Memory

🚧 *Docs under construction* 🚧

By default, Chains and Agents are stateless, meaning that they treat each incoming query independently (like the underlying LLMs and chat models themselves). In some applications, like chatbots, it is essential to remember previous interactions, both in the short and long-term. The **Memory** class does exactly that.

LangChain provides memory components in two forms. First, LangChain provides helper utilities for managing and manipulating previous chat messages. These are designed to be modular and useful regardless of how they are used. Secondly, LangChain provides easy ways to incorporate these utilities into chains.

Get started

Memory involves keeping a concept of state around throughout a user's interactions with an language model. A user's interactions with a language model are captured in the concept of ChatMessages, so this boils down to ingesting, capturing, transforming and extracting knowledge from a sequence of chat messages. There are many different ways to do this, each of which exists as its own memory type.

In general, for each type of memory there are two ways to understanding using memory. These are the standalone functions which extract information from a sequence of messages, and then there is the way you can use this type of memory in a chain.

Memory can return multiple pieces of information (for example, the most recent N messages and a summary of all previous messages). The returned information can either be a string or a list of messages.

We will walk through the simplest form of memory: "buffer" memory, which just involves keeping a buffer of all prior messages. We will show how to use the modular utility functions here, then show how it can be used in a chain (both returning a string as well as a list of messages).

ChatMessageHistory

One of the core utility classes underpinning most (if not all) memory modules is the `ChatMessageHistory` class. This is a super lightweight wrapper which exposes convenience methods for saving Human messages, AI messages, and then fetching them all.

You may want to use this class directly if you are managing memory outside of a chain.

```
from langchain.memory import ChatMessageHistory

history = ChatMessageHistory()

history.add_user_message("hi!")

history.add_ai_message("whats up?")
```

```
history.messages
```

```
    [HumanMessage(content='hi!', additional_kwargs={}),
     AIMessage(content='whats up?', additional_kwargs={})]
```

ConversationBufferMemory

We now show how to use this simple concept in a chain. We first showcase `ConversationBufferMemory` which is just a wrapper around ChatMessageHistory that extracts the messages in a variable.

We can first extract it as a string.

```
from langchain.memory import ConversationBufferMemory
```

```
memory = ConversationBufferMemory()
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

```
memory.load_memory_variables({})
```

```
    {'history': 'Human: hi!\nAI: whats up?'}
```

We can also get the history as a list of messages

```
memory = ConversationBufferMemory(return_messages=True)
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

```
memory.load_memory_variables({})
```

```
{'history': [HumanMessage(content='hi!', additional_kwargs={}),
  AIMessage(content='whats up?', additional_kwargs={})]}
```

Using in a chain

Finally, let's take a look at using this in a chain (setting `verbose=True` so we can see the prompt).

```
from langchain.llms import OpenAI
from langchain.chains import ConversationChain


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

```
conversation.predict(input="Hi there!")
```

```


> 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.

    Current conversation:

    Human: Hi there!
    AI:

    > Finished chain.
```

```
    " Hi there! It's nice to meet you. How can I help you today?"
```

```
conversation.predict(input="I'm doing well! Just having a conversation with an AI.")
```

```
    > 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.

    Current conversation:
    Human: Hi there!
    AI:  Hi there! It's nice to meet you. How can I help you today?
```

```
    Human: I'm doing well! Just having a conversation with an AI.
    AI:

    > Finished chain.
```

```
    " That's great! It's always nice to have a conversation with someone new. What would you like to talk about?"
```

```
conversation.predict(input="Tell me about yourself.")
```

```
    > 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.

    Current conversation:
    Human: Hi there!
    AI:  Hi there! It's nice to meet you. How can I help you today?
    Human: I'm doing well! Just having a conversation with an AI.
    AI:  That's great! It's always nice to have a conversation with someone new. What would you like to talk about?
    Human: Tell me about yourself.
    AI:
```

```
    > Finished chain.
```

```
    " Sure! I'm an AI created to help people with their everyday tasks. I'm programmed to understand natural language and provide helpful information. I'm also constantly learning and updating my knowledge base so I can provide more accurate and helpful answers."
```

Saving Message History

You may often have to save messages, and then load them to use again. This can be done easily by first converting the messages to normal python dictionaries, saving those (as json or something) and then loading those. Here is an example of doing that.

```
import json

from langchain.memory import ChatMessageHistory
from langchain.schema import messages_from_dict, messages_to_dict

history = ChatMessageHistory()

history.add_user_message("hi!")

history.add_ai_message("whats up?")
```

```
dicts = messages_to_dict(history.messages)
```

```
dicts
```

```
    [{'type': 'human', 'data': {'content': 'hi!', 'additional_kwargs': {}}},
     {'type': 'ai', 'data': {'content': 'whats up?', 'additional_kwargs': {}}}]
```

```
new_messages = messages_from_dict(dicts)
```

```
new_messages
```

```
    [HumanMessage(content='hi!', additional_kwargs={}),
     AIMessage(content='whats up?', additional_kwargs={})]
```

And that's it for the getting started! There are plenty of different types of memory, check out our examples to see them all