Using OpenAI functions

This walkthrough demonstrates how to incorporate OpenAI function-calling API's in a chain. We'll go over:

1. How to use functions to get structured outputs from ChatOpenAI
2. How to create a generic chain that uses (multiple) functions
3. How to create a chain that actually executes the chosen function

```
from typing import Optional

from langchain.chains.openai_functions import (
    create_openai_fn_chain,
    create_structured_output_chain,
)
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate,
HumanMessagePromptTemplate
from langchain.schema import HumanMessage, SystemMessage
```

API Reference:

- create_openai_fn_chain from `langchain.chains.openai_functions`
- create_structured_output_chain from `langchain.chains.openai_functions`
- ChatOpenAI from `langchain.chat_models`
- ChatPromptTemplate from `langchain.prompts`
- HumanMessagePromptTemplate from `langchain.prompts`
- HumanMessage from `langchain.schema`
- SystemMessage from `langchain.schema`

Getting structured outputs

We can take advantage of OpenAI functions to try and force the model to return a particular kind of structured output. We'll use the `create_structured_output_chain` to create our chain, which takes the desired structured output either as a Pydantic class or as JsonSchema.

See here for relevant reference docs.

Using Pydantic classes

When passing in Pydantic classes to structure our text, we need to make sure to have a docstring description for the class. It also helps to have descriptions for each of the classes attributes.

```
from pydantic import BaseModel, Field


class Person(BaseModel):
    """Identifying information about a person."""

    name: str = Field(..., description="The person's name")
    age: int = Field(..., description="The person's age")
    fav_food: Optional[str] = Field(None, description="The person's
favorite food")
```

```
# If we pass in a model explicitly, we need to make sure it supports
the OpenAI function-calling API.
llm = ChatOpenAI(model="gpt-4", temperature=0)
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a world class algorithm for extracting
information in structured formats."),
        ("human", "Use the given format to extract information from the
following input: {input}"),
        ("human", "Tip: Make sure to answer in the correct format"),
    ]
)

chain = create_structured_output_chain(Person, llm, prompt,
verbose=True)
chain.run("Sally is 13")
```

```



    > Entering new LLMChain chain...
    Prompt after formatting:
    System: You are a world class algorithm for extracting information
in structured formats.
    Human: Use the given format to extract information from the
```

```
following input: Sally is 13
    Human: Tip: Make sure to answer in the correct format

    > Finished chain.





    Person(name='Sally', age=13, fav_food='Unknown')
```

To extract arbitrarily many structured outputs of a given format, we can just create a wrapper Pydantic class that takes a sequence of the original class.

```
from typing import Sequence


class People(BaseModel):
    """Identifying information about all people in a text."""

    people: Sequence[Person] = Field(..., description="The people in
the text")


chain = create_structured_output_chain(People, llm, prompt,
verbose=True)
chain.run(
    "Sally is 13, Joey just turned 12 and loves spinach. Caroline is 10
years older than Sally."
)
```

```



    > Entering new LLMChain chain...
    Prompt after formatting:
    System: You are a world class algorithm for extracting information
in structured formats.
    Human: Use the given format to extract information from the
following input: Sally is 13, Joey just turned 12 and loves spinach.
Caroline is 10 years older than Sally.
    Human: Tip: Make sure to answer in the correct format

    > Finished chain.
```

```
    People(people=[Person(name='Sally', age=13, fav_food=''),
Person(name='Joey', age=12, fav_food='spinach'),
Person(name='Caroline', age=23, fav_food='')])
```

Using JsonSchema

We can also pass in JsonSchema instead of Pydantic classes to specify the desired structure. When we do this, our chain will output json corresponding to the properties described in the JsonSchema, instead of a Pydantic class.

```
json_schema = {
    "title": "Person",
    "description": "Identifying information about a person.",
    "type": "object",
    "properties": {
        "name": {"title": "Name", "description": "The person's name",
"type": "string"},
        "age": {"title": "Age", "description": "The person's age",
"type": "integer"},
        "fav_food": {
            "title": "Fav Food",
            "description": "The person's favorite food",
            "type": "string",
        },
    },
    "required": ["name", "age"],
}
```

```
chain = create_structured_output_chain(json_schema, llm, prompt,
verbose=True)
chain.run("Sally is 13")
```

```
    > Entering new LLMChain chain...
    Prompt after formatting:
    System: You are a world class algorithm for extracting information
in structured formats.
    Human: Use the given format to extract information from the
following input: Sally is 13
```

```
Human: Tip: Make sure to answer in the correct format

> Finished chain.




{'name': 'Sally', 'age': 13}
```

Creating a generic OpenAI functions chain

To create a generic OpenAI functions chain, we can use the `create_openai_fn_chain` method. This is the same as `create_structured_output_chain` except that instead of taking a single output schema, it takes a sequence of function definitions.

Functions can be passed in as:

- dicts conforming to OpenAI functions spec,
- Pydantic classes, in which case they should have docstring descriptions of the function they represent and descriptions for each of the parameters,
- Python functions, in which case they should have docstring descriptions of the function and args, along with type hints.

See here for relevant reference docs.

Using Pydantic classes

```
class RecordPerson(BaseModel):
    """Record some identifying information about a pe."""

    name: str = Field(..., description="The person's name")
    age: int = Field(..., description="The person's age")
    fav_food: Optional[str] = Field(None, description="The person's
favorite food")


class RecordDog(BaseModel):
    """Record some identifying information about a dog."""

    name: str = Field(..., description="The dog's name")
    color: str = Field(..., description="The dog's color")
```

```
    fav_food: Optional[str] = Field(None, description="The dog's
favorite food")
```

```
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a world class algorithm for recording
entities."),
        ("human", "Make calls to the relevant function to record the
entities in the following input: {input}"),
        ("human", "Tip: Make sure to answer in the correct format"),
    ]
)

chain = create_openai_fn_chain([RecordPerson, RecordDog], llm, prompt,
verbose=True)
chain.run("Harry was a chubby brown beagle who loved chicken")
```

```
    > Entering new LLMChain chain...
    Prompt after formatting:
    System: You are a world class algorithm for recording entities.
    Human: Make calls to the relevant function to record the entities
in the following input: Harry was a chubby brown beagle who loved
chicken
    Human: Tip: Make sure to answer in the correct format

    > Finished chain.





    RecordDog(name='Harry', color='brown', fav_food='chicken')
```

Using Python functions

We can pass in functions as Pydantic classes, directly as OpenAI function dicts, or Python functions. To pass Python function in directly, we'll want to make sure our parameters have type hints, we have a docstring, and we use Google Python style docstrings to describe the parameters.

NOTE: To use Python functions, make sure the function arguments are of primitive types (str, float, int, bool) or that they are Pydantic objects.

```
class OptionalFavFood(BaseModel):
    """Either a food or null."""

    food: Optional[str] = Field(
        None,
        description="Either the name of a food or null. Should be null 
if the food isn't known.",
    )


def record_person(name: str, age: int, fav_food: OptionalFavFood) -> 
str:
    """Record some basic identifying information about a person.

    Args:
        name: The person's name.
        age: The person's age in years.
        fav_food: An OptionalFavFood object that either contains the 
person's favorite food or a null value. Food should be null if it's not 
known.
    """
    return f"Recording person {name} of age {age} with favorite food 
{fav_food.food}!"


chain = create_openai_fn_chain([record_person], llm, prompt, 
verbose=True)
chain.run(
    "The most important thing to remember about Tommy, my 12 year old, 
is that he'll do anything for apple pie."
)
```

```



    > Entering new LLMChain chain...
    Prompt after formatting:
    System: You are a world class algorithm for recording entities.
    Human: Make calls to the relevant function to record the entities 
in the following input: The most important thing to remember about 
Tommy, my 12 year old, is that he'll do anything for apple pie.
    Human: Tip: Make sure to answer in the correct format
```

```
    > Finished chain.





    {'name': 'Tommy', 'age': 12, 'fav_food': {'food': 'apple pie'}}
```

If we pass in multiple Python functions or OpenAI functions, then the returned output will be of the form

```
{"name": "<<function_name>>", "arguments": {<<function_arguments>>}}
```

```
def record_dog(name: str, color: str, fav_food: OptionalFavFood) ->
str:
    """Record some basic identifying information about a dog.

    Args:
        name: The dog's name.
        color: The dog's color.
        fav_food: An OptionalFavFood object that either contains the
dog's favorite food or a null value. Food should be null if it's not
known.
    """
    return f"Recording dog {name} of color {color} with favorite food
{fav_food}!"


chain = create_openai_fn_chain([record_person, record_dog], llm,
prompt, verbose=True)
chain.run(
    "I can't find my dog Henry anywhere, he's a small brown beagle.
Could you send a message about him?"
)
```

```



    > Entering new LLMChain chain...
    Prompt after formatting:
    System: You are a world class algorithm for recording entities.
    Human: Make calls to the relevant function to record the entities
in the following input: I can't find my dog Henry anywhere, he's a
small brown beagle. Could you send a message about him?
    Human: Tip: Make sure to answer in the correct format
```

```
    > Finished chain.




    {'name': 'record_dog',
     'arguments': {'name': 'Henry', 'color': 'brown', 'fav_food':
{'food': None}}}
```

Other Chains using OpenAI functions

There are a number of more specific chains that use OpenAI functions.

- Extraction: very similar to structured output chain, intended for information/entity extraction specifically.
- Tagging: tag inputs.
- OpenAPI: take an OpenAPI spec and create + execute valid requests against the API, using OpenAI functions under the hood.
- QA with citations: use OpenAI functions ability to extract citations from text.