Concepts

The core idea of agents is to use a language model 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:

Schema

LangChain has several abstractions to make working with agents easy.

AgentAction

This is a dataclass that represents the action an agent should take. It has a `tool` property (which is the name of the tool that should be invoked) and a `tool_input` property (the input to that tool)

AgentFinish

This represents the final result from an agent, when it is ready to return to the user. It contains a `return_values` key-value mapping, which contains the final agent output. Usually, this contains an `output` key containing a string that is the agent's response.

Intermediate Steps

These represent previous agent actions and corresponding outputs from this CURRENT agent run. These are important to pass to future iteration so the agent knows what work it has already done. This is typed as a `List[Tuple[AgentAction, Any]]`. Note that observation is currently left as type `Any` to be maximally flexible. In practice, this is often a string.

Agent

This is the chain responsible for deciding what step to take next. This is usually powered by a language model, a prompt, and an output parser.

Different agents have different prompting styles for reasoning, different ways of encoding inputs, and different ways of parsing the output. For a full list of built-in agents see agent types. You can also **easily build custom agents**, should you need further control.

Agent Inputs

The inputs to an agent are a key-value mapping. There is only one required key: `intermediate_steps`, which corresponds to `Intermediate Steps` as described above.

Generally, the PromptTemplate takes care of transforming these pairs into a format that can best be passed into the LLM.

Agent Outputs

The output is the next action(s) to take or the final response to send to the user (`AgentAction`s or `AgentFinish`). Concretely, this can be typed as `Union[AgentAction, List[AgentAction], AgentFinish]`.

The output parser is responsible for taking the raw LLM output and transforming it into one of these three types.

AgentExecutor

The agent executor is the runtime for an agent. This is what actually calls the agent, executes the actions it chooses, passes the action outputs back to the agent, and repeats. In pseudocode, this looks roughly like:

```
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) to stdout and/or to LangSmith.

Tools

Tools are functions that an agent can invoke. The `Tool` abstraction consists of two components:

1. The input schema for the tool. This tells the LLM what parameters are needed to call the tool. Without this, it will not know what the correct inputs are. These parameters should be sensibly named and described.
2. The function to run. This is generally just a Python function that is invoked.

Considerations

There are two important design considerations around tools:

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

Without thinking through both, you won't be able to build a working agent. If you don't give the agent access to a correct set of tools, it will never be able to accomplish the objectives you give it. If you don't describe the tools well, the agent won't know how to use them properly.

LangChain provides a wide set of built-in tools, but also makes it easy to define your own (including custom descriptions). For a full list of built-in tools, see the tools integrations section

Toolkits

For many common tasks, an agent will need a set of related tools. For this LangChain provides the concept of toolkits - groups of around 3-5 tools needed to accomplish specific objectives. For example, the GitHub toolkit has a tool for searching through GitHub issues, a tool for reading a file, a tool for commenting, etc.

LangChain provides a wide set of toolkits to get started. For a full list of built-in toolkits, see the toolkits integrations section

Help us out by providing feedback on this documentation page: