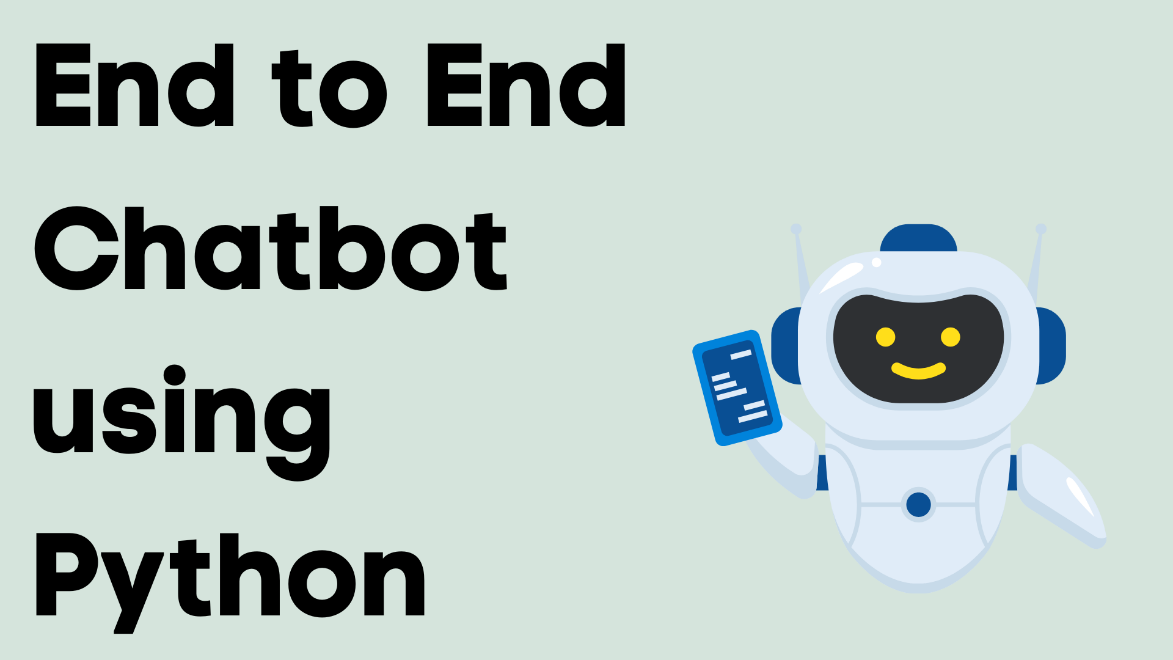
**CREATE A CHATBOT USING PYTHON**

**NAME:M.MOHAMED SHAHITH**

**REG.NO:952821106020**



**INTRODUCTION**

Chatbots are computer programs designed to simulate conversation with human users, and they have become increasingly popular in various applications, from customer support to personal assistants. Python is an excellent choice for building chatbots due to its simplicity, versatility, and the availability of powerful natural language processing (NLP) libraries and frameworks

.

**CONTENT FOR PROJECT PHASE 2:**

In this phase, consider exploring advanced techniques like using pre-trained language models (e.g., GPT-3) to enhance the quality of responses.

**ADVANCE TECHNIQUES:**

**1. List Comprehensions:**

List comprehensions provide a concise way to create lists. They are especially useful when you want to apply a function or expression to each element of an iterable.

**EXAMPLE**

squared\_numbers = [x\*\*2 for x in range(10)]

**2. Generators:**

Generators allow you to create iterable sequences without generating all the values at once, which can be memory-efficient. You can define generators using functions with the `yield` keyword.

**EXAMPLE**

def generate\_numbers(n):

for i in range(n):

yield i

**3. Decorators:**

Decorators are functions that can be used to modify the behavior of other functions or methods. They are commonly used for tasks like logging, authentication, and measuring execution time.

**EXAMPLE**

def my\_decorator(func):

def wrapper():

print("Something is happening before the function is called.")

func()

print("Something is happening after the function is called.")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

**4. Context Managers:**

Context managers, implemented using the `with` statement, are used for resource management. They are often used to handle files, database connections, and more.

**EXAMPLE**

with open('file.txt', 'r') as file:

data = file.read()

# File is automatically closed when exiting the block

**5. Concurrency and Multithreading:**

Python provides libraries like `threading` and `multiprocessing` for concurrent and parallel programming, which can help in optimizing CPU-bound and I/O-bound tasks.

**6. Asyncio:**

Python's `asyncio` library allows you to write asynchronous, non-blocking code, making it suitable for building scalable network applications and handling I/O-bound operations efficiently.

**7. Metaprogramming:**

Python's dynamic nature allows for advanced metaprogramming techniques, where you can modify or generate code at runtime. This includes using `eval`, `exec`, and manipulating classes and objects dynamically.

**8. Functional Programming:**

Python supports functional programming paradigms, including lambda functions, `map`, `filter`, and `reduce` functions from the `functools` module, as well as libraries like `itertools`.

**9. Duck Typing and Dynamic Typing:**

Python's dynamic typing and duck typing allow for flexible and generic programming, making it easier to write reusable and adaptable code**.**

**10. Type Annotations and Type Checking:**

With the introduction of type hints and tools like `mypy`, you can add static typing to Python code, which can improve code quality and catch errors early.

**11. Data Science and Machine Learning Libraries:**

Python has a rich ecosystem of libraries for data analysis and machine learning, including NumPy, pandas, scikit-learn, TensorFlow, and PyTorch.

**12. Web Frameworks:**

Python has popular web frameworks like Django and Flask for building web applications with ease.

**13. Cython and C Extensions:**

For performance-critical tasks, you can use Cython to write Python code that can be compiled to C. You can also create C extensions to integrate C code with Python.

**PROGRAMS**

**INPUT**

question =[]

answer = []

with open("../input/simple-dialogs-for-chatbot/dialogs.txt",'r') as f :

for line in f :

line = line.split('\t')

question.append(line[0])

answer.append(line[1])

print(len(question) == len(answer))

**OUTPUT**

question[:5]

answer[:5]

answer = [ i.replace("\n","") for i in answer]

**CODE**

def ask(sentence):

result, sentence = evaluate(sentence)

print('Question: %s' % (sentence))

print('Predicted answer: {}'.format(result))

ask(questions[100])

**output**

ask(questions[20])

print(answers[20])

ask(questions[10])

print(answers[10])

**CONCLUSION :**

In conclusion, building a chatbot using Python offers a powerful and flexible solution. Python’s rich ecosystem of libraries, natural language processing tools like NLTK and spaCy, and frameworks like Rasa or ChatterBot make it well-suited for chatbot development. With advanced techniques such as machine learning and deep learning, you can create intelligent and context-aware chatbots. Additionally, Python’s ease of integration with web frameworks and APIs allows you to deploy chatbots on various platforms, making it a popular choice for chatbot development.