LangChain is a framework designed to help developers build more powerful, flexible, and context-aware applications using language models by **abstracting and simplifying common tasks**.

It is focused on making it easier to develop, manage, and deploy applications that use LLMs by offering tools and abstractions for common tasks.

It's particularly useful for applications that require a combination of language model capabilities, external data interactions, and more complex workflows.

Here are key features and aspects of LangChain:

- **Chains**: LangChain provides abstractions for building chains of operations that can process inputs, pass them through various components, and generate outputs. A "chain" could involve multiple steps like **querying an LLM**, interacting with **external APIs**, or applying **custom logic**.
- **Prompts**: LangChain makes it easier to manage and format prompts that are sent to language models. It provides prompt templates, allowing you to structure and manage prompts dynamically, making them more reusable and adaptable.
- **Memory**: LangChain supports the concept of memory, allowing language models to remember context across multiple interactions. This is important for building conversational agents that need to track previous interactions over time.
- **Agents**: LangChain includes agents that allow LLMs to make decisions based on context. For example, an agent can decide whether it needs to call an external service, make a query, or simply generate a response based on the available input.
- **Integrations**: LangChain provides integrations with various data sources and APIs, such as databases, web scraping tools, and other external systems, making it easier for LLMs to fetch and utilize dynamic data in their responses.
- **Tooling and Utilities**: LangChain includes utility functions for common tasks like text summarization, question answering, document retrieval, and data parsing, allowing developers to use pre-built functionality rather than reimplementing these tasks.

Langchain Exercises

Install Package

```
pip install langchain
pip install langchain-openai
pip install langchain-community
```

1. Exercise: Chat Model for Translation

This exercise demonstrates how to use the ChatOpenAI model to translate a sentence from English to French.

```
from langchain_openai import ChatOpenAI
from util import getToken
from dotenv import load_dotenv

load_dotenv()
token = getToken()
load_dotenv()

# Initialize the OpenAI model
llm = ChatOpenAI(
    model="gpt-4o-2024-05-13",
    temperature=0,
    max_tokens=None,
    timeout=None,
    max_retries=2,
)

# Create a chain that combines the prompt and the model
messages = [
    (
        "system",
        "You are a helpful assistant that translates English to French. Translate the user sentence.",
    ),
    ("user", "I love programming."),
]
response = llm.invoke(messages)
print(response.content);
```

2. Using Prompt Templates

```
from langchain_openai import ChatOpenAI
from util import getToken
import os
from dotenv import load_dotenv
from langchain_core.prompts import ChatPromptTemplate

load_dotenv()
token = getToken()
load_dotenv()
```

```
# Initialize the OpenAI model
llm = ChatOpenAI(
    model="gpt-4o-2024-05-13",
    temperature=0,
    max_tokens=None,
    timeout=None,
    max_retries=2,
)

# Create a chain that combines the prompt and the model
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are a helpful assistant that translates {input_language} to {output_language}.",
        ),
        ("user", "{user_input}"),
    ]
)
parameter_values = {
    "input_language": "English",
    "output_language": "German",
    "user_input": "I love programming.",
}
# Invoke the chain with specific input parameters
chain = prompt | llm
response = chain.invoke(parameter_values, prompt=prompt)

print(response.content)
```

3. Exercise to Demo Chaining

```
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.schema import StrOutputParser
from langchain.schema.runnable import RunnablePassthrough
from langchain_openai import ChatOpenAI
from util import getToken
from dotenv import load_dotenv
from langchain_core.prompts import ChatPromptTemplate
import json

load_dotenv()
token = getToken()
load_dotenv()

topic = "A Lion and a Mouse"

title_prompt = PromptTemplate.from_template(
  "Write a great title for a story about {topic}"
)

story_prompt = PromptTemplate.from_template(
"""You are a writer. Given the title of story, it is your job to write a story for that title.
Title: {title}"""
)

llm = ChatOpenAI(model="gpt-4o-2024-05-13")

title_chain = title_prompt | llm | StrOutputParser()
story_chain = story_prompt | llm | StrOutputParser()

chain = {"title": title_chain} | RunnablePassthrough.assign(story=story_chain)

result = chain.invoke({"topic": topic})

print(result['title'])
print("=" * 100)
print(result['story'])
```

4. Exercise: Embedding Documents

This exercise shows how to embed documents using the OpenAIEmbeddings model.

```
from langchain_openai import OpenAIEmbeddings
from langchain.evaluation import load_evaluator, EmbeddingDistance

embeddings_model = OpenAIEmbeddings(model="text-embedding-3-large", dimensions=5)
```

```
items = [
    "Hi there!",
    "Oh, hello!",
    "What's your name?",
    "My friends call me World",
    "Hello World!",
  ]

# Getting Embeddings for all items
embeddings = embeddings_model.embed_documents(items)
for i, embedding in enumerate(embeddings):
  print(f"Embedding {i+1}: {embedding}")

# Calculate cosine similarities using langchain
evaluator = load_evaluator("embedding_distance", embeddings=embeddings_model,
distance_metric=EmbeddingDistance.EUCLIDEAN)
print(evaluator.evaluate_strings(prediction="I shall go", reference="I shall go"))
```

Alternative for distance:

```
pip install rapidfuzz
```

Add the following to python code:

```
evaluator = load_evaluator("string_distance")
print(evaluator.evaluate_strings(prediction="I shall go", reference="I shall go"))
```

Reading PDF and Chunking

```
pip install pypdf
```

```
from util import getToken
from dotenv import load_dotenv
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
import os

load_dotenv()
```

```python
token = getToken()
load_dotenv()

def main():
  # 1. Ask for a file path in console
  file_path = input("Please enter the path to the PDF file: ")

  if not file_path.endswith(".pdf"):
    print("Only PDF files are allowed")
    return

  if not os.path.exists(file_path):
    print("File does not exist")
    return

  try:
    loader = PyPDFLoader(file_path)
    text_content = loader.load()
    print("PDF Content Read Successfully")

    # 3. Split the PDF into multiple chunks
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=3000, chunk_overlap=100)
    chunks = text_splitter.split_documents(text_content)

    # 4. Loop and display the content in console
    for i, chunk in enumerate(chunks):
      print(f"Chunk {i + 1}:")
      print(chunk.page_content)
      print("-" * 80)

  except Exception as e:
    print(f"An error occurred: {e}")

if __name__ == "__main__":
  main()
```

```python
from util import getToken
from dotenv import load_dotenv
import os
from dotenv import load_dotenv
from opensearchpy import OpenSearch
from langchain_community.vectorstores import OpenSearchVectorSearch
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import CharacterTextSplitter
from langchain_community.document_loaders import PyPDFLoader

load_dotenv()
token = getToken()
load_dotenv()

host = os.environ.get('OPENSEARCH_HOST')
port = os.environ.get('OPENSEARCH_PORT')
username = os.environ.get('OPENSEARCH_USERNAME')
password = os.environ.get('OPENSEARCH_PASSWORD')

# OpenSearch configuration
OPENSEARCH_CONFIG = {
    "hosts": [{"host": host, "port": port}],
    "http_auth": (username, password),
    "http_compress": True,
    "use_ssl": True,
    "verify_certs": False,
    "ssl_assert_hostname": False,
    "ssl_show_warn": False
}

INDEX_NAME = "files"

# Main function to generate embeddings and insert documents
def main():
    embeddings_model = OpenAIEmbeddings(model="text-embedding-3-large")

    # 1. Ask for a file path in console
    file_path = input("Please enter the path to the PDF file: ")
    #file_path = "D:\onedrive\OneDrive - Deccansoft Software Services\1-StudyMaterial\AI-102\1-Get started with Azure AI Services\1-Introduction.pdf"
    loader = PyPDFLoader(file_path)
    text_content = loader.load()
    print("PDF Content Read Successfully")

    # 3. Split the PDF into multiple chunks
    text_splitter = CharacterTextSplitter(chunk_size=3000, chunk_overlap=100)
    chunks = text_splitter.split_documents(text_content)

    docsearch = OpenSearchVectorSearch.from_documents(
        chunks, embeddings_model,
        # is_appx_search=False,
        bulk_size=1000,
        opensearch_url= "https://localhost:9200",
        verify_certs=False,
        http_auth=(username, password)
    )

    query = "Challenges of GenAI"
    #performing similarity search
    results = docsearch.similarity_search(query, k=1)

    print(results[0].page_content)
    #print(results[1].page_content)

if __name__ == "__main__":
    main()
```