

# AI-tools with LLMs

How to Gradio

# Who are we ? Let's get acquainted

- Who am I?
  - I am Rolf Guescini
- What do you do?
  - I work as a developer and information architect at “Computer and Medienservice” at Humboldt-Universität zu Berlin
- Why are you here today?
  - I am interested in seeing what tools we might come up with
  - To have fun of course :D

# Prerequisites

- Chat AI API key for <https://chat-ai.academiccloud.de>
- MiniConda
  - Python 3.12 or higher
- PyCharm or PyCharm Community

# MiniConda / Python Environment

- Miniconda is a miniature installation of Anaconda Distribution that includes only **conda**, **Python**, the packages they both depend on, and a small number of other useful packages
- If one needs more packages, the **conda install** command enables **access to thousands of packages available by default in Anaconda's public repo**, or from other channels, like conda-forge or bioconda
- System [Requirements](#): / Older versions Archive: <https://repo.anaconda.com/miniconda/>
  - **Operating system:**
    - Windows 10 or later, 64-bit macOS 10.15+ (for Intel) or 64-bit macOS 11.1+ (for Apple Silicon), or Linux, including Ubuntu, RedHat, CentOS 7+, and others.
  - **System architecture:**  
Windows - 64-bit x86; macOS - 64-bit x86 or Apple Silicon (ARM64); Linux - 64-bit x86, 64-bit aarch64 (AWS Graviton2), or s390x (Linux on IBM Z & LinuxONE).
  - **Minimum 400 MB disk space to download and install**

# MiniConda : Install

- Windows

```
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe -  
o .\miniconda.exe  
start /wait "" .\miniconda.exe /S  
del .\miniconda.exe
```

- Mac

```
mkdir -p ~/miniconda3  
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-arm64.sh -o  
~/miniconda3/miniconda.sh  
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3  
rm ~/miniconda3/miniconda.sh
```

- Linux

```
mkdir -p ~/miniconda3  
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O  
~/miniconda3/miniconda.sh  
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3  
rm ~/miniconda3/miniconda.sh
```

# MiniConda : Install

- Windows
  - After installing, open the “Anaconda Prompt (miniconda3)” program to use Miniconda3.
- Mac, Linux
  - After installing, close and reopen your terminal application or refresh it by running the following command

***source ~/miniconda3/bin/activate***

- Then, initialize conda on all available shells by running the following command:

***conda init --all***


- Check installation:

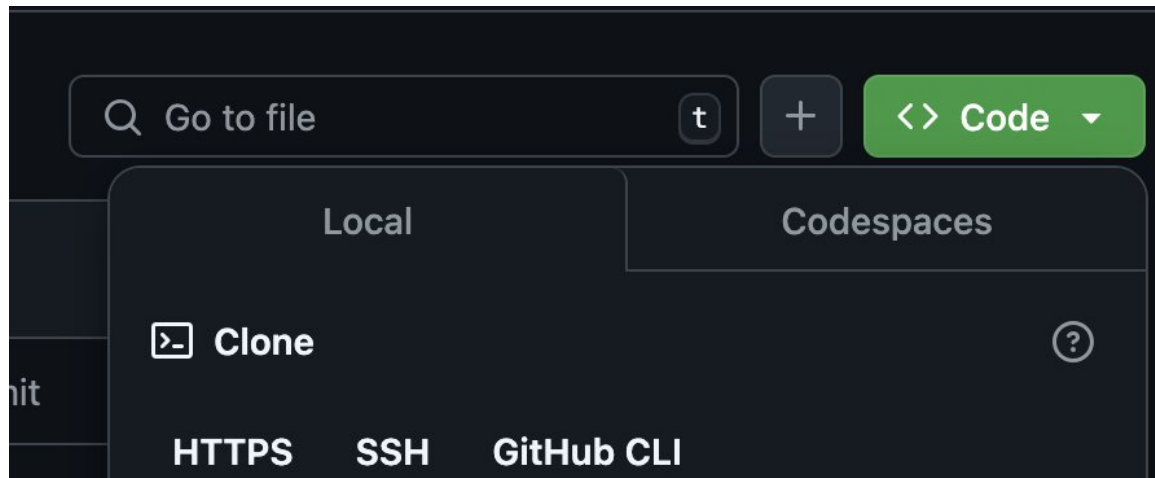
***conda info***

# conda - Python Environment

- Environments in conda are **self-contained, isolated spaces where you can install specific versions of**
  - software packages, including dependencies
  - libraries
  - Python versions.
- This isolation **helps avoid conflicts between package versions** and ensures that your projects have the exact libraries and tools they need.
- **Isolation of dependencies**
  - Environments **isolate software and their dependencies** from the rest of the software installed on your machine
- **Reproducibility**
  - By creating an environment for each project, you can **ensure that your code runs consistently across different machines**


# conda - Python Environment

- When first installing and using conda, all operations are done to the “**base** environment”
  - should only be used for installing anaconda, conda, and conda-related packages
- let us create an environment to contain our hackathon code:
  - go to: <https://github.com/HU-CMS/ki-konkret-hackathon>
  -  Fork click the „Fork“ button to create a duplicate to your own account
  - Clone your repository to your local machine

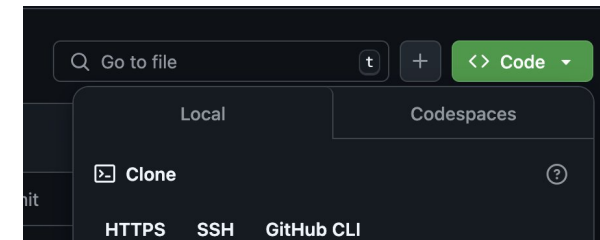




# conda - Python Environment

- When first installing and using conda, all operations are done to the “**base** environment”. This environment is where conda itself is installed, and should only be used for installing anaconda, conda, and conda-related packages
- let us create an environment to contain our hackathon code:
  - go to: <https://github.com/HU-CMS/ki-konkret-hackathon>
  -  click the „Fork“ button to create a duplicate of the repository to your own account
  - Untick ☐ Copy the `main` branch only  
Contribute back to HU-CMS/ki-konkret-hackathon by adding your own branch. [Learn more.](#)
  - Clone your repository to your local machine using the command line:

```
git clone https://github.com/<youruser>/ki-konkret-hackathon.git
```



# conda - Python Environment

- Because environments are isolated spaces, you can only work with one at a time. Selecting an environment to work with is called **activating** it
- Create the environment

```
# Replace <ENV_NAME> with a name for your environment  
# Replace <PACKAGE> with your desired package  
# Replace <VERSION> with your desired version (optional)
```

```
conda create --name <ENV_NAME> <PACKAGE>=<VERSION>
```

```
ex: conda create --name myenv python=3.11 beautifulsoup4 docutils jinja2=3.1.4 wheel
```

- Activate / deactivate the environment

```
# Replace <ENV_NAME> with the name of the environment you want to activate/deactivate
```

```
conda activate <ENV_NAME> / conda deactivate <ENV_NAME>:
```

# conda - Python Environment : packages

- You can use the **conda package manager** to access and install over 8,000 open-source **data science and machine learning conda packages from Anaconda's public repository** and thousands more community-developed packages published on Anaconda.org and conda-forge
- Use the conda **install** command to install packages **into an environment**. If no environment is specified in the command, conda installs the package in the **working environment**
- conda strongly recommends **installing your project packages in a separate conda environment**
  - **protects your base environment from breaking** due to complex dependency conflicts and allows you to easily manage and reproduce your environment on other machines.

# conda - Python Environment : packages

- Search packages

```
conda search <PACKAGE>
```

- Install packages

```
conda install  
<PACKAGE>=<VERSION> <PACKAGE>=<VERSION> <PACKAGE>=<VERSION> --name <ENVIRONMENT>
```

# conda - Python Environment : pip packages

- Most popular packages from the **PyPI repository** are available in either Anaconda's public repository, **anaconda.org**, or **conda-forge**.
  - However, you might need to use **pip package manager** if a package or specific version is not available through conda channels

```
pip install <PACKAGE>
```

- Installing packages using **pip modifies your conda environment**, but **conda is not aware of these modifications**
- As a result, when conda later attempts to modify the environment, there's a **high probability that dependency conflicts will arise** between the conda-tracked packages and the untracked pip packages, which can lead to a broken environment

# conda - Python Environment : **environment.yml**

- To create a stable environment that includes pip packages, Anaconda recommends writing an **environment.yml file and then building an environment from that file.**
- make sure to add **pip** and its dependencies last, since conda builds environments in the order listed
- Syntax example:

**name:** env-name

**channels:**

- conda-forge
- defaults

**dependencies:**

- python=3.7
- codecov

**name:** is the name of the environment

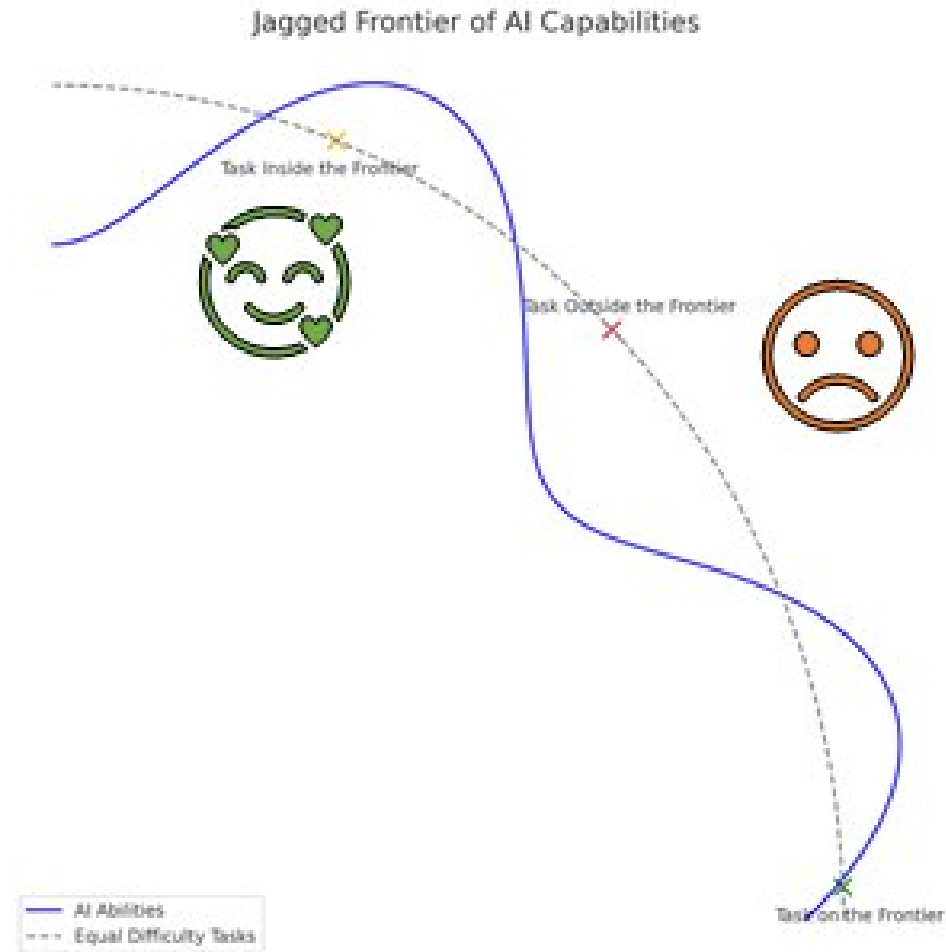
**channels:** are the locations where packages are stored.

**dependencies:** are the packages on which the project is dependent

*By default, packages are automatically downloaded and updated from the **default channel**, which may require a paid license, as described in the repository terms of service. The **conda-forge channel** is free for all to use.*

# Fähigkeiten von genKI

- genKI hat eine sehr unübersichtliche technologische Grenze



Notes: This figure displays the AI frontier as jagged. Tasks with the same perceived difficulty may be on one side or the other of the frontier. ChatGPT produced this image starting from the authors' prompts.

# conda - Python Environment : environment.yml

1. Open and create environment.yml which installs the following packages; make sure to search to see if any need to be installed with pip

- `gradio`
- `gradio_client`
- `transformers`

2. Create your environment using the environment.yml file

```
conda env create --file environment.yml
```

3. Add `torch` and `openai` to your environment file and update the environment

```
conda env update --file environment.yml --prune
```

4. Delete the environment, verify it is deleted

```
conda remove --name ki-konkret-hackathon --all  
conda info --envs
```



# PyCharm a Python IDE

- PyCharm is an IDE that **integrates with IPython Notebook**, has an **interactive Python console**, and **supports Anaconda** as well as multiple scientific packages
- PyCharm also supports creating virtual environments for Python with conda
  - creating a **new conda environment** when you create a **new Python project in PyCharm**
  - **configure an existing conda environment** for a new project, or switch conda environments within a project that already exists

# PyCharm a Python IDE

After opening PyCharm, **click New Project**

In the New Project screen, **create the project name and its location**

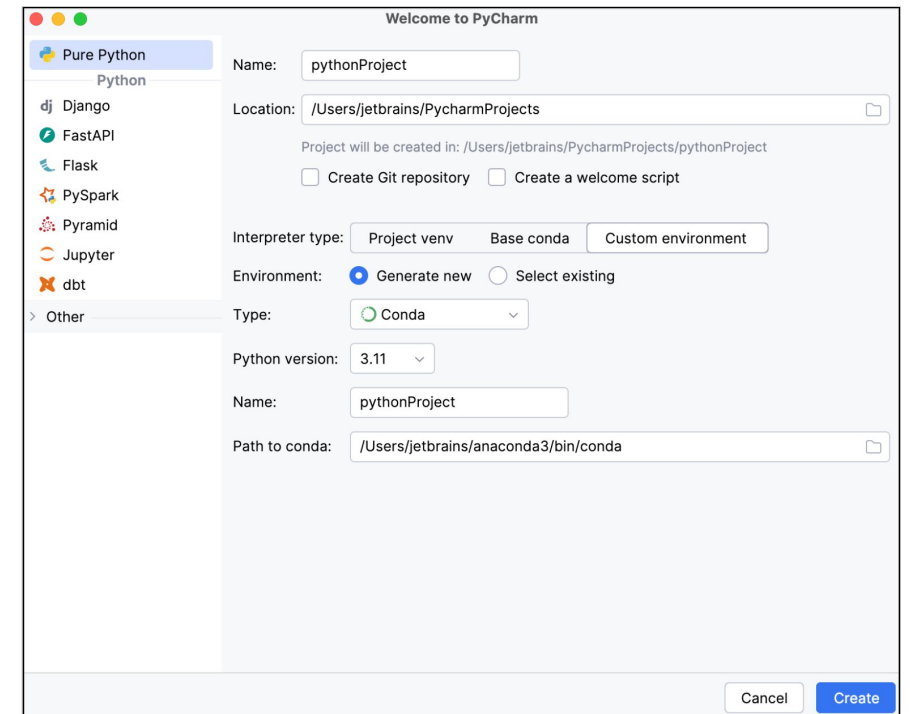
Add a **main.py** file to your project by selecting **Create welcome script**

Select **Custom environment**, then select **Generate New**

From the **Type** dropdown, select **Conda**

From the **Python version dropdown** select the Python version you want

Create your **environment name**.



# Kontext

- Alle eigenen Fragen und Nachfragen
- Alle Antworten der LLM
- Hintergrundanweisungen
- **Die Summe aller Tokens in einer Session**
- Beispiele:
  - GPT-3: Bis zu 2048 Tokens
  - Mistral 7B: Bis zu 8.192 Tokens, Ausgabe bis 4.096
  - GPT-4o: 60.000 bis 128.000 Token, Ausgabe bis 4.096
  - Claude 3.5: Bis zu 200.000 Token, Ausgabe bis zu 8.192
  - LLama 3.1: Bis zu 128.000 Token, Ausgabe 4.096
  - Gemini 1.5 Pro: Bis zu 1 Million Token, Ausgabe 8192

## Rollenmodell

```
System: ... ..  
User: ... ..  
Assistant: ... ..  
User: ... ..  
Assistant: ... ..  
User: ... ..  
Assistant: ... ..
```

## ReAct Agents



Typical use case:

Anything and everything:  
Find a solution to climate change  
Plan a trip for a group of 8 mountaineers to the Rockies



Make their own plan and execute it

Make plan:

1. Google w
2. Do x
3. Query y
4. Do z

Execute plan:



Giant problem / and solution space



AI used for planning and full execution of process



Does not work yet for non-trivial problems

## AI Workers



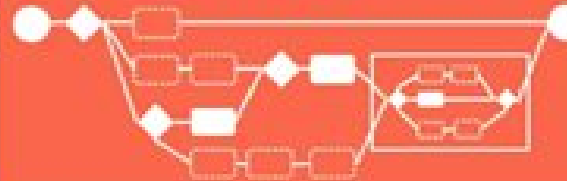
Typical use case:

Recurring, structured processes:  
Write offers to RFPs  
Process insurance cases  
Teach algebra to 5th graders



Follow a predefined workflow

Choose path based on the specific case.



Small problem space, focus on a single problem type



AI used for categorization, data generation, evaluation



Carefully developed, it will work for 60% - 95% of cases

# Präzise Prompts

- Aspekte für den System Prompt
  - Welche Aufgabe soll erledigt werden?
  - Was soll auf keinen Fall passieren?
- 
- In welchem Kontext wird die Aufgabe gestellt?
  - Duktus und Ansprache
  - Mehrsprachigkeit
  - Aktuelles Datum und Datum der Daten
  - Gewünschte Ausgabeformate

# Anthropic System Prompts

- <https://docs.anthropic.com/en/release-notes/system-prompts>

# Templates für die Ausgabe

- Analysiere den folgenden Fehler in meinem Code und liefere einen strukturierten Fehlerbericht nach genau diesem Template:
- # FEHLERANALYSE
- ## FEHLERBESCHREIBUNG
- [1-2 Sätze zum Fehler]
- ## PROBLEMATISCHE ZEILE(N)

Analysiere das folgende Dokument hinsichtlich Rechtschreibung, Stil, Zielgruppeneignung und Klarheit der Darstellung.

Halte dich EXAKT an folgendes Ausgabeformat ohne Abweichungen oder zusätzliche Erklärungen:

# Dokumentanalyse

## Kurzfassung

{{2-3 Sätze zur Gesamteinschätzung des Dokuments}}

## Rechtschreibprüfung

Bewertung: {{Gut/Mittel/Verbesserungswürdig}}

Top-3-Fehler:

- {{Fehler 1}}

- {{Fehler 2}}

- {{Fehler 3}}

## Stilprüfung

Bewertung: {{Gut/Mittel/Verbesserungswürdig}}

Stärken: {{Hauptstärke des Stils}}

Schwächen: {{Hauptschwäche des Stils}}

## Zielgruppenprüfung

Bewertung: {{Gut/Mittel/Verbesserungswürdig}}

Passt für: {{Zielgruppe}}

Anpassungsbedarf: {{Wichtigste Anpassung}}

- Zu analysierendes Dokument:

[Hier folgt der Text des zu analysierenden Dokuments]



# Einzelne Aufgaben

Du bist ein Expertensystem für die Überarbeitung von Texten

Deine Aufgabe ist es, Texte zu erkennen, die überarbeitet werden müssen

Antworte AUSSCHLIESSLICH in diesem JSON-Format:

```
{  
  "should_revise": false,  
  "revise_text": [],  
  "revise_reason": "Begründung"  
}
```

WICHTIGE REGELN

IMMER ÜBERARBEITEN

NIE ÜBERARBEITEN

# Gradio

- open-source Python library that allows you to **quickly create user-friendly, interactive web applications** that allow users to interact with models, data, and other types of Python code
- **Easy Integration:**
  - It's very **easy to integrate with any model**, whether it's from libraries like TensorFlow, PyTorch, Hugging Face Transformers, or even your own custom models
- **Interactive Interfaces:**
  - Gradio **automatically generates interactive interfaces** like textboxes, buttons, image uploaders, sliders, and more, depending on the input/output type of your model
- **Sharing and Collaboration:**
  - Gradio allows you to launch a web-based demo with just one line of code. You can **share** this demo with others by simply **sending them a link, enabling real-time collaboration or feedback**

# the `gr.Interface` class

- high-level abstraction in Gradio that allows you to quickly create a demo for any Python function simply by specifying the input types and the output types
- the `Interface` class is initialized with three required parameters:
- **fn:**
  - the **function** to wrap a user interface (UI) around
- **inputs:** which Gradio component(s) to use for the input.
  - The number of components should match the number of arguments in your function.
- **outputs:** which Gradio component(s) to use for the output.
  - The number of components should match the number of return values from your function

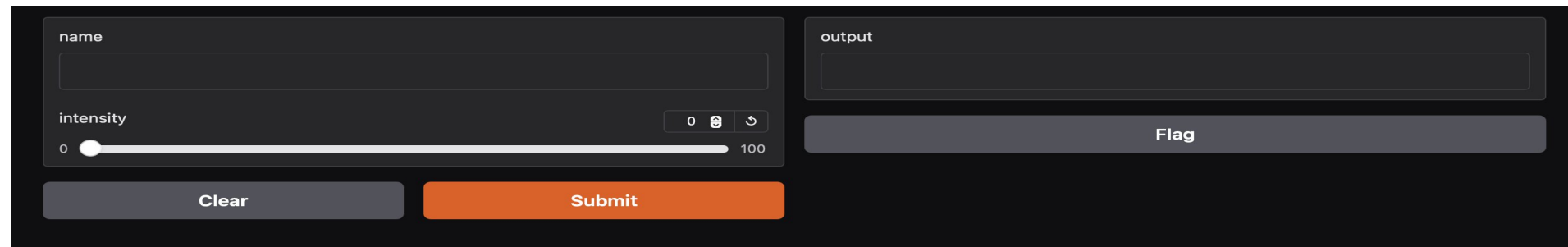
# gr.Interface: Hello World

```
import gradio as gr

def greet(name, intensity):
    return "Hello, " + name + "!" * int(intensity)

demo = gr.Interface(
    fn=greet,
    inputs=["text", "slider"],
    outputs=["text"]
)
```

[demo.launch\(\)](#)



The image shows a Gradio web interface. On the left, there are two input fields: 'name' (a text box) and 'intensity' (a slider ranging from 0 to 100). Below these inputs are two buttons: 'Clear' (grey) and 'Submit' (orange). On the right, there is an 'output' text area and a 'Flag' button (grey).

# Gradio Components

- Gradio includes more than 30 pre-built components (as well as many community-built custom components) that can be used as inputs or outputs in your demo

- These components correspond to common data types in machine learning and data science, i.e:

- the **gr.Image** component is designed to handle input or output images

- the **gr.Label** component displays classification labels and probabilities

- the **gr.LinePlot** component displays line plots

- the **gr.Dataset** component Creates a gallery or table to display data samples



# Component Attributes

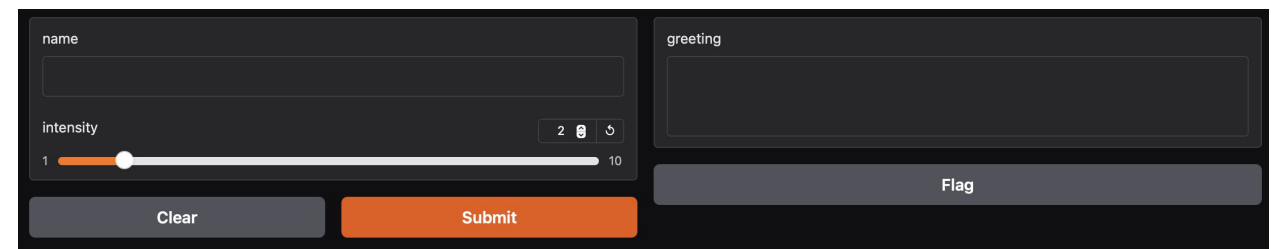
- In Hello world, we used the **default versions** of the `gr.Textbox` and `gr.Slider` with default attributes
- using the **actual classes** for `gr.Textbox` and `gr.Slider` instead of the string shortcuts, you have access to much more **customizability through component attributes**

```
import gradio as gr

def greet(name, intensity):
    return "Hello, " + name + "!" * intensity

demo = gr.Interface(
    fn=greet,
    inputs=["text",
            gr.Slider(value=2, minimum=1,
                      maximum=10, step=1)
            ],
    outputs=[gr.Textbox(
        label="greeting", lines=3)
            ] )

demo.launch()
```



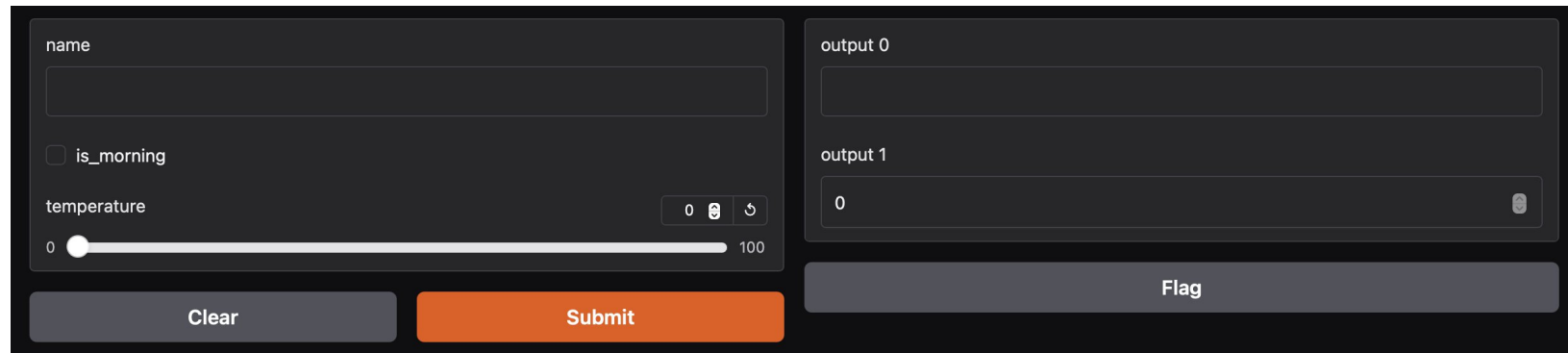
The screenshot shows the Gradio web interface. On the left, there is a 'name' text input field and an 'intensity' slider. The slider has a range from 1 to 10, with a current value of 2. Below the slider are 'Clear' and 'Submit' buttons. On the right, there is a 'greeting' text area. At the bottom right, there is a 'Flag' button.

# Multiple Input and Output Components

```
import gradio as gr

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()
```



The image shows a Gradio web interface for the 'greet' function. It features a dark-themed layout with the following components:

- name:** A text input field.
- is\_morning:** A checkbox.
- temperature:** A slider control ranging from 0 to 100, with a current value of 0.
- output 0:** A text output field.
- output 1:** A numerical output field showing the value 0.
- Buttons:** A 'Clear' button, a 'Submit' button, and a 'Flag' button.

# Component example

- Gradio supports many types of components, such as **Image**, **DataFrame**, **Video**, or **Label**
- When using the Image component as input, your function will receive a **NumPy array with the shape** (height, width, 3), where the last dimension represents the RGB values.
- We'll **return** an image as well in the form of a **NumPy array**.
- Gradio handles the preprocessing and postprocessing to convert images to NumPy arrays and vice versa

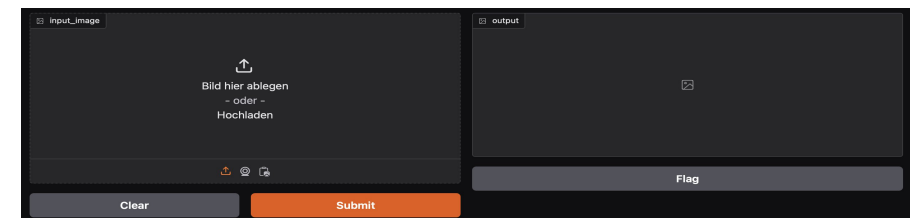
```
import numpy as np
import gradio as gr

def sepia(input_image):
    sepia_filter = np.array([
        [0.393, 0.769, 0.189],
        [0.349, 0.686, 0.168],
        [0.272, 0.534, 0.131]
    ])
    sepia_image = input_image.dot(sepia_filter.T)
    # sepia_image = sepia_image / sepia_image.max()
    sepia_image /= sepia_image.max()

    return sepia_image

demo = gr.Interface(sepia, gr.Image(), "image")

demo.launch()
```





# Example Inputs

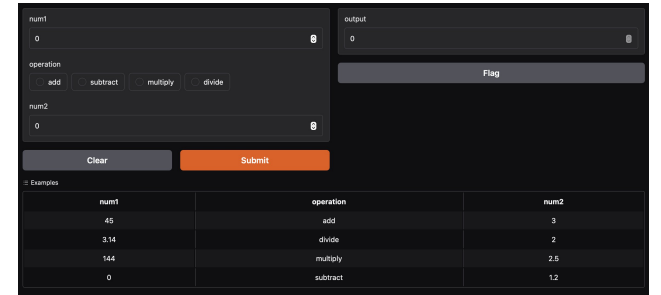
- You can provide example data that a user can easily load into Interface.
- This can be helpful to **demonstrate the types of inputs the model expects**, as well as to provide a way to **explore your dataset** in conjunction with your model.
- To load example data, you can provide a **nested list** to the **examples=** keyword argument of the Interface constructor
- The format of example data for each component is specified in the [Docs](#)

# Example Inputs : Toy Calculator

```
import gradio as gr

def calculator(num1, operation, num2):
    if operation == "add":
        return num1 + num2
    elif operation == "subtract":
        return num1 - num2
    elif operation == "multiply":
        return num1 * num2
    elif operation == "divide":
        if num2 == 0:
            raise gr.Error("Cannot divide by zero!")
        return num1 / num2

demo = gr.Interface(
    calculator,
    [
        "number",
        gr.Radio(["add", "subtract",
                  "multiply", "divide"]),
        "number"
    ],
    "number",
    examples=[
        [45, "add", 3],
        [3.14, "divide", 2],
        [144, "multiply", 2.5],
        [0, "subtract", 1.2],
    ],
    title="Example Calculator",
    description="Here's a sample calculator.",
)
demo.launch()
```



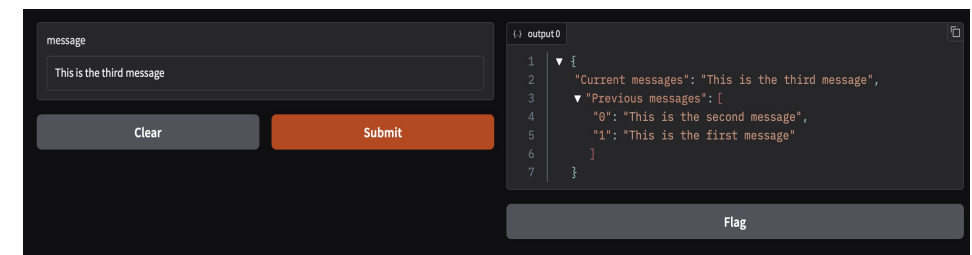
# Interface State

- There are two approaches to state in Gradio: global state and session state
- **Global State**
  - If the state is something that should be **accessible to all function calls** and **all users**, you can create a variable outside the function call and access it inside the function
- **Session State**
  - Gradio supports is session state, where **data persists across multiple submits within a page session**. However, data is not shared between different users of your model

# Interface State: Session state

- To store data in a session state, you need to do three things:
  1. Pass in an **extra parameter into your function, which represents the state** of the interface.
  2. At the end of the function, **return the updated value of the state as an extra return value**
  3. **Add the 'state' input and 'state' output components** when creating your Interface

```
def store_message(message: str, history: list[str]):  
    output = {  
        "Current messages": message,  
        "Previous messages": history[:-1]  
    }  
  
    history.append(message)  
  
    return output, history  
  
demo = gr.Interface(  
    fn=store_message,  
    inputs=["textbox", gr.State(value=[])],  
    outputs=["json", gr.State()] )  
  
demo.launch()
```



# Blocks and block layout

- First, note the **with gr.Blocks() as demo:** clause
- The Blocks app code will be contained within this clause
- Next come the Components
- Components are automatically added to the Blocks as they are created within the with clause
- By **default**, components in Blocks are arranged **vertically**
  - with **gr.Row()** to arrange **vertically**
  - with **gr.Column()** to arrange **horizontally**
- Under the hood, this layout structure uses the **flexbox** model of web development

```
with gr.Blocks() as demo:
    with gr.Blocks() as demo:
        with gr.Row():
            name = gr.Textbox(label="Name")
            output = gr.Textbox(label="Output Box")
        with gr.Column(scale=1, min_width=300):
            greet_btn = gr.Button("Greet")
            _name = gr.Textbox(label="Name")
        demo.launch()
        with gr.Column(scale=1, min_width=300):
            age = gr.Textbox(label="Name")
            with gr.Row():
                output = gr.Textbox(label="Output Box")
            with gr.Row():
                greet_btn = gr.Button("Greet")

demo.launch()
```

Name	Birth year
<input type="text"/>	<input type="text"/>
Output Name	Output Year
<input type="text"/>	<input type="text"/>
<input type="button" value="GO!"/>	

# Blocks and event listeners

- Components also come with certain events that they support. These are **methods that are triggered with user actions**
- **Event listeners define the data flow within the app**
- an Event listener **can take multiple inputs or outputs**, as a **dictionary** or a **list**
  - *The callback function must, in the case of multiple outputs, return a list or dictionary of VALUES for the components*
  - One can also return a new component in case one wants to change the configuration of the component
- Finally, the **click() event listener**

```
with gr.Blocks() as demo:
    with gr.Row():
        with gr.Column(scale=1,min_width=300):
            name = gr.Textbox(label="Name")
        with gr.Column(scale=1,min_width=300):
            age = gr.Textbox(label="Age")
    with gr.Row():
        output = gr.Textbox(label="Output Box")
    with gr.Row():
        greet_btn = gr.Button("Greet")

    greet_btn.click(
        fn=greet,
        inputs=[name, age],
        outputs=output
    )
```

`demo.launch()`

# Blocks and event listeners: I / O

- The inputs and outputs between components and events are basically of one of two formats

```
def greet(name_string, year_string):  
    return [name_string, year_string]
```

- A list of arguments

```
greet_btn.click(  
    fn=greet,  
    inputs=[name, age],  
    outputs=[output_name, output_year, output_age])
```

- A **dictionary** keyed by the **component name**

```
def greet(data):  
    return {output_name: data[name], output_year: data[age]}
```

- Dictionary returns are helpful when an event listener affects many components on return, or conditionally affects outputs and not others

```
def greet(data):  
    if data[age] is None:  
        return {output_name: data[name]}  
    else:  
        return {output_name: data[name],  
                output_year: data[age]}
```

- Using keyed dictionaries for instance, gives us the possibility of not updating some components

# Types of Event Listeners

- Components support different event listeners , and event listener parameters i.e:
  - Audio
    - stream
    - play
    - pause
    - start\_recording
  - **Image**
    - clear
    - upload
- many are common with mutiple components
  - click
  - select
  - double\_click
  - change
  - submit
  - focus
  - blur
  - key\_up



# Types of Event Listeners:EventData helper

- When **gr.EventData** or one of its subclasses is **added as a type hint** to an argument of a function, a **gr.EventData object will automatically be passed as the value of that argument.**
- The attributes of this object **contains information about the event that triggered the listener.**
- The gr.EventData object itself contains a **.target** attribute that refers to the component that triggered the event
- subclasses of gr.EventData contains additional attributes that are different for each class.

```
import gradio as gr

with gr.Blocks() as demo:
    images = [("images/cat.jpg", "Cat"), ("images/dog.jpg",
"Dog"), ("images/cat2.jpg", "Cat too"), ("images/dog2.jpg",
"Dog too")]

    table = gr.Dataframe([[1, 2, 3], [4, 5, 6]])

    gallery = gr.Gallery( images,  object_fit="contain",
height="auto")

    textbox = gr.Textbox("Hello Cats & Dogs!")

    statement = gr.Textbox()

    def on_select(value, evt: gr.EventData):
        return f"The {evt.target} component was selected, and
its value was {value}."

    table.select(on_select, table, statement)
    gallery.select(on_select, gallery, statement)
    textbox.select(on_select, textbox, statement)

demo.launch()
```

# Types of Event Listeners: on (.....)helper

- Sets up an event listener that triggers a function when the specified event(s) occur.
- This is especially useful **when the same function should be triggered by multiple events**

One can use **decorator syntax** as well:

```
import gradio as gr

with gr.Blocks() as demo:
    with gr.Row():
        input = gr.Textbox()
        button = gr.Button("Submit")
    output = gr.Textbox()
    gr.on(
        triggers=[button.click,
                  input.submit],
        fn=lambda x: x,
        inputs=[input],
        outputs=[output]
    )

demo.launch()
```

```
import gradio as gr

with gr.Blocks() as demo:
    name = gr.Textbox(label="Name")
    output = gr.Textbox(label="Output Box")
    greet_btn = gr.Button("Greet")

    @gr.on(
        triggers=[name.submit,
                  greet_btn.click],
        inputs=name,
        outputs=output)
    def greet(name):
        return "Hello " + name + "!"

demo.launch()
```

# Multiple Data Flows

- A Blocks app is **not limited to a single data flow** the way Interfaces are
- As apps get more complex, many data flows connect various Components
- An example, could be a demo where **the output of one model (a speech-to-text model) gets fed into the next model** (a sentiment classifier)

# Multiple Data Flows

```
from transformers import pipeline
import gradio as gr

asr = pipeline("automatic-speech-recognition",
               "facebook/wav2vec2-base-960h")
classifier = pipeline("text-classification")

def speech_to_text(speech):
    text = asr(speech)["text"]
    return text

def text_to_sentiment(text):
    return classifier(text)[0]["label"]

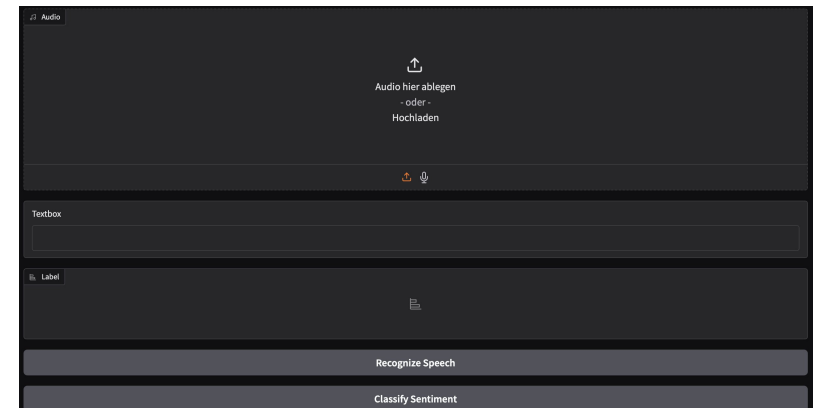
demo = gr.Blocks()
```

```
with demo:
    audio_file = gr.Audio(type="filepath")
    text = gr.Textbox()
    label = gr.Label()

    b1 = gr.Button("Recognize Speech")
    b2 = gr.Button("Classify Sentiment")

    b1.click(speech_to_text,
             inputs=audio_file, outputs=text)
    b2.click(text_to_sentiment,
             inputs=text, outputs=label)
```

[demo.launch\(\)](#)



# Blocks: Managing State

- Managing state in blocks, follow the same patterns as for Interfaces
- Blocks, in addition, add the possibility of using **browser state** (v > 5.17.1, still not packaged for conda) , persisting values for each user in the browser's **localStorage**, allowing data to **persist even after the page is refreshed or closed**.
- To use local state:
  1. Create a **gr.BrowserState** object. You can optionally provide an initial default value and a key to identify the data in the browser's localStorage
  2. Use it like a regular gr.State component in event listeners as inputs and outputs

# Dynamic Apps with the Render Decorator

- The components and event listeners you define in a Blocks so far have been fixed, the **@gr.render decorator** introduces the ability to dynamically change this
1. Create a function and attach the **@gr.render decorator** to it
  2. Add the input components to the inputs= argument of @gr.render, and create a corresponding argument in your function for each component
  3. Add all components inside the function that you want to render based on the inputs

```
import gradio as gr

with gr.Blocks() as demo:
    input_text = gr.Textbox(label="input")
    mode = gr.Radio(["textbox", "button"],
value="textbox")
    @gr.render(
        inputs=[input_text, mode],
        triggers=[input_text.submit])
    def show_split(text, mode):
        if len(text) == 0:
            gr.Markdown("## No Input Provided")
        else:
            for letter in text:
                if mode == "textbox":
                    gr.Textbox(letter)
                else:
                    gr.Button(letter)

demo.launch()
```

# OpenAI API

- OpenAI has the first easy-to-use API for an LLM
- OpenAI's REST API Specification allows one to interact with models like GPT through standard HTTP requests, but is seemingly also becoming a **de facto standard API implementation for interaction with other, open source models**, such as the KISSKI Chat AI
- Typical end points include:
  - POST /v1/chat/completions
    - Given a prompt, the model will return one or more predicted completions along with the probabilities of alternative tokens at each position
  - POST /embeddings
    - Get a vector representation of a given input that can be easily consumed by machine learning models and algorithms to measure the relatedness of text strings

# OpenAI API Class for our projects

```
from openai import OpenAI
```

```
class LLM:
```

```
    def __init__(self, api_key, base_url, model_name):
```

```
        self.api_key = api_key
```

```
        self.base_url = base_url
```

```
        self.model_name = model_name
```

```
        # Start OpenAI client
```

```
        self.client = OpenAI(
```

```
            api_key=self.api_key,
```

```
            base_url=self.base_url
```

```
        )
```

```
    def create_chat_completion(self, message):
```

```
        chat_completion = self.client.chat.completions.create(
```

```
            messages = [
```

```
                {"role": "system",
```

```
                  "content": "You are a helpful assistant"}, #set behavioral context of model
```

```
                {"role": "user", "content": message}
```

```
            ],
```

```
            model = self.model_name,
```

```
        )
```

```
        return chat_completion
```



# OpenAI API metadata

- The openAi requires 3 parameters, *base\_url*, *api\_key* and *model*
- To make the class configurable we will create a YAML file: ***settings.yml*** which provide the metadata for for the Client

models:

- meta-llama-3.1-8b-instruct
- codestral-22b

api\_key:

- '<api\_key>' # Replace with your API key

base\_url:

- <https://chat-ai.academiccloud.de/v1>

# OpenAI API class usage

```
def load_metadata():
    try:
        with open("settings.yml", "r") as file:
            meta_data = yaml.safe_load(file)
    except Exception as e:
        logger.info(f"Failed to read config file settings.yml \nException was:")
        logger.error(e)
        raise e
    return [meta_data["models"], meta_data['api_key'], meta_data['base_url']]

model, api_key, base_url = load_metadata()

llmConnection = LLM(api_key[0],base_url[0], model[0])
```

# A Simple Chatbot Demo

- We import our LLM class from a „functions“ package created in the project
- the yaml package to read our meta data and time to throttle our chatbot responses
- We create a block setup with components, and assign variables containing meta data and set up the openAI connection

```
import gradio as gr
from functions.LLMConnection import LLM
import yaml
import time
```

```
with gr.Blocks() as demo:
    chatbot = gr.Chatbot(type="messages")
    msg = gr.Textbox()
    clear = gr.Button("Clear")

    model, api_key, base_url = load_metadata()
    llmConnection = LLM(api_key[0], base_url[0], model[0])
```

# A Simple Chatbot Demo

- the first method `user()` updates the chatbot with the user message and clears the input field.
- The second method, `bot()`
  - updates the chatbot history with the **bot's response**
  - Gradio **automatically** turns any function with the `yield` keyword into a **streaming output interface**
  - when a user submits their message, we now **chain two events** with `.then()`: the return value of the `user()` event is fed to the `then` clause as argument for the `bot()` function
  - The chatbot's history is appended with `{"role": "user", "content": user_message}`.

```
def user(user_message, history: list):
    return "", history +
        [{"role": "user", "content": user_message}]

def bot(history: list):
    user_query = history[len(history)-1]['content']

    bot_message =
        llmConnection.create_chat_completion(user_query)
    history.append({"role": "assistant", "content":
        bot_message.choices[0].message.content})
    for character in
        bot_message.choices[0].message.content:
        history[-1]['content'] += character
        time.sleep(0.05)
        yield history

msg.submit(user, [msg, chatbot], [msg, chatbot], queue=False)
    .then(
        bot, chatbot, chatbot
    )

clear.click(lambda: None, None, chatbot, queue=False)
demo.launch()
```

# Questions? Let's Brainstorm!

