Custom agent

This notebook goes through how to create your own custom agent.

In this example, we will use OpenAI Tool Calling to create this agent. **This is generally the most reliable way to create agents.**

We will first create it WITHOUT memory, but we will then show how to add memory in. Memory is needed to enable conversation.

Load the LLM

First, let's load the language model we're going to use to control the agent.

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-3.5-turbo",
temperature=0)
```

Define Tools

Next, let's define some tools to use. Let's write a really simple Python function to calculate the length of a word that is passed in.

Note that here the function docstring that we use is pretty important. Read more about why this is the case here

```
from langchain.agents import tool


@tool
def get_word_length(word: str) -> int:
    """Returns the length of a word."""
    return len(word)


get_word_length.invoke("abc")
```

```
3
```

```
tools = [get_word_length]
```

Create Prompt

Now let us create the prompt. Because OpenAI Function Calling is finetuned for tool usage, we hardly need any instructions on how to reason, or how to output format. We will just have two input variables: `input` and `agent_scratchpad`. `input` should be a string containing the user objective. `agent_scratchpad` should be a sequence of messages that contains the previous agent tool invocations and the corresponding tool outputs.

```
from langchain_core.prompts import ChatPromptTe
MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are very powerful assistant, b
don't know current events",
        ),
        ("user", "{input}"),

MessagesPlaceholder(variable_name="agent_scratc
    ]
)
```

Bind tools to LLM

How does the agent know what tools it can use?

In this case we're relying on OpenAI tool calling LLMs, which take tools as a separate argument and have been specifically trained to know when to invoke those tools.

To pass in our tools to the agent, we just need to format them to the OpenAI tool format and pass them to our model. (By `bind`-ing the functions, we're making sure that they're passed in each time the model is invoked.)

```
llm_with_tools = llm.bind_tools(tools)
```

Create the Agent

Putting those pieces together, we can now create the agent. We will import two last utility functions: a component for formatting intermediate steps (agent action, tool output pairs) to input messages that can be sent to the model, and a component for converting the output message into an agent action/agent finish.

```
from
langchain.agents.format_scratchpad.openai_tools
import (
    format_to_openai_tool_messages,
)
from
langchain.agents.output_parsers.openai_tools
import OpenAIToolsAgentOutputParser

agent = (
    {
        "input": lambda x: x["input"],
        "agent_scratchpad": lambda x:
format_to_openai_tool_messages(
            x["intermediate_steps"]
        ),
    }
    | prompt
    | llm_with_tools
    | OpenAIToolsAgentOutputParser()
)
```

```
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent=agent,
tools=tools, verbose=True)
```

```
list(agent_executor.stream({"input": "How
many letters in the word eudca"}))
```

```
> Entering new AgentExecutor chain...

Invoking: `get_word_length` with `{'word':
'eudca'}`


5There are 5 letters in the word "eudca".

> Finished chain.
```

```
[{'actions': [OpenAIToolAgentAction(tool='get_w
tool_input={'word': 'eudca'}, log="\nInvoking:
`get_word_length` with `{'word': 'eudca'}`\n\n\
[AIMessageChunk(content='', additional_kwargs={
[{'index': 0, 'id': 'call_A07D5TuyqcNIL0DIEVRPp
'function': {'arguments': '{\n  "word": "eudca"
'get_word_length'}, 'type': 'function'}]})],
tool_call_id='call_A07D5TuyqcNIL0DIEVRPpZkg')],
  'messages': [AIMessageChunk(content='', addit
{'tool_calls': [{'index': 0, 'id':
'call_A07D5TuyqcNIL0DIEVRPpZkg', 'function': {'
"word": "eudca"\n}', 'name': 'get_word_length'}
'function'}]})]},
```

```
 {'steps':
[AgentStep(action=OpenAIToolAgentAction(tool='g
tool_input={'word': 'eudca'}, log="\nInvoking:
`get_word_length` with `{'word': 'eudca'}`\n\n\
[AIMessageChunk(content='', additional_kwargs={
[{'index': 0, 'id': 'call_A07D5TuyqcNIL0DIEVRPp
'function': {'arguments': '{\n  "word": "eudca"
'get_word_length'}, 'type': 'function'}]})],
tool_call_id='call_A07D5TuyqcNIL0DIEVRPpZkg'),
  'messages': [FunctionMessage(content='5',
name='get_word_length')]},
 {'output': 'There are 5 letters in the word "e
  'messages': [AIMessage(content='There are 5 l
word "eudca".')]}]
```

If we compare this to the base LLM, we can see that the LLM alone struggles

```
llm.invoke("How many letters in the word
educa")
```

```
AIMessage(content='There are 6 letters in the
word "educa".')
```

Adding memory

This is great - we have an agent! However, this agent is stateless - it doesn't remember anything about previous interactions. This means you can't ask follow up questions easily. Let's fix that by adding in memory.

In order to do this, we need to do two things:

1. Add a place for memory variables to go in the prompt
2. Keep track of the chat history

First, let's add a place for memory in the prompt. We do this by adding a placeholder for messages with the key `"chat_history"`. Notice that we put this ABOVE the new user input (to follow the conversation flow).

```
from langchain.prompts import MessagesPlacehold

MEMORY_KEY = "chat_history"
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are very powerful assistant, b
at calculating lengths of words.",
        ),
        MessagesPlaceholder(variable_name=MEMOR
        ("user", "{input}"),

MessagesPlaceholder(variable_name="agent_scratc
```

```
    ]
)
```

We can then set up a list to track the chat history

```
from langchain_core.messages import
AIMessage, HumanMessage

chat_history = []
```

We can then put it all together!

```
agent = (
    {
        "input": lambda x: x["input"],
        "agent_scratchpad": lambda x:
format_to_openai_tool_messages(
            x["intermediate_steps"]
        ),
        "chat_history": lambda x:
x["chat_history"],
    }
    | prompt
    | llm_with_tools
    | OpenAIToolsAgentOutputParser()
)
```

```
agent_executor = AgentExecutor(agent=agent,
tools=tools, verbose=True)
```

When running, we now need to track the inputs and outputs as chat history

```
input1 = "how many letters in the word
educa?"
result = agent_executor.invoke({"input":
input1, "chat_history": chat_history})
chat_history.extend(
    [
        HumanMessage(content=input1),
        AIMessage(content=result["output"]),
    ]
)
agent_executor.invoke({"input": "is that a
real word?", "chat_history": chat_history})
```

```
> Entering new AgentExecutor chain...

Invoking: `get_word_length` with `{'word':
'educa'}`


5There are 5 letters in the word "educa".
```

```
> Finished chain.


> Entering new AgentExecutor chain...
No, "educa" is not a real word in English.

> Finished chain.
```

```
{'input': 'is that a real word?',
 'chat_history': [HumanMessage(content='how
many letters in the word educa?'),
  AIMessage(content='There are 5 letters in
the word "educa".')],
 'output': 'No, "educa" is not a real word in
English.'}
```