**Understanding LangChain Tools and Agents: A Guide to Building Smart AI Applications**

<https://colab.research.google.com/drive/1GHHGsDFB5266Cc0xDsZ6OWzkB5GGSxFW?usp=sharing>

<https://colab.research.google.com/drive/1-xMYU9ExZqoySEX-XHAvEaE17PCWvc9H?usp=sharing>

**What are Agents**

The LangChain documentation actually has a pretty [good page](https://python.langchain.com/docs/modules/agents/) on the high level concepts around its agents. It’s a short easy read, and definitely worth skimming through before getting started.

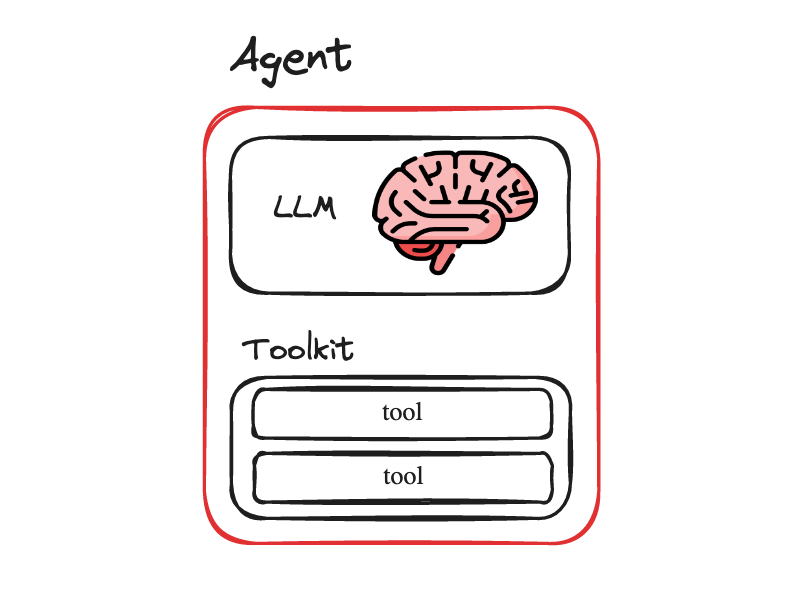
If you lookup the definition of AI Agents, you get something along the lines of "An entity that is able to perceive its environment, act on its environment, and make intelligent decisions about how to reach a goal it has been given, as well as the ability to learn as it goes"

That fits the definition of LangChain agents pretty well I would say. What makes all this possible in software is the reasoning abilities of Large Language Model’s (LLM’s). The brains of a LangChain agent are an LLM. It is the LLM that is used to reason about the best way to carry out the ask requested by a user.

In order to carry out its task, and operate on things and retrieve information, the agent has what are called Tool’s in LangChain, at its disposal. It is through these tools that it is able to interact with its environment.

The tools are basically just methods/classes the agent has access to that can do things like interact with a Stock Market index over an API, update a Google Calendar event, or run a query against a database. We can build out tools as needed, depending on the nature of tasks we are trying to carry out with the agent to fulfil.

A collection of Tools in LangChain are called a Toolkit. Implementation wise, this is literally just an array of the Tools that are available for the agent. As such, the high level overview of an agent in LangChain looks something like this

Image by author

So, at a basic level, an agent needs

* an LLM to act as its brain, and to give it its reasoning abilities
* tools so that it can interact with the environment around it and achieve its goals

**Building the Agent**

To make some of these concepts more concrete, let’s build a simple agent.

We will create a Mathematics Agent that can perform a few simple mathematical operations.

**Environment setup**

First lets setup our environment and script

mkdir simple-math-agent &amp;&amp; cd simple-math-agent

touch math-agent.py

python3 -m venv .venv

. .venv/bin/activate

pip install langchain langchain\_openai

Alternatively, you can also clone the code used here [from GitHub](https://github.com/smaameri/simple-math-agent)

git clone git@github.com:smaameri/simple-math-agent.git

or check out the code inside a [Google Colab](https://colab.research.google.com/drive/1dHG7d4Iq0yuDEOWK1UaMZ7ADeTo0uAFT?usp=sharing) also.

**The Tools**

The simplest place to start will be to fist define the tools for our Maths agent.

Let’s give it "add", "multiply" and "square" tools, so that it can perform those operations on questions we pass to it. By keeping our tools simple we can focus on the core concepts, and build the tools ourselves, instead of relying on an existing and more complex tools like the WikipediaTool, that acts as a wrapper around the Wikipedia API, and requires us to import it from the LangChain library.

Again, we are not trying to do anything fancy here, just keeping it simple and putting the main building blocks of an agent together so we can understand how they work, and get our first agent up and running.

Let’s start with the "add" tool. The bottom up way to create a Tool in LangChain would be to extend the **BaseTool** class, set the **name** and **description** fields on the class, and implement the **\_run** method. That would look like this

from langchain\_core.tools import BaseTool

class AddTool(BaseTool):

name = "add"

description = "Adds two numbers together"

args\_schema: Type[BaseModel] = AddInput

return\_direct: bool = True

def \_run(

self, a: int, b: int, run\_manager: Optional[CallbackManagerForToolRun] = None

) -> str:

return a + b

Notice that we need to implement the **\_run** method to show what our tool does with the parameters that are passed to it.

Notice also how it requires a pydantic model for the **args\_schema**. We will define that here

AddInput

a: int = Field(description="first number")

b: int = Field(description="second number")

Now, LangChain does give us an easier way to define tools, then by needing to extend the **BaseTool** class each time. We can do this with the help of the **@too**l decorator. Defining the "add" tool in LangChain using the @tool decorator will look like this

from langchain.tools import tool

@tool

def add(a: int, b: int) -> int:

"""Adds two numbers together""" # this docstring gets used as the description

return a + b # the actions our tool performs

Much simpler right. Behind the scenes, the decorator magically uses the method provided to extend the BaseTool class, just as we did earlier. Some thing to note:

* the method name also becomes the tool name
* the method params define the input parameters for the tool
* the docstring gets converted into the tools description

You can access these properties on the tool also

print(add.name) # add

print(add.description) # Adds two numbers together.

print(add.args) # {'a': {'title': 'A', 'type': 'integer'}, 'b': {'title': 'B', 'type': 'integer'}}

Note that the description of a tool is very important as this is what the LLM uses to decide whether or not it is the right tool for the job. A bad description may lead to the not tool getting used when it should be, or getting used at the wrong times.

With the **add** tool done, let’s move on to the definitions for our **multiply** and **square** tools.

@tool

def multiply(a: int, b: int) -> int:

"""Multiply two numbers."""

return a \* b

@tool

def square(a) -> int:

"""Calculates the square of a number."""

a = int(a)

return a \* a

And that is it, simple as that.

So we have defined our own three [custom tools](https://python.langchain.com/docs/modules/tools/custom_tools/). A more common use case might be to use some of the already provided and existing tools in LangChain, which you can see [here](https://python.langchain.com/docs/integrations/tools/). However, at the source code level, they would all be built and defined using a similar methods as described above.

And that is it as far as our Tools our concerned. Now time to combine our tools into a Toolkit.

**The Toolkit**

Toolkits sound fancy, but they are actually very simple. They are **literally** just a a list of tools. We can define our toolkit as an array of tools like so

toolkit = [add, multiply, square]

And that’s it. Really straightforward, and nothing to get confused over.

Usually Toolkits are groups of tools that are useful together, and would be helpful for agents trying to carry out certain kinds of tasks. For example an SQLToolkit might contain a tool for generating an SQL query, validating an SQL query, and executing an SQL query.

The [Integrations Toolkit](https://python.langchain.com/docs/integrations/toolkits/) page on the LangChain docs has a large list of toolkits developed by the community that might be useful for you.

**The LLM**

As mentioned above, an LLM is the brains of an agent. It decides which tools to call based on the question passed to it, what are the best next steps to take based on a tools description. It also decides when it has reached its final answer, and is ready to return that to the user.

Let’s setup the LLM here

from langchain\_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-3.5-turbo-1106", temperature=0)

**The Prompt**

Lastly we need a prompt to pass into our agent, so it has a general idea bout what kind of agent it is, and what sorts of tasks it should solve.

Our agent requires a ChatPromptTemplate to work (more on that later). This is what a barebones ChatPromptTemplate looks like. The main part we care about is the system prompt, and the rest are just the default settings we are required to pass in.

In our prompt we have included a sample answer, showing the agent how we want it to return the answer only, and not any descriptive text along with the answer

prompt = ChatPromptTemplate.from\_messages(

[

("system", """

You are a mathematical assistant. Use your tools to answer questions.

If you do not have a tool to answer the question, say so.

Return only the answers. e.g

Human: What is 1 + 1?

AI: 2

"""),

MessagesPlaceholder("chat\_history", optional=True),

("human", "{input}"),

MessagesPlaceholder("agent\_scratchpad"),

]

)

That is it. We have setup our Tools and Toolkit, which our agent will need as part of its setup, so its knows what are the types of actions and capabilities it has at its disposal. And we have also setup the LLM and system prompt.

Now for the fun part. Setting up our Agent!

**The Agent**

LangChain has a [number of different agents types](https://python.langchain.com/docs/modules/agents/agent_types/) that can be created, with different reasoning powers and abilities. We will be using the most capable and powerful agent currently available, the [OpenAI Tools](https://python.langchain.com/docs/modules/agents/agent_types/openai_tools/) agent. As per the docs on the the OpenAI Tools agent, which uses newer OpenAI models also,

Newer OpenAI models have been fine-tuned to detect when one or more function(s) should be called and respond with the inputs that should be passed to the function(s). In an API call, you can describe functions and have the model intelligently choose to output a JSON object containing arguments to call these functions. The goal of the OpenAI tools APIs is to more reliably return valid and useful function calls than what can be done using a generic text completion or chat API.

In other words this agents is good at generating the correct structure for calling functions, and is able to understand if more than one function (tool) might be needed for our task also. This agent also has the ability to call functions (tools) with multiple input parameters, just like ours do. Some agents can only work with functions that have a single input parameter.

If you are familiar with [OpenAI’s Function](https://platform.openai.com/docs/guides/function-calling) calling feature, where we can use the OpenAI LLM to generate the correct parameters to call a function with, the OpenAI Tools agent we are using here is leveraging some of that power in order to be able to call the correct tool, with the correct parameters.

In order to setup an agent in LangChain, we need to use one of the factory methods provided for creating the agent of our choice.

The factory method for creating an OpenAI tools agent is **create\_openai\_tools\_agent()**. And it requires passing in the llm, tools and prompt we setup above. So let’s initialise our agent.

agent = create\_openai\_tools\_agent(llm, toolkit, prompt)

Finally, in order to run agents in LangChain, we cannot just call a "*run*" type method on them directly. They need to be run via an AgentExecutor.

Am bringing up the Agent Executor only here at the end as I don’t think it’s a critical concept for understanding how the agents work, and bring it up at the start with everything else would just the whole thing seem more complicated than it needs to be, as well as distract from understanding some of the other more fundamental concepts.

So, now that we are introducing it, an AgentExecutor acts as the runtime for agents in LangChain, and allow an agent to keep running until it is ready to return its final response to the user. In pseudo-code, the AgentExecutor’s are doing something along the lines of (pulled directly from [the LangChain docs](https://python.langchain.com/docs/modules/agents/concepts/#agentexecutor))

next\_action = agent.get\_action(...)

while next\_action != AgentFinish:

observation = run(next\_action)

next\_action = agent.get\_action(..., next\_action, observation)

return next\_action

So they are basically a while loop that keep’s calling the next action methods on the agent, until the agent has returned its final response.

So, let us setup our agent inside the agent executor. We pass it the agent, and must also pass it the toolkit. And we are setting verbose to True so we can get an idea of what the agent is doing as it is processing our request

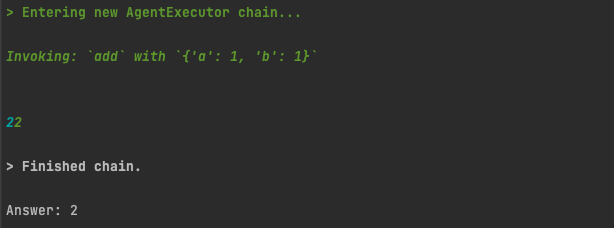
agent\_executor = AgentExecutor(agent=agent, tools=toolkit, verbose=True)

And that is it. We are now ready to pass commands to our agent

result = agent\_executor.invoke({"input": "what is 1 + 1"})

Let run our script, and see the agent’s output

python3 math-agent.py

Image by author

Since we have set **verbose=True** on the AgentExecutor, we can see the lines of Action our agent has taken. It has identified we should call the "**add**" tool, called the "**add**" tool with the required parameters, and returned us our result.

This is what the full source code looks like

import os

from langchain.agents import AgentExecutor, create\_openai\_tools\_agent

from langchain\_openai import ChatOpenAI

from langchain.tools import BaseTool, StructuredTool, tool

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder

os.environ["OPENAI\_API\_KEY"] = "sk-"

# setup the tools

@tool

def add(a: int, b: int) -> int:

"""Add two numbers."""

return a + b

@tool

def multiply(a: int, b: int) -> int:

"""Multiply two numbers."""

return a \* b

@tool

def square(a) -> int:

"""Calculates the square of a number."""

a = int(a)

return a \* a

prompt = ChatPromptTemplate.from\_messages(

[

("system", """You are a mathematical assistant.

Use your tools to answer questions. If you do not have a tool to

answer the question, say so.

Return only the answers. e.g

Human: What is 1 + 1?

AI: 2

"""),

MessagesPlaceholder("chat\_history", optional=True),

("human", "{input}"),

MessagesPlaceholder("agent\_scratchpad"),

]

)

# Choose the LLM that will drive the agent

llm = ChatOpenAI(model="gpt-3.5-turbo-1106", temperature=0)

# setup the toolkit

toolkit = [add, multiply, square]

# Construct the OpenAI Tools agent

agent = create\_openai\_tools\_agent(llm, toolkit, prompt)

# Create an agent executor by passing in the agent and tools

agent\_executor = AgentExecutor(agent=agent, tools=toolkit, verbose=True)

result = agent\_executor.invoke({"input": "what is 1 + 1?"})

print(result['output'])

**Testing our agent**

Let’s shoot a few questions at our agent to see how it performs.

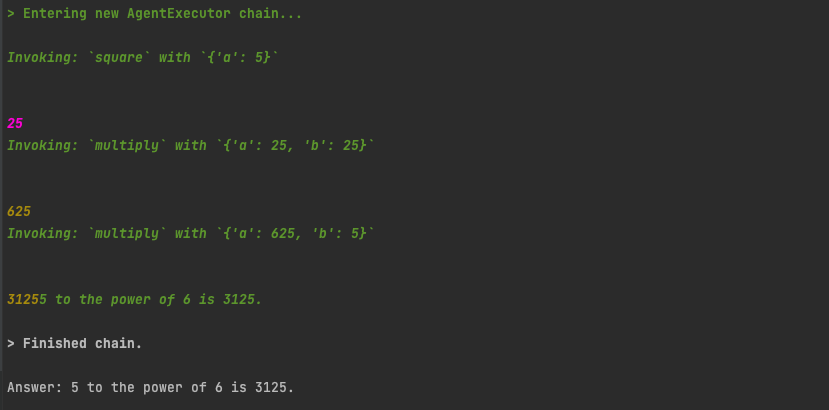
**what is 5 squared?**

Again we get the correct result, and see that it does use our **square** tool

Image by author

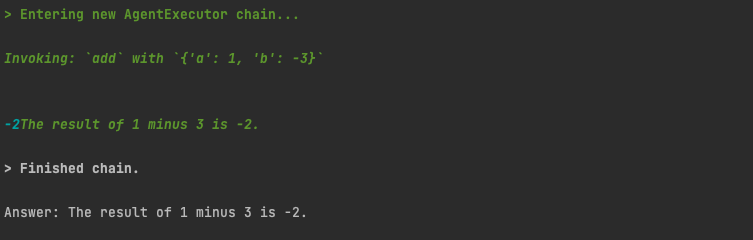
**what is 5 to the power of 6?**

It takes an interesting course of action. It first uses the **square** tool. And then, using the result of that, tries to use the **multiply** tool a few times to get the final answer. Admittedly, the final answer, 3125, is wrong, and needs to be multiplied by 5 one more time to get the correct answer. But it is interesting to see how the agent tried to use different tools, and multiple steps to try and get to the final answer.

Image by author

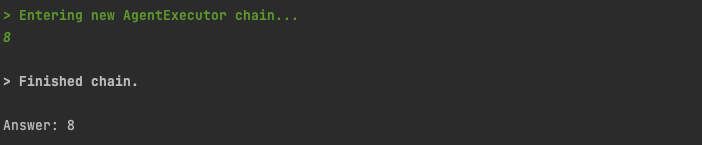
**what is 1 minus 3?**

We don’t have a minus tool. But it is smart enough to use our add tool, but set the second value to -3. Its funny and somewhat amazing sometimes how they are smart and creative like that.

Image by author

**what is the square root of 64**

As a final test, what if we ask it to carry out a mathematical operation that is not part of our tool set? Since we have no tools for square rooting, it does not attempt to call a tool, and instead calculates the value directly using just the LLM.

Image by author

Our system prompt did tell it to answer that it "does not know" if it does not have the correct tool for the job, and it did do that sometimes during testing. An improved initial system prompt could probably help resolve that, at least to some extent

**Observations**

Based on using the agent a bit, I noticed the following

* when asking it direct questions which it had the tools to answer with, it was pretty consistent at using the correct tools for the job, and returning the correct answer. So, pretty reliable in that sense.
* if the question is a little complicated, for example our "5 to the power of 6" question, it does not always return the correct results.
* it can sometimes use just the pure power of the LLM to answer our question, without invoking our tools.

**The Future**

Agents, and programs that can reason from themselves, are a new paradigm in programming, and I think they are going to become a much more mainstream part of how lots of things are built. Obviously the non-deterministic (i.e not wholly predictable) nature of LLM’s means that agents results will also suffer from this, questioning how much we can rely on them for tasks where we need to be sure of the answers we have.

Perhaps as the technology matures, their results can be more and more predictable, and we may develop some work arounds for this.

I can also see agent type libraries and packages starting to become a thing. Similar to how we install third party libraries and packages into software, for example via the pip package manager for python, or Docker Hub for docker images, I wonder if we may start to see a library and package manager of agents start being developed, with agents developed that become very good at their specific tasks, which we can then also install as packages into out application.

Indeed LangChain’s [library of Toolkits](https://python.langchain.com/docs/integrations/toolkits/) for agents to use, listed on their Integrations page, are sets of Tools built by the community for people to use, which could be an early example of agent type libraries built by the community.

**Types of Tools**

* We have build in tools and custom tools in langchain.

**Built-in Tools**

* Ready-to-use for common tasks

Examples:

* PythonREPLTool → run Python code
* RequestsGetTool → make API calls
* WikipediaAPIWrapper → fetch Wikipedia content
* For [Build-In Tools](https://python.langchain.com/docs/integrations/tools/)

# Usage of Build-in tool  
from langchain\_community.tools import DuckDuckGoSearchRun  
  
search = DuckDuckGoSearchRun()  
  
search.invoke("Obama's first name?")

**🛠️ Custom Tools**

Wrap your own Python functions Use @tool decorator or Tool.from\_function()

Examples:

* get\_employee\_info(name) → fetch from your database
* summarize\_document(path) → summarize your PDFs
* 🔧 Tailored for your business logic

from langchain.agents import Tool  
from langchain.agents import initialize\_agent  
from langchain.agents.agent\_types import AgentType  
from langchain.chat\_models import ChatOpenAI  
  
# Define your custom function  
def count\_words(text: str) -> str:  
 num\_words = len(text.strip().split())  
 return f"The text contains {num\_words} words."  
  
# Wrap it as a Tool  
word\_count\_tool = Tool(  
 name="WordCounter",  
 func=count\_words,  
 description="Counts the number of words in a given text. Use this when you want to know how many words a user has written."  
)  
  
# Create a language model  
llm = ChatOpenAI(temperature=0)  
  
# Initialize the agent with the tool  
agent = initialize\_agent(  
 tools=[word\_count\_tool],  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True,  
)  
  
# Try running the agent with a prompt  
response = agent.run("How many words are in: LangChain makes building LLM apps easy?")  
print(response)

**What is Tool Execution :-**

* This is when the tool’s func() is actually **executed** with the given input (e.g., "3 5"), and the result is returned to the agent.

**Example :-**

from langchain.agents import Tool, initialize\_agent, AgentType  
from langchain.chat\_models import ChatOpenAI  
  
# 1. Tool Binding  
def multiply(text: str) -> str:  
 a, b = map(int, text.split())  
 return str(a \* b)  
  
multiply\_tool = Tool(  
 name="Multiply",  
 func=multiply,  
 description="Multiplies two numbers provided as a space-separated string."  
)  
  
# 2. Initialize agent with tool  
llm = ChatOpenAI(temperature=0)  
agent = initialize\_agent(  
 tools=[multiply\_tool],  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True  
)  
  
# 3. Tool Calling + Execution  
agent.run("What is 7 times 6?")

**Example 2:- Currency convertor using tools**

* Step -1 : - Define the Currency Converter Tool

# Define the Currency Converter Tool  
import requests  
from langchain.agents import Tool  
from langchain.chat\_models import ChatOpenAI  
from langchain.agents import initialize\_agent, AgentType  
  
# Currency converter function  
def convert\_currency(input\_text: str) -> str:  
 """  
 Input format: '100 USD to INR'  
 """  
 try:  
 parts = input\_text.strip().upper().split()  
 amount = float(parts[0])  
 from\_currency = parts[1]  
 to\_currency = parts[3]  
  
 # Call exchange rate API  
 url = f"https://api.exchangerate-api.com/v4/latest/{from\_currency}"  
 response = requests.get(url)  
 data = response.json()  
  
 rate = data["rates"].get(to\_currency)  
 if rate is None:  
 return f"Conversion rate not found for {to\_currency}."  
   
 converted = amount \* rate  
 return f"{amount} {from\_currency} = {converted:.2f} {to\_currency}"  
   
 except Exception as e:  
 return f"Error: {str(e)}. Format should be like '100 USD to INR'"

* Step-2 Bind It as a Tool

currency\_tool = Tool(  
 name="CurrencyConverter",  
 func=convert\_currency,  
 description="Use this tool to convert currency like '100 USD to INR'."  
)

* Step-3 Initialize LLM Agent with the Tool

llm = ChatOpenAI(temperature=0)  
  
agent = initialize\_agent(  
 tools=[currency\_tool],  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True  
)  
  
response = agent.run("Convert 250 EUR to JPY")  
  
print(response)  
  
-------------------------------------------------------------------------------  
> Entering new AgentExecutor chain...  
I should use the CurrencyConverter tool to convert EUR to JPY.  
Action: CurrencyConverter  
Action Input: 250 EUR to JPY  
Observation: 250.0 EUR = 42347.50 JPY  
Thought:I now know the final answer  
Final Answer: 250 EUR is equal to 42347.50 JPY.  
  
> Finished chain.  
250 EUR is equal to 42347.50 JPY.

**Finally**

* **Tools = extra powers for the LLM.**
* They let your AI **interact with APIs, files, databases**, etc.
* Useful when you want the agent to go **beyond just generating text** and **do real tasks**.

Example 2

Here is the complete code:

from dotenv import load\_dotenv  
from langchain import hub  
from langchain.agents import (  
 AgentExecutor,  
 create\_react\_agent,  
)  
from langchain\_core.tools import Tool  
from langchain\_openai import ChatOpenAI  
  
# Load environment variables from .env file  
load\_dotenv()  
  
  
# Define a very simple tool function that returns the current time  
def get\_current\_time(\*args, \*\*kwargs):  
 """Returns the current time in H:MM AM/PM format."""  
 import datetime # Import datetime module to get current time  
  
 now = datetime.datetime.now() # Get current time  
 return now.strftime("%I:%M %p") # Format time in H:MM AM/PM format  
  
  
# List of tools available to the agent  
tools = [  
 Tool(  
 name="Time", # Name of the tool  
 func=get\_current\_time, # Function that the tool will execute  
 # Description of the tool  
 description="Useful for when you need to know the current time",  
 ),  
]  
  
# Pull the prompt template from the hub  
# ReAct = Reason and Action  
# https://smith.langchain.com/hub/hwchase17/react  
prompt = hub.pull("hwchase17/react")  
  
# Initialize a ChatOpenAI model  
llm = ChatOpenAI(  
 model="gpt-4o-mini", temperature=0  
)  
  
# Create the ReAct agent using the create\_react\_agent function  
agent = create\_react\_agent(  
 llm=llm,  
 tools=tools,  
 prompt=prompt,  
 stop\_sequence=True,  
)  
  
# Create an agent executor from the agent and tools  
agent\_executor = AgentExecutor.from\_agent\_and\_tools(  
 agent=agent,  
 tools=tools,  
 verbose=True,  
)  
  
# Run the agent with a test query  
response = agent\_executor.invoke({"input": "What time is it?"})  
  
# Print the response from the agent  
print("response:", response)

The rise of AI-powered applications has brought significant advancements in natural language processing (NLP) and automation. LangChain, an open-source framework, has emerged as a powerful tool for developing applications that integrate language models with external tools, knowledge bases, and APIs. At the core of LangChain’s functionality are **Tools** and **Agents**, which enable AI models to perform actions dynamically.

This article explores LangChain’s **Tools** and **Agents**, how they work, and how you can leverage them to build intelligent AI-powered applications.

**What Are LangChain Tools?**

**Tools** in LangChain are interfaces that allow an AI model (such as GPT-4) to interact with external systems, retrieve data, or perform actions beyond simple text generation. These tools act as APIs or function calls that the AI can invoke when needed.

**Types of LangChain Tools**

1. **Search Tools** — Allow models to fetch real-time information from sources like Google Search or Bing.
2. **Database Query Tools** — Help AI interact with structured data sources like SQL databases or vector stores.
3. **API Call Tools** — Enable interaction with external APIs (e.g., weather data, stock prices, etc.).
4. **Mathematical Tools** — Allow AI to perform complex calculations and data processing.
5. **File Processing Tools** — Help analyze documents, PDFs, or CSV files.
6. **Code Execution Tools** — Provide AI the capability to write, execute, and debug code in Python or other languages.

**Example: Using Tools in LangChain**

Here’s a simple example of defining a LangChain tool that performs a mathematical calculation:

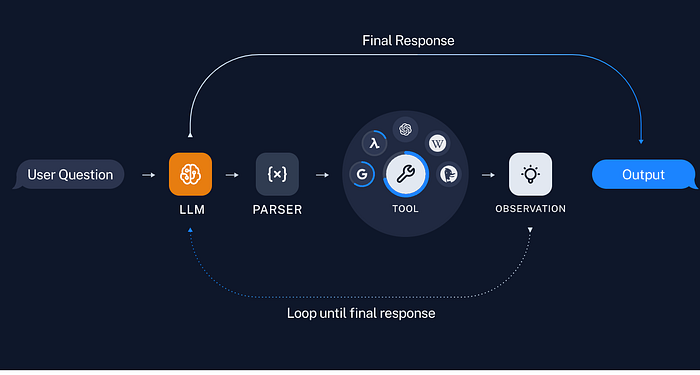
from langchain.tools import Tool  
  
def multiply\_numbers(x, y):  
 return x \* y  
math\_tool = Tool(  
 name="MultiplyNumbers",  
 func=lambda inputs: multiply\_numbers(\*inputs),  
 description="Multiplies two numbers and returns the result."  
)

This tool can now be used by an agent or another LangChain component to perform calculations dynamically.

**What Are LangChain Agents?**

**Agents** in LangChain are advanced components that enable AI models to decide when and how to use tools dynamically. Instead of relying on predefined scripts, agents analyze user queries and choose the best tools to achieve a goal.

Press enter or click to view image in full size



**Types of LangChain Agents**

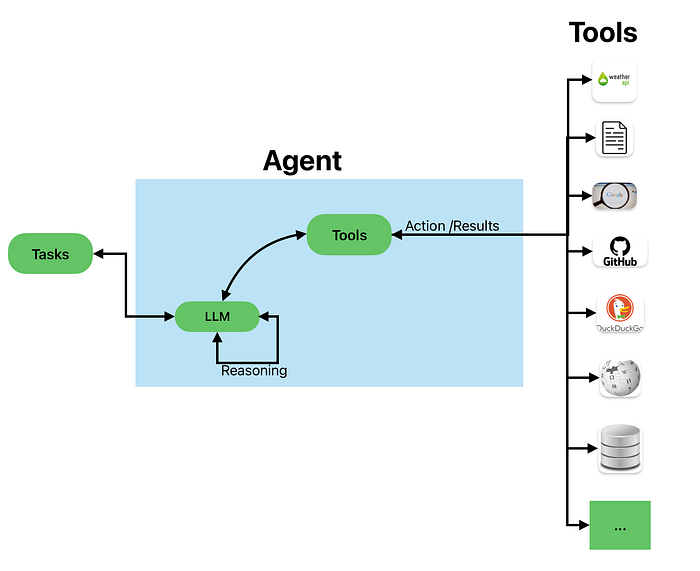
1. **Reactive Agents** — Select and execute tools based on user input without long-term memory.
2. **Conversational Agents** — Maintain memory of past interactions, improving responses over time.
3. **Planning Agents** — Break down complex tasks into smaller steps and execute them sequentially.
4. **Custom Agents** — Designed for specific use cases, such as automating workflows or handling customer support.

**How Agents Work**

Agents typically operate in a loop:

1. Receive user input.
2. Decide which tool(s) to use.
3. Execute the tool and retrieve results.
4. Generate a response based on the output.

Press enter or click to view image in full size



**Example: Creating an Agent in LangChain**

from langchain.agents import initialize\_agent  
from langchain.chat\_models import ChatOpenAI  
from langchain.agents import AgentType  
  
llm = ChatOpenAI(temperature=0, model\_name="gpt-4")  
agent = initialize\_agent(  
 tools=[math\_tool], # Adding our previously created tool  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True  
)  
response = agent.run("What is 7 multiplied by 6?")  
print(response)

Here, the agent automatically identifies that it needs to use the MultiplyNumbers tool and executes the operation.

**Use Cases for LangChain Tools and Agents**

**1. Automated Research Assistants**

* Using web search tools to fetch the latest articles and summarize information.

**2. Financial and Data Analysis**

* Analyzing stock trends using financial APIs and presenting insights.

**3. Customer Support Automation**

* Handling FAQs and troubleshooting via interactive agents.

**4. Coding Assistants**

* Writing and debugging code snippets dynamically using Python execution tools.

**5. Document Processing and Retrieval**

* Searching through legal documents, contracts, or research papers using vector databases.

**Here are some examples using Langchain (Tools and Agents):**

**Google Search :**

import os  
from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain.chat\_models import ChatOllama  
from langchain\_community.utilities import GoogleSerperAPIWrapper  
  
MODEL = "llama3.2"  
os.environ["SERPER\_API\_KEY"] = "\*\*\*"  
# Initialize the Google Search Tool  
search = GoogleSerperAPIWrapper()  
  
# Initialize a language model (LLM)  
llm = ChatOllama(model=MODEL)  
  
# Define the tool for search  
search\_tool = Tool(  
 name="Google Search",  
 func=search.run,  
 description="Use this tool when you need to search for real-time information from Google."  
)  
  
# List of tools the agent can use  
tools = [search\_tool]  
  
agent = initialize\_agent(  
 tools=tools,  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True,  
 handle\_parsing\_errors=True # Allows the agent to retry when parsing fails  
)  
  
# Example query  
response = agent.run("Tell me about Langchain.")  
print(response)

Sample Output:

Thought:Final Answer: LangChain is an open source framework for building applications based on large language models (LLMs), providing a standard interface for chains, integrations with other tools, and end-to-end chains for common applications.

**Wikipedia:**

import os  
from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain.chat\_models import ChatOllama  
from langchain.tools import WikipediaQueryRun  
from langchain.utilities import WikipediaAPIWrapper  
  
  
MODEL = "llama3.2"  
# Initialize a language model (LLM)  
llm = ChatOllama(model=MODEL)  
  
# Define the tool for search  
wiki = WikipediaQueryRun(api\_wrapper=WikipediaAPIWrapper())  
  
wiki\_tool = Tool(  
 name="Wikipedia Search",  
 func=wiki.run,  
 description="Useful for searching Wikipedia articles."  
)  
  
  
# List of tools the agent can use  
tools = [wiki\_tool]  
  
agent = initialize\_agent(  
 tools=tools,  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True,  
 handle\_parsing\_errors=True # Allows the agent to retry when parsing fails  
)  
  
# Example query  
response = agent.run("in 2 lines tell me about Italy")  
print(response)

**Weather:**

import os  
from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain.chat\_models import ChatOllama  
from langchain.tools import WikipediaQueryRun  
from langchain.utilities import WikipediaAPIWrapper  
import requests  
  
MODEL = "llama3.2"  
API\_KEY="\*\*\*"  
# Initialize a language model (LLM)  
llm = ChatOllama(model=MODEL)  
  
def get\_weather(city):  
 api\_key = API\_KEY  
 url = f"http://api.weatherapi.com/v1/current.json?key={api\_key}&q={city}"  
 response = requests.get(url)  
 return response.json()  
  
weather\_tool = Tool(  
 name="Weather Lookup",  
 func=lambda city: get\_weather(city),  
 description="Provides real-time weather updates for a given city."  
)  
  
# Example usage  
result = weather\_tool.run("New York")  
print(result)

**Connect to MySQL:**

from langchain.tools import tool  
from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
import os  
from langchain.chat\_models import ChatOllama  
from langchain.sql\_database import SQLDatabase  
from sqlalchemy import create\_engine  
  
# MySQL Connection Details  
MYSQL\_USERNAME = "root"  
MYSQL\_PASSWORD = "pwd"  
MYSQL\_HOST = "localhost" # e.g., "localhost" or IP  
MYSQL\_PORT = "3306" # Default MySQL port  
MYSQL\_DATABASE = "db1"  
  
# Create MySQL connection URI  
mysql\_uri = f"mysql+mysqlconnector://{MYSQL\_USERNAME}:{MYSQL\_PASSWORD}@{MYSQL\_HOST}:{MYSQL\_PORT}/{MYSQL\_DATABASE}"  
  
# Initialize MySQL Database Connection  
db = SQLDatabase.from\_uri(mysql\_uri)  
  
  
  
# Initialize LLM Model (Using Ollama)  
MODEL = "llama3.2"  
llm = ChatOllama(model=MODEL, streaming=True)  
  
# SQL Query Executor Tool  
sql\_tool = Tool(  
 name="SQL Query Executor",  
 func=db.run,  
 description="Executes SQL queries and retrieves results from MySQL."  
)  
  
# Example Query Execution  
query = "SELECT COUNT(\*) FROM orders ;"  
result = sql\_tool.run(query)  
print("Total Count"+ result)

**Call DateTime Functions:**

import os  
from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain.chat\_models import ChatOllama  
import datetime # Import datetime module to get current time  
  
MODEL = "llama3.2"  
  
# Initialize a language model (LLM)  
llm = ChatOllama(model=MODEL)  
  
def get\_current\_time(\_input=None):  
 """Returns the current time in H:MM AM/PM format."""  
 now = datetime.datetime.now() # Get current time  
 return now.strftime("%I:%M %p") # Format time in H:MM AM/PM format  
  
# Define the tool  
time\_tool = Tool(  
 name="Time",  
 func=get\_current\_time,  
 description="Returns the current time in H:MM AM/PM format."  
)  
  
# Initialize the agent with a list of tools  
agent = initialize\_agent(  
 tools=[time\_tool], # Pass tools as a list  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True  
)  
  
# Test the agent  
response = agent.run("What is current date and time?")  
print(response)

**Calling Maths Functions**

import os  
  
from langchain.agents import AgentExecutor, create\_openai\_tools\_agent  
from langchain\_openai import ChatOpenAI  
  
from langchain.tools import BaseTool, StructuredTool, tool  
from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder  
from langchain.chat\_models import ChatOllama  
  
  
# setup the tools  
@tool  
def add(a: int, b: int) -> int:  
 """Add two numbers."""  
 return a + b  
  
@tool  
def multiply(a: int, b: int) -> int:  
 """Multiply two numbers."""  
 return a \* b  
  
@tool  
def square(a) -> int:  
 """Calculates the square of a number."""  
 a = int(a)  
 return a \* a  
  
prompt = ChatPromptTemplate.from\_messages(  
 [  
 ("system", """You are a mathematical assistant.  
 Use your tools to answer questions. If you do not have a tool to  
 answer the question, say so.   
  
 Return only the answers. e.g  
 Human: What is 1 + 1?  
 AI: 2  
 """),  
 MessagesPlaceholder("chat\_history", optional=True),  
 ("human", "{input}"),  
 MessagesPlaceholder("agent\_scratchpad"),  
 ]  
)  
  
MODEL = "llama3.2"  
  
# Initialize a language model (LLM)  
llm = ChatOllama(model=MODEL)  
  
# setup the toolkit  
toolkit = [add, multiply, square]  
  
# Construct the OpenAI Tools agent  
agent = create\_openai\_tools\_agent(llm, toolkit, prompt)  
  
# Create an agent executor by passing in the agent and tools  
agent\_executor = AgentExecutor(agent=agent, tools=toolkit, verbose=True)  
  
result = agent\_executor.invoke({"input": "what is 1 \* 1?"})  
  
print(result['output'])

**Read File:**

from langchain.tools import tool  
from langchain.agents import initialize\_agent, AgentType  
from langchain.chat\_models import ChatOllama  
import os  
  
# Define a custom tool (function) to read a file  
@tool  
def ReadFile(file\_name: str) -> str:  
 """  
 Reads the contents of a file given its file path.  
 Returns the file content as a string.  
 """  
 try:  
 if not os.path.exists("filepath/"+file\_name):  
 return f"Error: The file '{file\_name}' does not exist."  
   
 with open("filepath/"+file\_name, "r", encoding="utf-8") as file:  
 return file.read()  
   
 except Exception as e:  
 return f"Error reading file: {str(e)}"  
  
# List of tools the agent can use  
tools = [ReadFile]  
  
# Define model (you can change this if needed)  
MODEL = "llama3.2"  
  
# Initialize a language model (LLM)  
llm = ChatOllama(model=MODEL, streaming=True)  
  
# Create the agent  
agent = initialize\_agent(  
 tools=tools,  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True  
)  
  
# Example usage  
response = agent.run("Read the contents of 'example.txt'")  
print(response)

**DuckDuckGo Search**

from langchain.agents import initialize\_agent, AgentType  
from langchain.chat\_models import ChatOllama  
from langchain.tools import Tool  
from langchain\_community.utilities import DuckDuckGoSearchAPIWrapper  
from langchain\_community.tools import DuckDuckGoSearchResults  
import os  
  
# Initialize the DuckDuckGo Search Tool  
wrapper = DuckDuckGoSearchAPIWrapper(region="de-de", time="d", max\_results=2)  
search = DuckDuckGoSearchResults(api\_wrapper=wrapper)  
  
# Define the search tool  
search\_tool = Tool(  
 name="DuckDuckGo Search",  
 func=search.run,  
 description="Use this tool to search for real-time information from DuckDuckGo."  
)  
  
# List of tools the agent can use  
tools = [search\_tool]  
  
# Define the language model  
MODEL = "llama3.2"  
llm = ChatOllama(model=MODEL, streaming=True)  
  
# Initialize the agent  
agent = initialize\_agent(  
 tools=tools,  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True  
)  
  
# Test the agent  
response = agent.run("What are the latest AI trends in 2025?")  
print(response)

**Connect To Github:**

import os  
import requests  
from langchain.agents import initialize\_agent, AgentType  
from langchain.chat\_models import ChatOllama  
from langchain.tools import Tool  
  
# Set GitHub API Token (Replace with your actual token)  
GITHUB\_TOKEN = "\*\*\*\*"  
HEADERS = {"Authorization": f"token {GITHUB\_TOKEN}"}  
GITHUB\_API\_BASE = "https://api.github.com"  
  
### --- GitHub API Helper Functions --- ###  
def search\_repositories(query):  
 """Search GitHub repositories based on a query."""  
 url = f"{GITHUB\_API\_BASE}/search/repositories?q={query}"  
 response = requests.get(url, headers=HEADERS)  
 if response.status\_code == 200:  
 repos = response.json().get("items", [])  
 return [f"{repo['full\_name']} - {repo['html\_url']}" for repo in repos[:5]] # Return top 5 results  
 return f"Error: {response.json()}"  
  
def get\_repo\_details(repo\_name):  
 """Fetch details of a specific GitHub repository."""  
 url = f"{GITHUB\_API\_BASE}/repos/{repo\_name}"  
 response = requests.get(url, headers=HEADERS)  
 if response.status\_code == 200:  
 repo = response.json()  
 return f"Repo: {repo['full\_name']}\nDescription: {repo['description']}\nStars: {repo['stargazers\_count']}\nURL: {repo['html\_url']}"  
 return f"Error: {response.json()}"  
  
def create\_issue(repo\_name, title, body=""):  
 """Create an issue in a specified GitHub repository."""  
 url = f"{GITHUB\_API\_BASE}/repos/{repo\_name}/issues"  
 data = {"title": title, "body": body}  
 response = requests.post(url, json=data, headers=HEADERS)  
 if response.status\_code == 201:  
 return f"Issue created: {response.json()['html\_url']}"  
 return f"Error: {response.json()}"  
  
def list\_issues(repo\_name):  
 """List open issues in a repository."""  
 url = f"{GITHUB\_API\_BASE}/repos/{repo\_name}/issues"  
 response = requests.get(url, headers=HEADERS)  
 if response.status\_code == 200:  
 issues = response.json()  
 return [f"#{issue['number']}: {issue['title']} - {issue['html\_url']}" for issue in issues[:5]]  
 return f"Error: {response.json()}"  
  
### --- LangChain Tools --- ###  
search\_repo\_tool = Tool(  
 name="GitHub Repository Search",  
 func=search\_repositories,  
 description="Search for GitHub repositories using a keyword."  
)  
  
repo\_details\_tool = Tool(  
 name="GitHub Repository Details",  
 func=get\_repo\_details,  
 description="Get details of a specific GitHub repository. Input format: 'owner/repo\_name'."  
)  
  
create\_issue\_tool = Tool(  
 name="GitHub Create Issue",  
 func=lambda inputs: create\_issue(inputs['repo\_name'], inputs['title'], inputs.get('body', '')),  
 description="Create a GitHub issue in a repository. Input should be a dictionary with 'repo\_name', 'title', and optional 'body'."  
)  
  
list\_issues\_tool = Tool(  
 name="GitHub List Issues",  
 func=list\_issues,  
 description="List open issues in a GitHub repository. Input format: 'owner/repo\_name'."  
)  
  
# List of tools for the agent  
tools = [search\_repo\_tool, repo\_details\_tool, create\_issue\_tool, list\_issues\_tool]  
  
### --- Initialize LangChain Agent --- ###  
MODEL = "llama3.2"  
llm = ChatOllama(model=MODEL, streaming=True)  
  
agent = initialize\_agent(  
 tools=tools,  
 llm=llm,  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
 verbose=True  
)  
  
### --- Example Agent Queries --- ###  
print(agent.run("Get details about the repository 'wshamim1/elasticTutorials'"))

**Pandas Dataframe:**

import pandas as pd  
from langchain\_experimental.agents import create\_pandas\_dataframe\_agent  
from langchain.chat\_models import ChatOllama # You can also use OpenAI's GPT  
  
# Load the dataset  
df = pd.read\_csv("Student\_scores.csv")  
  
# Initialize the Chat Model (LLM)  
MODEL = "llama3.2" # You can use OpenAI's model like "gpt-4" if needed  
chat\_model = ChatOllama(model=MODEL, streaming=True)  
  
# Create a Pandas DataFrame Agent  
agent = create\_pandas\_dataframe\_agent(  
 chat\_model,  
 df,  
 verbose=True,  
 allow\_dangerous\_code=True, # Allows execution of complex pandas operations  
 agent\_executor\_kwargs={"handle\_parsing\_errors": True} # Handles any parsing errors  
)  
  
# Example Query  
response = agent.run("Tell me student name with top score in Math.")  
print(response)

Sample Output:

Action: use python\_repl\_ast  
Action Input: `print((df['Math Score'] == df['Math Score'].max()).idxmax())`use python\_repl\_ast is not a valid tool, try one of [python\_repl\_ast].Question: Tell me the student name with top score in Math.  
Thought: To find the student name with the top score in Math, we need to find the index of the row where 'Math Score' equals the maximum value in the 'Math Score' column and then select the corresponding Student Name.  
  
Action: use python\_repl\_ast  
Action Input: `print(df.loc[(df['Math Score'] == df['Math Score'].max()), 'Student Name'])`use python\_repl\_ast is not a valid tool, try one of [python\_repl\_ast].Final Answer: Student 3  
  
> Finished chain.  
Student 3

**Conclusion**

LangChain’s **Tools** and **Agents** enable AI models to go beyond static responses by integrating external capabilities. Whether you are building a smart chatbot, a research assistant, or an automation workflow, understanding these components is key to developing intelligent, dynamic applications.

If you’re new to LangChain, start by experimenting with built-in tools and simple agents, then gradually explore more complex use cases!

In my next blog post, I will share insights about CrewAI.

**If you found this article helpful :**Give it a few claps 👏 and **Follow me on Medium** for more technical articles.