Agents

The core idea of agents is to use an LLM to choose a sequence of actions to take. In chains, a sequence of actions is hardcoded (in code). In agents, a language model is used as a reasoning engine to determine which actions to take and in which order.

There are several key components here:

Agent

This is the class responsible for deciding what step to take next. This is powered by a language model and a prompt. This prompt can include things like:

1. The personality of the agent (useful for having it respond in a certain way)
2. Background context for the agent (useful for giving it more context on the types of tasks it's being asked to do)
3. Prompting strategies to invoke better reasoning (the most famous/widely used being ReAct)

LangChain provides a few different types of agents to get started. Even then, you will likely want to customize those agents with parts (1) and (2). For a full list of agent types see agent types

Tools

Tools are functions that an agent calls. There are two important considerations here:

1. Giving the agent access to the right tools
2. Describing the tools in a way that is most helpful to the agent

Without both, the agent you are trying to build will not work. If you don't give the agent access to a correct set of tools, it will never be able to accomplish the objective. If you don't describe the tools properly, the agent won't know how to properly use them.

LangChain provides a wide set of tools to get started, but also makes it easy to define your own (including custom descriptions). For a full list of tools, see here

Toolkits

Often the set of tools an agent has access to is more important than a single tool. For this LangChain provides the concept of toolkits - groups of tools needed to accomplish specific objectives. There are generally around 3-5 tools in a toolkit.

LangChain provides a wide set of toolkits to get started. For a full list of toolkits, see here

AgentExecutor

The agent executor is the runtime for an agent. This is what actually calls the agent and executes the actions it chooses. Pseudocode for this runtime is below:

```
next_action = agent.get_action(...)
while next_action != AgentFinish:
    observation = run(next_action)
    next_action = agent.get_action(..., next_action, observation)
return next_action
```

While this may seem simple, there are several complexities this runtime handles for you, including:

1. Handling cases where the agent selects a non-existent tool
2. Handling cases where the tool errors
3. Handling cases where the agent produces output that cannot be parsed into a tool invocation
4. Logging and observability at all levels (agent decisions, tool calls) either to stdout or LangSmith.

Other types of agent runtimes

The `AgentExecutor` class is the main agent runtime supported by LangChain. However, there are other, more experimental runtimes we also support. These include:

- Plan-and-execute Agent
- Baby AGI
- Auto GPT

Get started

This will go over how to get started building an agent. We will use a LangChain agent class, but show how to customize it to give it specific context. We will then define custom tools, and then run it all in the standard LangChain AgentExecutor.

Set up the agent

We will use the OpenAIFunctionsAgent. This is easiest and best agent to get started with. It does however require usage of ChatOpenAI models. If you want to use a different language model, we would recommend using the ReAct agent.

For this guide, we will construct a custom agent that has access to a custom tool. We are choosing this example because we think for most use cases you will NEED to customize either the agent or the tools. The tool we will give the agent is a tool to calculate the length of a word. This is useful because this is actually something LLMs can mess up due to tokenization. We will first create it WITHOUT memory, but we will then show how to add memory in. Memory is needed to enable conversation.

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

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(temperature=0)
```

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.

```
from langchain.agents import tool

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

tools = [get_word_length]
```

Now let us create the prompt. We can use the `OpenAIFunctionsAgent.create_prompt` helper function to create a prompt automatically. This allows for a few different ways to customize, including passing in a custom SystemMessage, which we will do.

```
from langchain.schema import SystemMessage
from langchain.agents import OpenAIFunctionsAgent
system_message = SystemMessage(content="You are very powerful
assistant, but bad at calculating lengths of words.")
prompt =
OpenAIFunctionsAgent.create_prompt(system_message=system_message)
```

Putting those pieces together, we can now create the agent.

```
agent = OpenAIFunctionsAgent(llm=llm, tools=tools, prompt=prompt)
```

Finally, we create the AgentExecutor - the runtime for our agent.

```
from langchain.agents import AgentExecutor
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

Now let's test it out!

```
agent_executor.run("how many letters in the word educa?")
```

```
> Entering new AgentExecutor chain...

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

5

There are 5 letters in the word "educa".

> Finished chain.

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

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. Add memory to the AgentExecutor (note that we add it here, and NOT to the agent, as this is the outermost chain)

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"`.

```
from langchain.prompts import MessagesPlaceholder

MEMORY_KEY = "chat_history"
prompt = OpenAIFunctionsAgent.create_prompt(
    system_message=system_message,
    extra_prompt_messages=
[MessagesPlaceholder(variable_name=MEMORY_KEY)]
)
```

Next, let's create a memory object. We will do this by using `ConversationBufferMemory`. Importantly, we set `memory_key` also equal to `"chat_history"` (to align it with the prompt) and set `return_messages` (to make it return messages rather than a string).

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(memory_key=MEMORY_KEY,
return_messages=True)
```

We can then put it all together!

```
agent = OpenAIFunctionsAgent(llm=llm, tools=tools, prompt=prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, memory=memory,
verbose=True)
agent_executor.run("how many letters in the word educa?")
agent_executor.run("is that a real word?")
```