How to create a custom Memory class

Although there are a few predefined types of memory in LangChain, it is highly possible you will want to add your own type of memory that is optimal for your application. This notebook covers how to do that.

For this notebook, we will add a custom memory type to `ConversationChain`. In order to add a custom memory class, we need to import the base memory class and subclass it.

```python
from langchain import OpenAI, ConversationChain
from langchain.schema import BaseMemory
from pydantic import BaseModel
from typing import List, Dict, Any
```

In this example, we will write a custom memory class that uses spacy to extract entities and save information about them in a simple hash table. Then, during the conversation, we will look at the input text, extract any entities, and put any information about them into the context.

- Please note that this implementation is pretty simple and brittle and probably not useful in a production setting. Its purpose is to showcase that you can add custom memory implementations.

For this, we will need spacy.

```python
# !pip install spacy
# !python -m spacy download en_core_web_lg
```

```python
import spacy
nlp = spacy.load('en_core_web_lg')
```

```python
class SpacyEntityMemory(BaseMemory, BaseModel):
    """Memory class for storing information about entities."""
```

Skip to main content

```python
    entities: dict = {}
    # Define key to pass information about entities into prompt.
    memory_key: str = "entities"

    def clear(self):
        self.entities = {}

    @property
    def memory_variables(self) -> List[str]:
        """Define the variables we are providing to the prompt."""
        return [self.memory_key]

    def load_memory_variables(self, inputs: Dict[str, Any]) -> Dict[str, str]:
        """Load the memory variables, in this case the entity key."""
        # Get the input text and run through spacy
        doc = nlp(inputs[list(inputs.keys())[0]])
        # Extract known information about entities, if they exist.
        entities = [self.entities[str(ent)] for ent in doc.ents if str(ent) in
self.entities]
        # Return combined information about entities to put into context.
        return {self.memory_key: "\n".join(entities)}

    def save_context(self, inputs: Dict[str, Any], outputs: Dict[str, str]) ->
None:
        """Save context from this conversation to buffer."""
        # Get the input text and run through spacy
        text = inputs[list(inputs.keys())[0]]
        doc = nlp(text)
        # For each entity that was mentioned, save this information to the
dictionary.
        for ent in doc.ents:
            ent_str = str(ent)
            if ent_str in self.entities:
                self.entities[ent_str] += f"\n{text}"
            else:
                self.entities[ent_str] = text
```

We now define a prompt that takes in information about entities as well as user input

```python
from langchain.prompts.prompt import PromptTemplate

template = """The following is a friendly conversation between a human and an AI.
The AI is talkative and provides lots of specific details from its context. If the
AI does not know the answer to a question, it truthfully says it does not know.
You are provided with information about entities the Human mentions, if relevant.

Relevant entity information:
{entities}

Conversation:
Human: {input}
```

Skip to main content

```
        input_variables=["entities", "input"], template=template
    )
```

And now we put it all together!

```
llm = OpenAI(temperature=0)
conversation = ConversationChain(llm=llm, prompt=prompt, verbose=True,
memory=SpacyEntityMemory())
```

In the first example, with no prior knowledge about Harrison, the "Relevant entity information" section is empty.

```
conversation.predict(input="Harrison likes machine learning")
```

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is
talkative and provides lots of specific details from its context. If the AI does
not know the answer to a question, it truthfully says it does not know. You are
provided with information about entities the Human mentions, if relevant.

Relevant entity information:


Conversation:
Human: Harrison likes machine learning
AI:

> Finished ConversationChain chain.
```

```
" That's great to hear! Machine learning is a fascinating field of study. It
involves using algorithms to analyze data and make predictions. Have you ever
studied machine learning, Harrison?"
```

Now in the second example, we can see that it pulls in information about Harrison.

```
conversation.predict(input="What do you think Harrison's favorite subject in
college was?")
```

```
> Entering new ConversationChain chain...
```

Skip to main content

```
talkative and provides lots of specific details from its context. If the AI does
not know the answer to a question, it truthfully says it does not know. You are
provided with information about entities the Human mentions, if relevant.

Relevant entity information:
Harrison likes machine learning

Conversation:
Human: What do you think Harrison's favorite subject in college was?
AI:

> Finished ConversationChain chain.
```

```
' From what I know about Harrison, I believe his favorite subject in college was
machine learning. He has expressed a strong interest in the subject and has
mentioned it often.'
```

Again, please note that this implementation is pretty simple and brittle and probably not useful in a production setting. Its purpose is to showcase that you can add custom memory implementations.