CREATING A CHATBOT USING PYTHON

PHASE 5:
PROJECT DOCUMENTATION AND SUBMISSION

- **TOPIC: In** this part you will document your project and prepare it for submission.

INTRODUCTION:

- Chatbots can provide real-time customer support and are therefore a valuable asset in many industries. When you understand the basics of the ChatterBot library, you can build and train a self-learning chatbot with just a few lines of Python code. You'll get the basic chatbot up and running right away in step one, but the most interesting part is the learning phase, when you get to train your chatbot. The quality and preparation of your training data will make a big difference in your chatbot's performance. To simulate a real-world process that you might go through to create an industry-relevant Chatbot.

STEPS:

Step:1-Preparing the Dependencies:

The right dependencies need to be established before we can create a chatbot. Python and a ChatterBot library must be installed on our machine. With Pip, the Chatbot Python package manager, we can install ChatterBot.

Step:2- Creating and Training the Chatbot:

Once the dependence has been established, we can build and train our chatbot. We will import the ChatterBot module and start a new Chatbot Python instance. If so, we might incorporate the dataset into our chatbot's design or provide it with unique chat data.

Step:3-Communicating with the Python chatbot:

We can send a message and get a response once the chatbot Python has been trained. Creating a function that analyses user input and uses the chatbot's knowledge store to produce appropriate responses will be necessary.

Step:4-Complete Project Code

We will give you a full project code outlining every step and enabling you to start. This code can be modified to suit your unique requirements and used as the foundation for a chatbot.

APPROACH TO SOLVING THE PROBLEM FOR CREATE A CHATBOT IN PYTHON:

- 1: **Programming Language**:
- Python is the primary language for chatbot development.
- 2. **IDE (Integrated Development Environment)**:
- Choose an IDE for coding. Popular options include PyCharm, Visual Studio Code, or Jupyter Notebook for Python development
- .3. **Natural Language Processing (NLP) Libraries**:
- - **NLTK (Natural Language Toolkit)**: A powerful library for working with human language data.
- **SpaCy**: Another popular NLP library for natural language understanding and processing.
- **TextBlob**: Simplifies text processing for tasks like sentiment analysis and part-of-speech tagging.
- 4. **Machine Learning Libraries**:
- **scikit-learn**: Useful for machine learning tasks, such as text classification or clustering.
- - **TensorFlow or PyTorch**: For deep learning and building neural networks for chatbot intelligence
- .5. **Chatbot Frameworks**:
- **Rasa**: An open-source framework for building conversational AI chatbots.
- **ChatterBot**: A Python library for building chatbots with minimal coding.

APPROACH PROBLEM DEFINITION:

- Defining the problem is a crucial first step in creating a chatbot in Python. Here's how to approach problem definition:
- 1. ****Identify the Purpose**:**
- Determine the primary purpose of your chatbot. What specific problem or need will it address? For example, it could be for customer support, information retrieval, or process automation.
- 2. ****Target Audience**:**
- Understand who your target audience is. Who will be using the chatbot, and what are their characteristics, preferences, and pain points?
- 3. ****Use Cases**:**
- List the specific tasks or functions the chatbot will perform. Define the use cases that align with the bot's purpose.
- 4. ****Data Sources**:**
- Identify the data sources and databases required to support the chatbot's functions. Ensure you have access to the necessary data
- 5. ****User Flow**:**
- Create a flowchart or diagram illustrating how users will interact with the chatbot. Define the conversation paths and user interactions.

- 6. **Integration Points**:
- Determine if the chatbot needs to integrate with external systems or APIs. Identify these integration points and how data will be exchanged.
- 7. **Language Support**:
- Decide on the languages the chatbot will support, and consider multilingual capabilities if necessary.
- 8. **User Input Types**:
- Define the types of user inputs the chatbot will handle, such as text, voice, images, or a combination
- **9. **Response Output**:**
- Specify how the chatbot should respond to user queries, whether through text, voice, or other media.
- 10. **Security and Privacy**:
- Address security and privacy concerns. Define how user data will be handled and protected to ensure compliance with data protection regulations.

DESIGN THINKING:

- Design thinking is a valuable approach when creating a chatbot in Python to ensure that you develop a user-centered and effective solution. Here's how to apply design thinking principles in the process:
- 1. ****Empathize**:**
- Understand your users and their needs. Conduct user research, interviews, and surveys to gain insights into their preferences and challenges when interacting with a chatbot.
- 2. ****Define**:**
- Clearly define the problem or challenge that the chatbot is meant to solve. Create a problem statement that encapsulates the user's needs and goals.
- 3. ****Ideate**:**
- Brainstorm creative solutions for the chatbot's design and functionality. Encourage a diverse team to come up with innovative ideas.
- 4. ****Prototype**:**
- Create low-fidelity prototypes of the chatbot's user interface and conversation flow using Python libraries for UI design or prototyping tools. These could be sketches, wireframes, or interactive mockups.
- 5.****Test**:**
- Gather feedback on the prototypes by testing them with potential users. Learn from their interactions and adapt the design and features based on their input.

- .6. **Iterate**:
- Use the feedback to make improvements to the chatbot's design and functionality. Continue iterating and refining the chatbot's features and user experience.
- 7. **Develop and Build**:
- Begin the actual development of the chatbot using Python and relevant libraries for NLP and AI, like NLTK, SpaCy, scikit-learn, TensorFlow, or PyTorch.
- 8. **Test and Validate**:
- Perform rigorous testing of the chatbot's functionality, both in terms of NLP capabilities and user interactions. Validate that it effectively addresses the defined problem.
- 9. **Launch and Monitor**:
- Deploy the chatbot to your target audience and monitor its performance. Collect real-time data on user interactions and feedback.
- 10. **Feedback Loop**:
- Continue to gather user feedback post-launch and make necessary updates to enhance the chatbot's performance and user experience.

PROGRAM I :

```
•   import json
•   import re
•   import random_responses
•   # Load JSON data
•   def load_json(file):
•     with open(file) as bot_responses:
•       print(f"Loaded '{file}' successfully!")
•         return json.load(bot_responses)
•   # Store JSON data
•   Response_data = load_json("bot.json")
•   def get_response(input_string):
•           split_message = re.split(r'\s+|[,;?!.-]\s*', input_string.lower())
•     score_list = []
•     # Check all the responses    for response in response_data:
•      response_score = 0
•    required_score = 0
•        required_words = response["required_words"]
•        # Check if there are any required words
•            if required_words:
•                for word in split_message:
•                    if word in required_words:
•                        required_score += 1
•          # Amount of required words should match the required score
•         if required_score == len(required_words):
•            # print(required_score == len(required_words))
•           # Check each word the user has typed
•          for word in split_message:
•          # If the word is in the response, add to the score
•          if word in response["user_input"]:
•            response_score += 1
•        # Add score to list
•       score_list.append(response_score)
•         # Debugging: Find the best phrase
•          # print(response_score, response["user_input"])
•    # Find the best response and return it if they're not all 0
•      best_response = max(score_list)
•      response_index = score_list.index(best_response)
•       # Check if input is empty     if input_string == "":
•        return "Please type something so we can chat :("
•       # If there is no good response, return a random one..
•       if best_response != 0:
•          return
•     response_data[response_index]
•    ["bot_response"]
•    Return
•     random_responses.random_string()
•    while True:
•      user_input = input("You: ")
•     print("Bot:", get_response(user_input))
```

PROGRAM 2

- import random
- def random_string():
- random_list = [
-    "Please try writing something more descriptive.",
-       "Oh! It appears you wrote something I don't understand yet",
-   "Do you mind trying to rephrase that?",
-        "I'm terribly sorry, I didn't quite catch that.",
-        "I can't answer that yet, please try asking something else."
-    ]
-

- list_count = len(random_list)
- random_item = random.randrange(list_count)
- return random_list[random_item]

PROGRAM 3

- [
- {
- "response_type":"greeting",
- "user_input":["hello", "hi", "hey"],
- "bot_response":"Hey there!",
- "required_words":[]
- },
- {
- "response_type":"greeting",
- "user_input":["see you", "goodbye", "bye"],

```
    "bot_response": "See you later!",
   "required_words": []
  },
 {
   "response_type": "greeting",
    "user_input": ["nice", "to", "meet", "you"],
   "bot_response": "The pleasure is all mine!",
   "required_words": ["nice", "meet", "you"]
  },
   {
    "response_type": "question",
    "user_input": ["how", "to", "learn", "code", "coding", "apps"],
    "bot_response": "Start by typing:
 'How to learn coding' on Google.",
   "required_words": ["learn", "code"]
},
{
"response_type": "question",
 "user_input": ["refund", "how", "can", "I", "get"],
 "bot_response": "We don't offer refunds for free education.",
   "required_words": ["refund", "i"]
 },
{
"response_type": "question",
 "user_input": ["how", "are", "you"],
 "bot_response": "I'm great! Thanks for asking.",
  "required_words": ["how", "are", "you"]
}
]
```

IDEA:1

- <?xml version="1.0"
- encoding="UTF-8"?>
- <module type="PYTHON_MODULE"
- version="4">
- <component name="NewModuleRootManager">
-    <content url="file://$MODULE_DIR$">
-   <excludeFolder url="file://$MODULE_DIR$/venv" />
-    </content>
-   <orderEntry type="inheritedJdk" />
- <orderEntry type="sourceFolder" forTests="false" />
-   </component>
- </module>

IDEA:2

- <project version="4">
- <component
- name="ProjectRootManager"
- version="2"
- project-jdk-name="Python 3.10 (json_chatbot)"
- project-jdk-type="Python SDK"/>
- </project>

IDEA:3

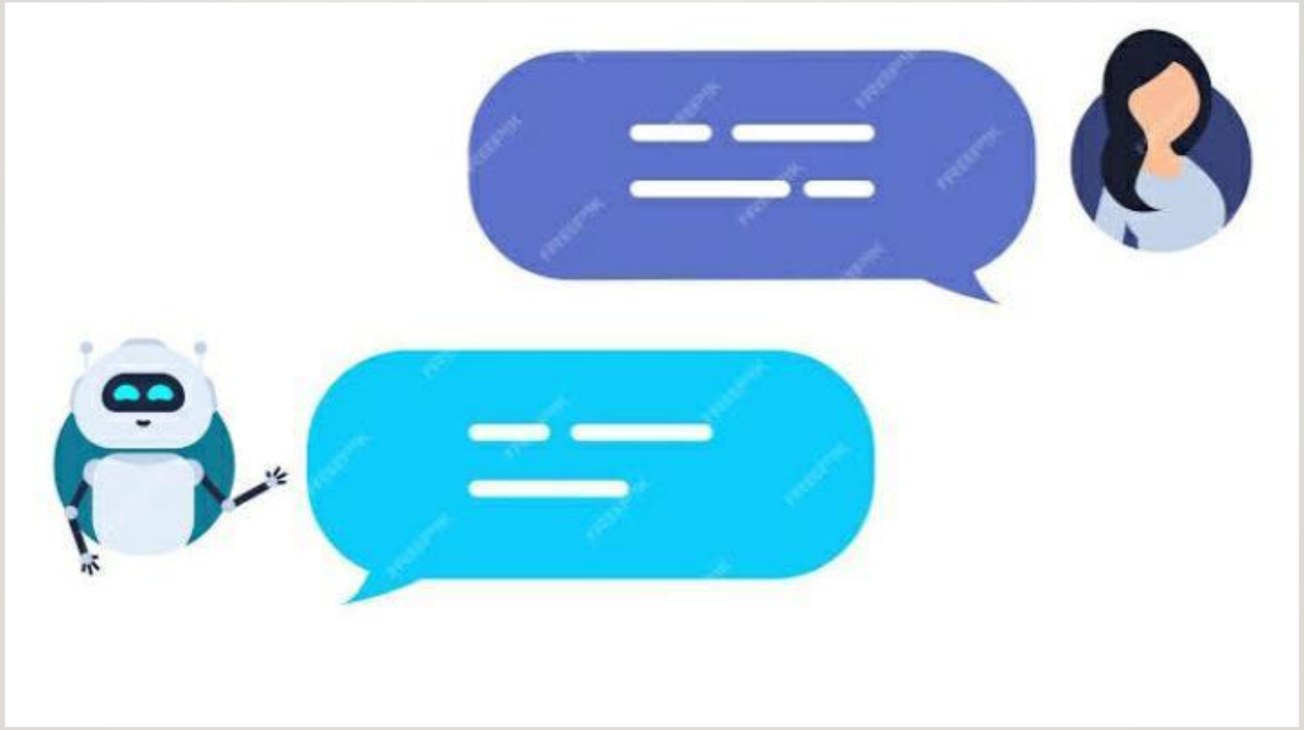
- <project version="4">
- <component name="ProjectModuleManager">
- <modules>
- <module fileurl="file://$PROJECT_DIR$/.idea/json_chatbot.iml" filepath="$PROJECT_DIR$/.idea/json_chatbot.iml"/>
- </modules>
- </component>
- </project>

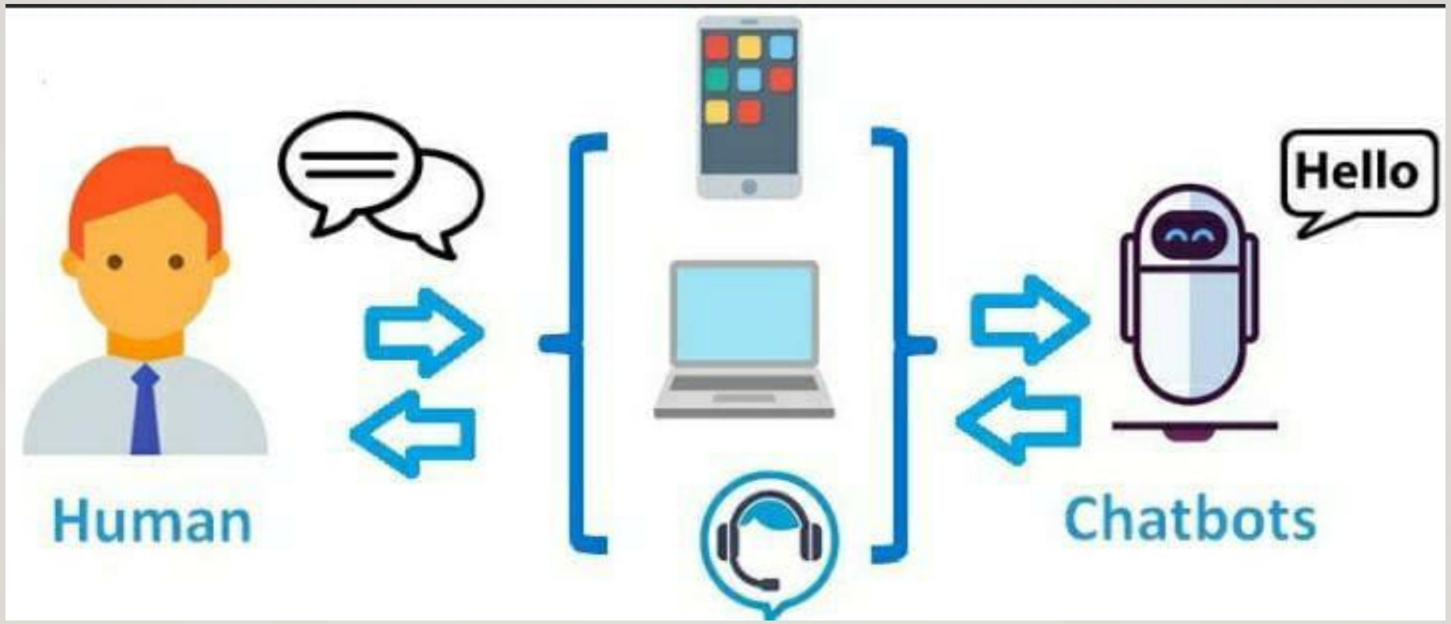
IDEA:4

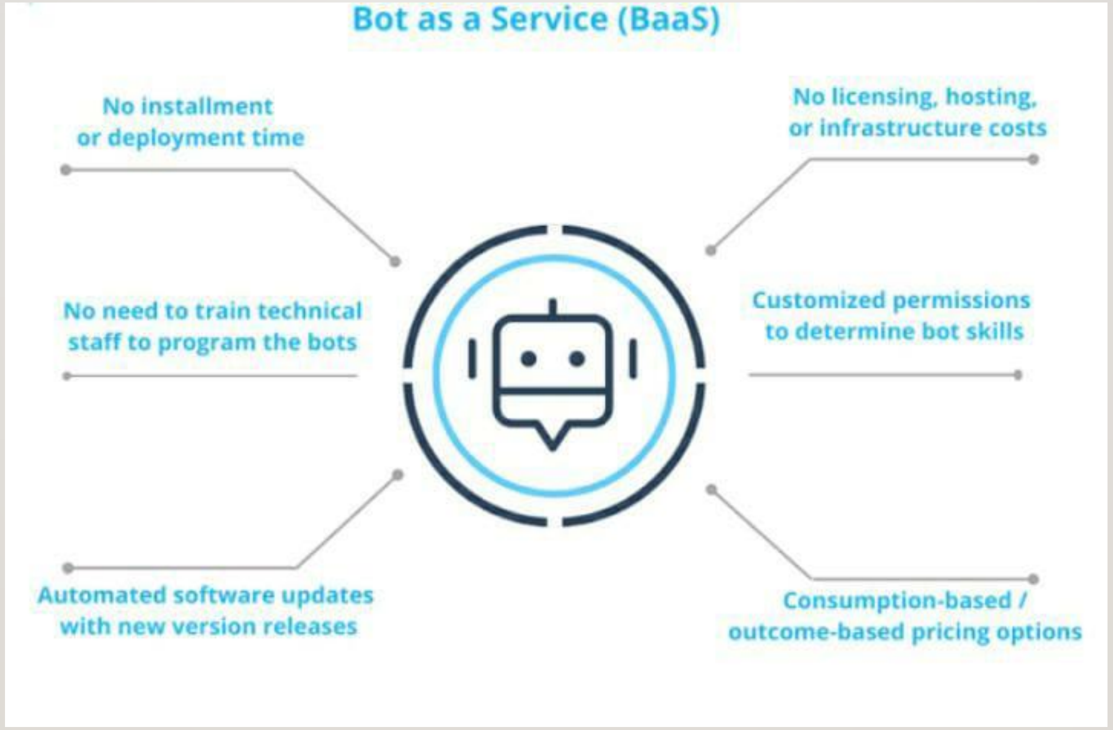
- <project version="4">
- <component name="VcsDirectoryMappings">
- <mapping directory="$PROJECT_DIR$" vcs="Git"/>
- </component>
- </project>

DATASET LINK:

- https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot




Human
Hello
Chatbots


Bot as a Service (BaaS)
No installment or deployment time
No licensing, hosting, or infrastructure costs
No need to train technical staff to program the bots
Customized permissions to determine bot skills
Automated software updates with new version releases
Consumption-based / outcome-based pricing options

CONCLUSION:

- Chatbot system is implemented to meet the requirements of the users. simulation / generating response from a chatbot is whenever the user context is matched. When a user begins asking queries in the chatbot GUI. the query is searched in the database. If the response is found in the database it is displayed to the user or else the system notifies the admin about the missing response in the database and gives a predefined response to the user. Chatbot system is implemented to meet the requirements of the users. simulation / generating response from a chatbot is whenever the user context is matched. When a user begins asking queries in the chatbot GUI. the query is searched in the database. If the response is found in the database it is displayed to the user or else the system notifies the admin about the missing response in the database and gives a predefined response to the user.