Defining Custom Tools

When constructing your own agent, you will need to provide it with a list of Tools that it can use. Besides the actual function that is called, the Tool consists of several components:

- `name` (str), is required and must be unique within a set of tools provided to an agent
- `description` (str), is optional but recommended, as it is used by an agent to determine tool use
- `args_schema` (Pydantic BaseModel), is optional but recommended, can be used to provide more information (e.g., few-shot examples) or validation for expected parameters.

There are multiple ways to define a tool. In this guide, we will walk through how to do for two functions:

1. A made up search function that always returns the string “LangChain”
2. A multiplier function that will multiply two numbers by eachother

The biggest difference here is that the first function only requires one input, while the second one requires multiple. Many agents only work with functions that require single inputs, so it's important to know how to work with those. For the most part, defining these custom tools is the same, but there are some differences.

```
# Import things that are needed generically
from langchain.pydantic_v1 import BaseModel,
Field
from langchain.tools import BaseTool,
StructuredTool, tool
```

@tool decorator

This `@tool` decorator is the simplest way to define a custom tool. The decorator uses the function name as the tool name by default, but this can be overridden by passing a string as the first argument. Additionally, the decorator will use the function's docstring as the tool's description - so a docstring MUST be provided.

```
@tool
def search(query: str) -> str:
```

```
    """Look up things online."""
    return "LangChain"
```

```
print(search.name)
print(search.description)
print(search.args)
```

```
search
search(query: str) -> str - Look up things
online.
{'query': {'title': 'Query', 'type':
'string'}}
```

```
@tool
def multiply(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b
```

```
print(multiply.name)
print(multiply.description)
print(multiply.args)
```

```
multiply
multiply(a: int, b: int) -> int - Multiply
```

```
two numbers.
{'a': {'title': 'A', 'type': 'integer'}, 'b':
{'title': 'B', 'type': 'integer'}}
```

You can also customize the tool name and JSON args by passing them into the tool decorator.

```
class SearchInput(BaseModel):
    query: str = Field(description="should be
a search query")


@tool("search-tool", args_schema=SearchInput,
return_direct=True)
def search(query: str) -> str:
    """Look up things online."""
    return "LangChain"
```

```
print(search.name)
print(search.description)
print(search.args)
print(search.return_direct)
```

```
search-tool
search-tool(query: str) -> str - Look up
things online.
{'query': {'title': 'Query', 'description':
```

```
'should be a search query', 'type':
'string'}}
True
```

Subclass BaseTool

You can also explicitly define a custom tool by subclassing the BaseTool class. This provides maximal control over the tool definition, but is a bit more work.

```
from typing import Optional, Type

from langchain.callbacks.manager import (
    AsyncCallbackManagerForToolRun,
    CallbackManagerForToolRun,
)


class SearchInput(BaseModel):
    query: str = Field(description="should be
a search query")


class CalculatorInput(BaseModel):
    a: int = Field(description="first
number")
    b: int = Field(description="second
number")
```

```
class CustomSearchTool(BaseTool):
    name = "custom_search"
    description = "useful for when you need
to answer questions about current events"
    args_schema: Type[BaseModel] =
SearchInput

    def _run(
        self, query: str, run_manager:
Optional[CallbackManagerForToolRun] = None
    ) -> str:
        """Use the tool."""
        return "LangChain"

    async def _arun(
        self, query: str, run_manager:
Optional[AsyncCallbackManagerForToolRun] =
None
    ) -> str:
        """Use the tool asynchronously."""
        raise
NotImplementedError("custom_search does not
support async")


class CustomCalculatorTool(BaseTool):
    name = "Calculator"
    description = "useful for when you need
to answer questions about math"
```

```
    args_schema: Type[BaseModel] =
CalculatorInput
    return_direct: bool = True

    def _run(
        self, a: int, b: int, run_manager:
Optional[CallbackManagerForToolRun] = None
    ) -> str:
        """Use the tool."""
        return a * b

    async def _arun(
        self,
        a: int,
        b: int,
        run_manager:
Optional[AsyncCallbackManagerForToolRun] =
None,
    ) -> str:
        """Use the tool asynchronously."""
        raise NotImplementedError("Calculator
does not support async")
```

```
search = CustomSearchTool()
print(search.name)
print(search.description)
print(search.args)
```

```
custom_search
useful for when you need to answer questions
about current events
{'query': {'title': 'Query', 'description':
'should be a search query', 'type':
'string'}}
```

```
multiply = CustomCalculatorTool()
print(multiply.name)
print(multiply.description)
print(multiply.args)
print(multiply.return_direct)
```

```
Calculator
useful for when you need to answer questions
about math
{'a': {'title': 'A', 'description': 'first
number', 'type': 'integer'}, 'b': {'title':
'B', 'description': 'second number', 'type':
'integer'}}
True
```

StructuredTool dataclass

You can also use a `StructuredTool` dataclass. This methods is a mix between the previous two. It's more convenient than inheriting from the BaseTool class, but provides more functionality than just using a decorator.

```
def search_function(query: str):
    return "LangChain"


search = StructuredTool.from_function(
    func=search_function,
    name="Search",
    description="useful for when you need to
answer questions about current events",
    # coroutine= ... <- you can specify an
async method if desired as well
)
```

```
print(search.name)
print(search.description)
print(search.args)
```

```
Search
Search(query: str) - useful for when you need
to answer questions about current events
```

```
{'query': {'title': 'Query', 'type':
'string'}}
```

You can also define a custom `args_schema` to provide more information about inputs.

```
class CalculatorInput(BaseModel):
    a: int = Field(description="first
number")
    b: int = Field(description="second
number")


def multiply(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b


calculator = StructuredTool.from_function(
    func=multiply,
    name="Calculator",
    description="multiply numbers",
    args_schema=CalculatorInput,
    return_direct=True,
    # coroutine= ... <- you can specify an
async method if desired as well
)
```

```
print(calculator.name)
print(calculator.description)
print(calculator.args)
```

```
Calculator
Calculator(a: int, b: int) -> int - multiply
numbers
{'a': {'title': 'A', 'description': 'first
number', 'type': 'integer'}, 'b': {'title':
'B', 'description': 'second number', 'type':
'integer'}}
```

Handling Tool Errors

When a tool encounters an error and the exception is not caught, the agent will stop executing. If you want the agent to continue execution, you can raise a `ToolException` and set `handle_tool_error` accordingly.

When `ToolException` is thrown, the agent will not stop working, but will handle the exception according to the `handle_tool_error` variable of the tool, and the processing result will be returned to the agent as observation, and printed in red.

You can set `handle_tool_error` to `True`, set it a unified string value, or set it as a function. If it's set as a function, the function should take a `ToolException` as a parameter and return a `str` value.

Please note that only raising a `ToolException` won't be effective. You need to first set the `handle_tool_error` of the tool because its default value is `False`.

```
from langchain_core.tools import
ToolException


def search_tool1(s: str):
    raise ToolException("The search tool1 is
not available.")
```

First, let's see what happens if we don't set `handle_tool_error` - it will error.

```
search = StructuredTool.from_function(
    func=search_tool1,
    name="Search_tool1",
    description="A bad tool",
)
```

```
search.run("test")
```

```
ToolException: The search tool1 is not
available.
```

Now, let's set `handle_tool_error` to be True

```
search = StructuredTool.from_function(
    func=search_tool1,
    name="Search_tool1",
    description="A bad tool",
    handle_tool_error=True,
)

search.run("test")
```

```
'The search tool1 is not available.'
```

We can also define a custom way to handle the tool error

```
def _handle_error(error: ToolException) ->
str:
    return (
        "The following errors occurred during
```

```
tool execution:"
        + error.args[0]
        + "Please try another tool."
    )


search = StructuredTool.from_function(
    func=search_tool1,
    name="Search_tool1",
    description="A bad tool",
    handle_tool_error=_handle_error,
)

search.run("test")
```

```
'The following errors occurred during tool
execution:The search tool1 is not
available.Please try another tool.'
```