframe>
```

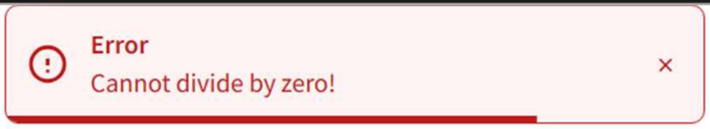


```
<iframe
    src="https://mjavadpur-basicdip.hf.space" frameborder="0" width="850" height="450"
></iframe>
```

Handling errors and debugging

1 Gradio.Error

```
raise gr.Error("Cannot divide by zero!")
```



Error
Cannot divide by zero!

2 Warning

```
gr.Warning("Cannot divide by zero!")
```

3 Info

```
gr.Info("Cannot divide by zero!")
```

Scaling Gradio applications

1. Use a **hosting service** like Heroku or AWS Elastic Beanstalk.
These services will automatically scale your application up or down depending on the number of users.

2. Use a **distributed computing framework** like Ray Serve.
Ray Serve can parallelize the inference requests for your model, which can significantly improve the performance of your application.

3. Use a **caching mechanism** to store the results of previous inference requests.
This can help to reduce the load on your model and improve the performance of your application.

4. Use a **load balancer** to distribute the traffic to your application across multiple servers. This can help to improve the performance of your application by reducing the latency of individual requests.

Working with Gradio API

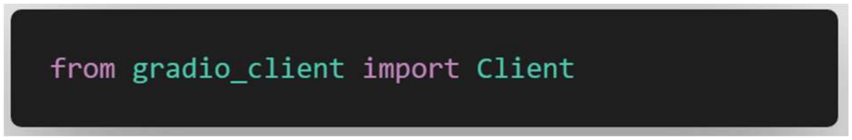
The Gradio client(gradio_client) makes it very easy to use any Gradio app as an API.

1

Client:

The main Client class for the Python client.

This class is used to connect to a remote Gradio app and call its API endpoints.

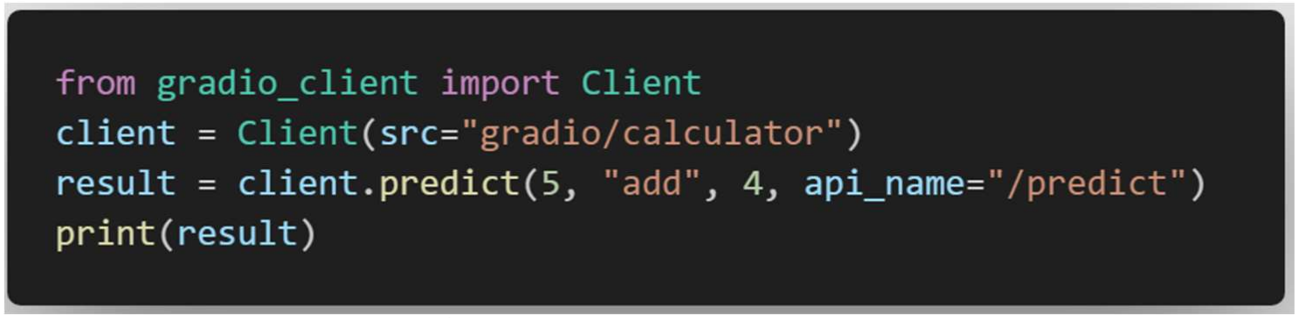


```python
from gradio_client import Client
```

2

Predict:

Calls the Gradio API and returns the result .

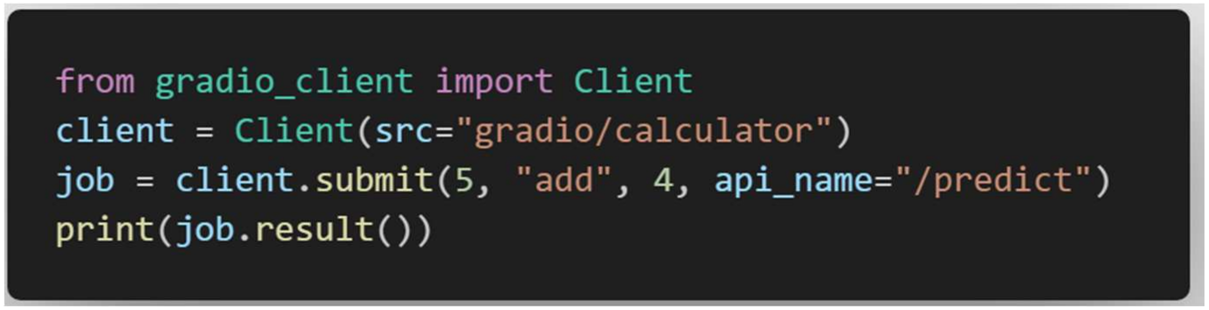


```python
from gradio_client import Client
client = Client(src="gradio/calculator")
result = client.predict(5, "add", 4, api_name="/predict")
print(result)
```

Working with Gradio API

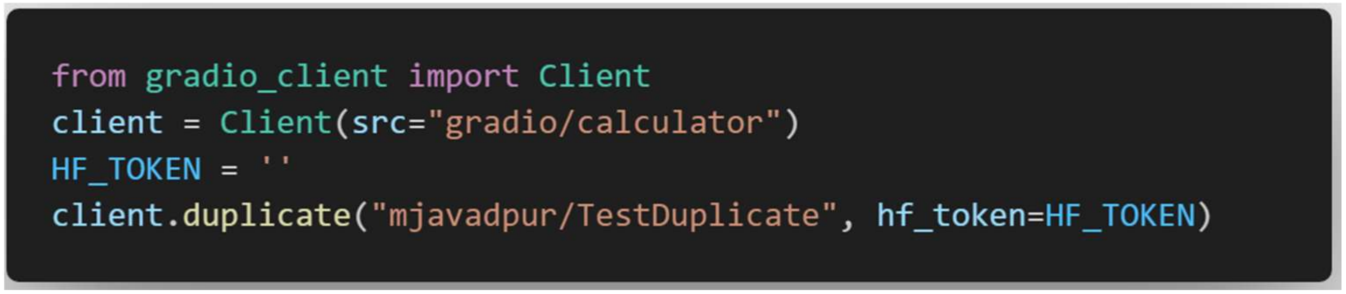
3

submit:
Creates and returns a Job object which calls the Gradio API in a background thread.


```
from gradio_client import Client
client = Client(src="gradio/calculator")
job = client.submit(5, "add", 4, api_name="/predict")
print(job.result())
```

4

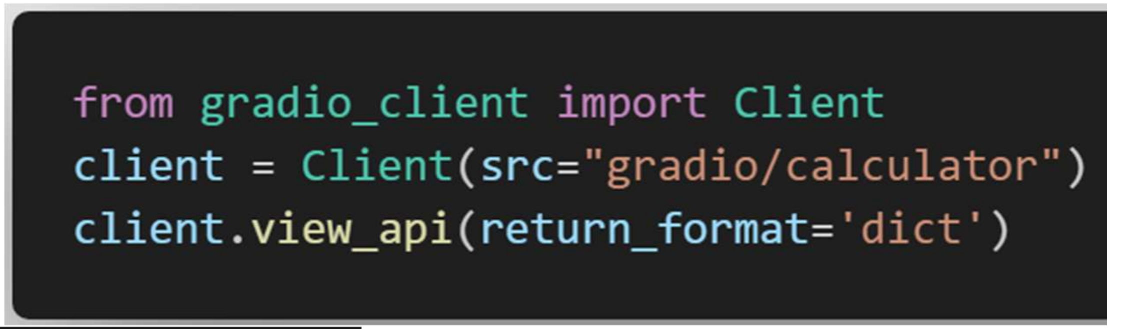
duplicate:
Duplicates a Hugging Face Space under your account and returns a Client object for the new Space.


```
from gradio_client import Client
client = Client(src="gradio/calculator")
HF_TOKEN = ''
client.duplicate("mjavadpur/TestDuplicate", hf_token=HF_TOKEN)
```

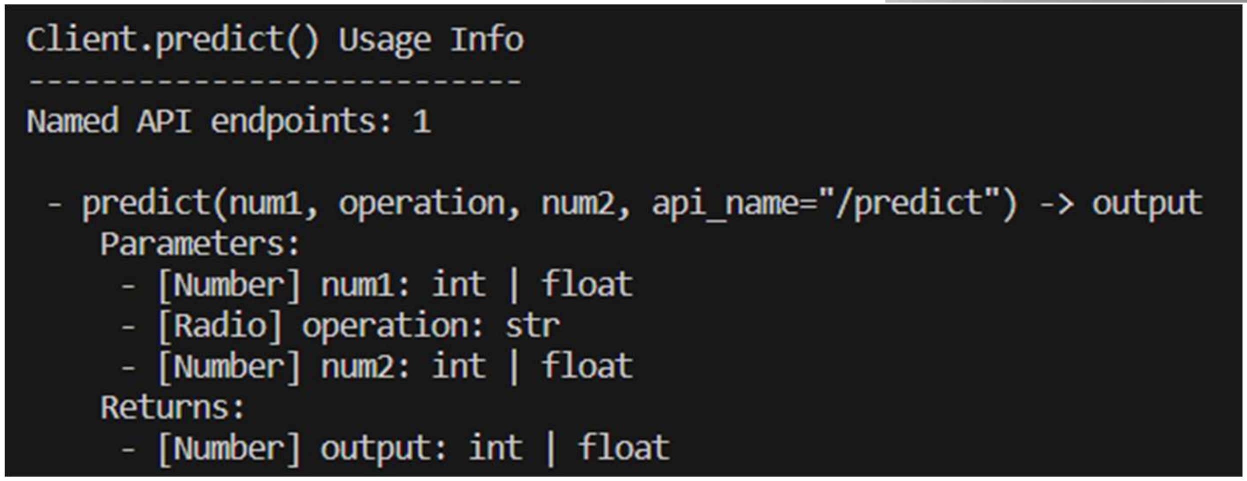
Working with Gradio API

5

view_api:
Prints the usage info for the API.



```python
from gradio_client import Client
client = Client(src="gradio/calculator")
client.view_api(return_format='dict')
```



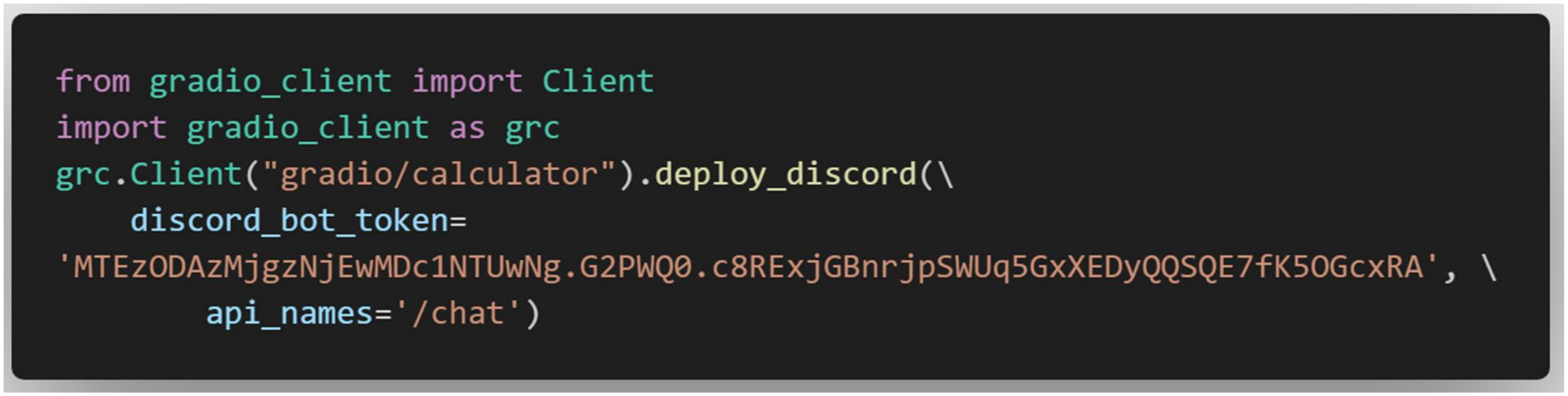
```
Client.predict() Usage Info
---------------------------
Named API endpoints: 1

 - predict(num1, operation, num2, api_name="/predict") -> output
    Parameters:
     - [Number] num1: int | float
     - [Radio] operation: str
     - [Number] num2: int | float
    Returns:
     - [Number] output: int | float
```

Working with Gradio API

6

deploy_discord:

Deploy the upstream app as a discord bot. Currently only supports gr.ChatInterface.



```
from gradio_client import Client
import gradio_client as grc
grc.Client("gradio/calculator").deploy_discord(\
    discord_bot_token=
'MTEzODAzMjgzNjEwMDc1NTUwNg.G2PWQ0.c8RExjGBnrjpSWUq5GxXEDyQQSQE7fK5OGcxRA', \
        api_names='/chat')
```

👍Question?!👍

👍 Thank for Amin and Zekai Group 👍

👍 Thank for Amin and Zekai Group 👍

@zekaigroup