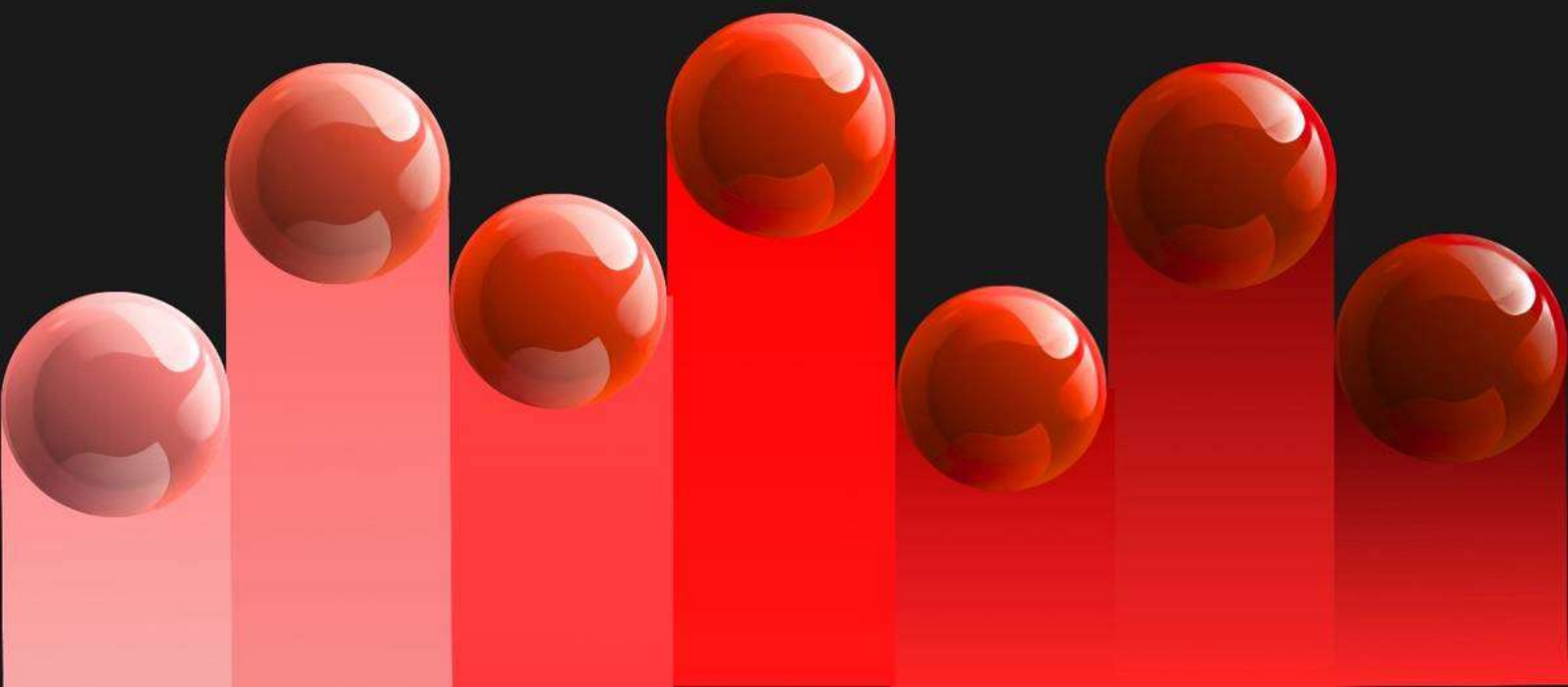
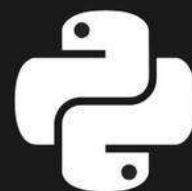




Building an LLM Agent in Python.


**A Step-by-Step Guide.
With Source Code**



Introduction to LLM Agent

This presentation will guide you through building a Language Learning Model (LLM) agent using Python. We will cover the setup of the development environment, loading a pre-trained model, preparing inputs, generating responses, creating an interactive web interface, handling errors, and deploying the application.

To begin, we need to ensure Python is installed and set up a virtual environment for our project. This isolates the dependencies and avoids conflicts with other projects. We will then install the necessary libraries.



```
# Install Python if not already installed
# Create and activate a virtual environment
python -m venv llm-agent-env
source llm-agent-env/bin/activate

# Install necessary packages
pip install transformers torch flask
```

follow for more

Installing the Hugging Face Transformers Library

The Transformers library by Hugging Face provides state-of-the-art pre-trained models for various NLP tasks. We will install this library to use one of their pre-trained models for our LLM



```
pip install transformers
```

Swipe next →

Loading a Pre-trained Model

We will load a pre-trained language model using the Transformers library. In this example, we use the GPT-2 model, which is known for its text generation capabilities. Loading the model and tokenizer will allow us to process and generate text.



```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load the tokenizer and model
tokenizer = AutoTokenizer.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_pretrained("gpt2")
```


Tokenizing Input

To prepare input text for the model, we need to tokenize it. Tokenization converts text into a format that the model can understand. This involves breaking the text into tokens and converting them to numerical IDs.



```
# Sample input text
input_text = "Hello, how are you?"

# Tokenize input
inputs = tokenizer(input_text, return_tensors="pt")
```

Generating Responses

Using the tokenized input, we can generate a response from the model. The model will output token IDs, which we then decode back into human-readable text. We can control the length and style of the generated text by adjusting parameters.



```
# Generate response
output = model.generate(inputs['input_ids'], max_length=50)
response = tokenizer.decode(output[0], skip_special_tokens=True)

print(response)
```

Creating a Simple Interface

To make our LLM agent accessible, we will create a web interface using Flask. Flask is a lightweight web framework for Python that allows us to build web applications quickly and easily.

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. The text 'pip install flask' is displayed in a white, monospaced font in the center of the window.

```
pip install flask
```

Building the Flask App

We will set up a basic Flask application with an endpoint to interact with the LLM agent. This endpoint will accept POST requests containing input text, process the input through the model, and return the generated response.

```
from flask import Flask, request, jsonify

app = Flask(__name__)


@app.route('/predict', methods=['POST'])
def predict():
    input_text = request.json['text']
    inputs = tokenizer(input_text, return_tensors="pt")
    output = model.generate(inputs['input_ids'], max_length=50)
    response = tokenizer.decode(output[0], skip_special_tokens=True)
    return jsonify(response=response)

if __name__ == '__main__':
    app.run(debug=True)
```


follow for more

Testing the Flask API

To ensure our Flask API is working correctly, we can test it using tools like curl or Postman. By sending a POST request with some input text, we should receive a generated response from the model.



```
# Example curl request
curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d '{"text": "Hello, LLM!"}'
```

Swipe next →

Handling Edge Cases

Robust applications handle errors gracefully. We will add error handling to our Flask endpoint to manage cases like missing input text or unexpected issues during processing. This ensures our application provides meaningful feedback to users.

```
● ● ●  
@app.route('/predict', methods=['POST'])  
def predict():  
    try:  
        input_text = request.json['text']  
        if not input_text:  
            return jsonify(error="No input text provided"), 400  
        inputs = tokenizer(input_text, return_tensors="pt")  
        output = model.generate(inputs['input_ids'], max_length=50)  
        response = tokenizer.decode(output[0], skip_special_tokens=True)  
        return jsonify(response=response)  
    except Exception as e:  
        return jsonify(error=str(e)), 500
```

follow for more

Deploying the Flask App

Finally, we will deploy our Flask application to a cloud platform such as Heroku or AWS. Deployment makes our application accessible to users worldwide. We will demonstrate the basic steps for deploying to Heroku.



```
# Example steps for Heroku deployment
heroku create llm-agent-app
git push heroku main
heroku open
```

Swipe next →

Linear Regression

save for later 

Linear regression is a statistical technique used to model the relationship between a dependent variable and one or more independent variables. It has applications in various fields, including finance, economics, and machine learning. In Python, we can perform linear regression using NumPy and scikit-learn.

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 5, 7, 9])

# Creating a linear regression model
model = LinearRegression()

# Fitting the model to the data
model.fit(X, y)

# Printing the coefficients
print("Slope (m):", model.coef_[0])
print("Intercept (c):", model.intercept_)
```

Results:

```
Slope (m): 1.7999999999999998
Intercept (c): 0.200000000000000018
```

Swipe next →

Principal Component Analysis (PCA)

follow for more

Principal Component Analysis (PCA) is a dimensionality reduction technique used to transform a high-dimensional dataset into a lower-dimensional subspace while preserving as much of the original variance as possible. It has applications in various fields, including data visualization, image processing, and machine learning. In Python, we can perform PCA using scikit-learn.

```
import numpy as np
from sklearn.decomposition import PCA

# Sample data
X = np.array([[2.5, 2.4], [0.5, 0.7],
              [2.2, 2.9], [1.9, 2.2], [3.1, 3.0],
              [2.3, 2.7], [2.0, 1.6], [1.0, 1.1],
              [1.5, 1.6], [1.1, 0.9]])

# Creating a PCA model
pca = PCA(n_components=2)

# Fitting the model to the data
X_transformed = pca.fit_transform(X)

print("Transformed data:")
print(X_transformed)
```

Swipe next →

Results:



Transformed data:

```
[[ 1.16480731  0.32385473]
 [-1.55292161 -0.70629919]
 [ 1.05410889  0.93713796]
 [ 0.10637062 -0.21928395]
 [ 1.82350474  0.28161197]
 [ 0.74940996  0.51286699]
 [-0.07819791 -0.78120193]
 [-1.35812771 -0.31992443]
 [-0.69413859  0.09744573]
 [-1.21481571 -0.12620789]]
```

Data scientist & ML Engineer



**Follow For More Data
Science Content**