GEN AI_HANDSON

NAME : CHITRA MADARAKHANDI

SRN : PES2UG23CS157

SECTION : C

Problem Statement

Build a system using LangChain that accepts a movie name, retrieves its release year, and calculates how many years ago it was released using an LLM pipeline.

Abstract

This assignment explores the use of LangChain and Prompt Engineering to build modular and composable AI pipelines using Google Gemini models. The objective was to understand how structured prompting, temperature control, few-shot learning, and LCEL (LangChain Expression Language) improve AI system design. The implementation demonstrates how prompts influence model behavior and how LCEL enables clean chaining of prompt templates, models, and output parsers.

1. Understanding LangChain

LangChain provides a standardized interface to interact with LLMs. It abstracts model APIs and enables chaining components such as:

- Prompt Templates
- LLM Models
- Output Parsers

Using LCEL (| operator), I built composable pipelines that pass data from one component to another.

2. Prompt Engineering Concepts Learned

- **Temperature Control:**
  Lower temperature gives deterministic output; higher temperature increases creativity.
- **Structured Prompting (CO-STAR Framework):**
  Clear context, objective, constraints, style, and output format reduce ambiguity.

- **Zero-shot vs Few-shot Learning:**
  Few-shot prompting improves output quality by providing structured examples.
- **Data Quality Importance:**
  Poor examples lead to poor output patterns due to in-context learning behavior.

3. What I Built

1. A structured prompt to generate a recursive Python function.
2. A comparison between lazy prompts and structured prompts.
3. Few-shot learning demonstrations showing pattern influence.
4. A composable LCEL chain that:
   - Takes a movie name
   - Retrieves its release year
   - Calculates how many years ago it was released

4. Observations

- Ambiguity leads to unpredictable model output.
- Clear constraints improve reliability.
- Few-shot examples strongly influence tone and format.
- LCEL makes AI logic clean and reusable.
- API usage limits must be managed carefully.

Sample Output

Movie: Inception

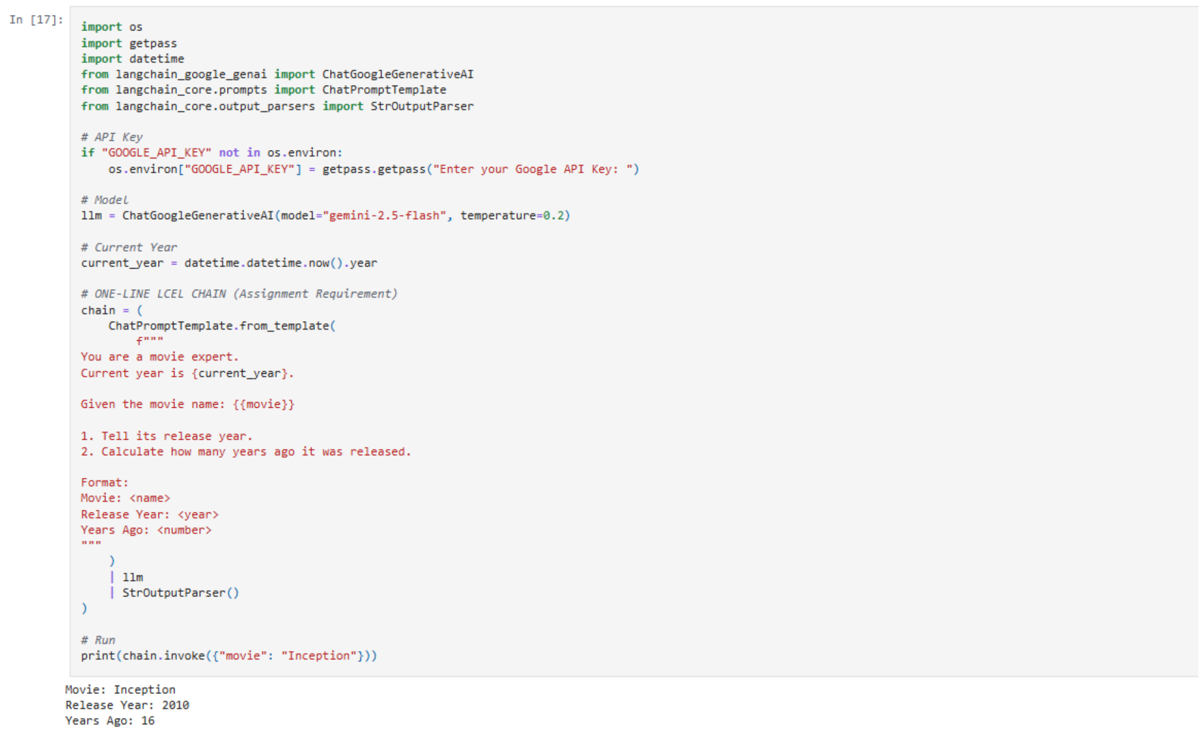
Release Year: 2010

Years Ago: 16

O/P SCREENSHOTS :

1_LangChain_Foundation (2).ipynb

→Assignments

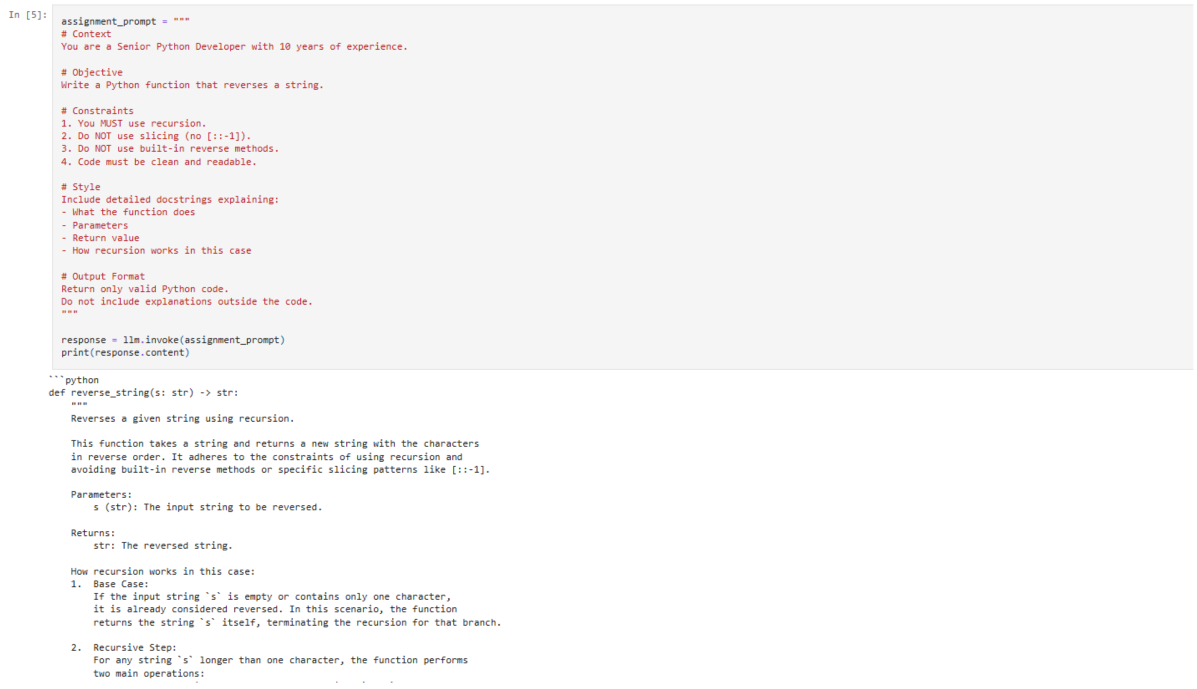


```
In [17]: import os
         import getpass
         import datetime
         from langchain_google_genai import ChatGoogleGenerativeAI
         from langchain_core.prompts import ChatPromptTemplate
         from langchain_core.output_parsers import StrOutputParser

         # API Key
         if "GOOGLE_API_KEY" not in os.environ:
             os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter your Google API Key: ")

         # Model
         llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.2)

         # Current Year
         current_year = datetime.datetime.now().year

         # ONE-LINE LCEL CHAIN (Assignment Requirement)
         chain = (
             ChatPromptTemplate.from_template(
                 f"""
         You are a movie expert.
         Current year is {current_year}.

         Given the movie name: {{movie}}

         1. Tell its release year.
         2. Calculate how many years ago it was released.

         Format:
         Movie: <name>
         Release Year: <year>
         Years Ago: <number>
         """
             )
             | llm
             | StrOutputParser()
         )

         # Run
         print(chain.invoke({"movie": "Inception"}))
```

```
Movie: Inception
Release Year: 2010
Years Ago: 16
```

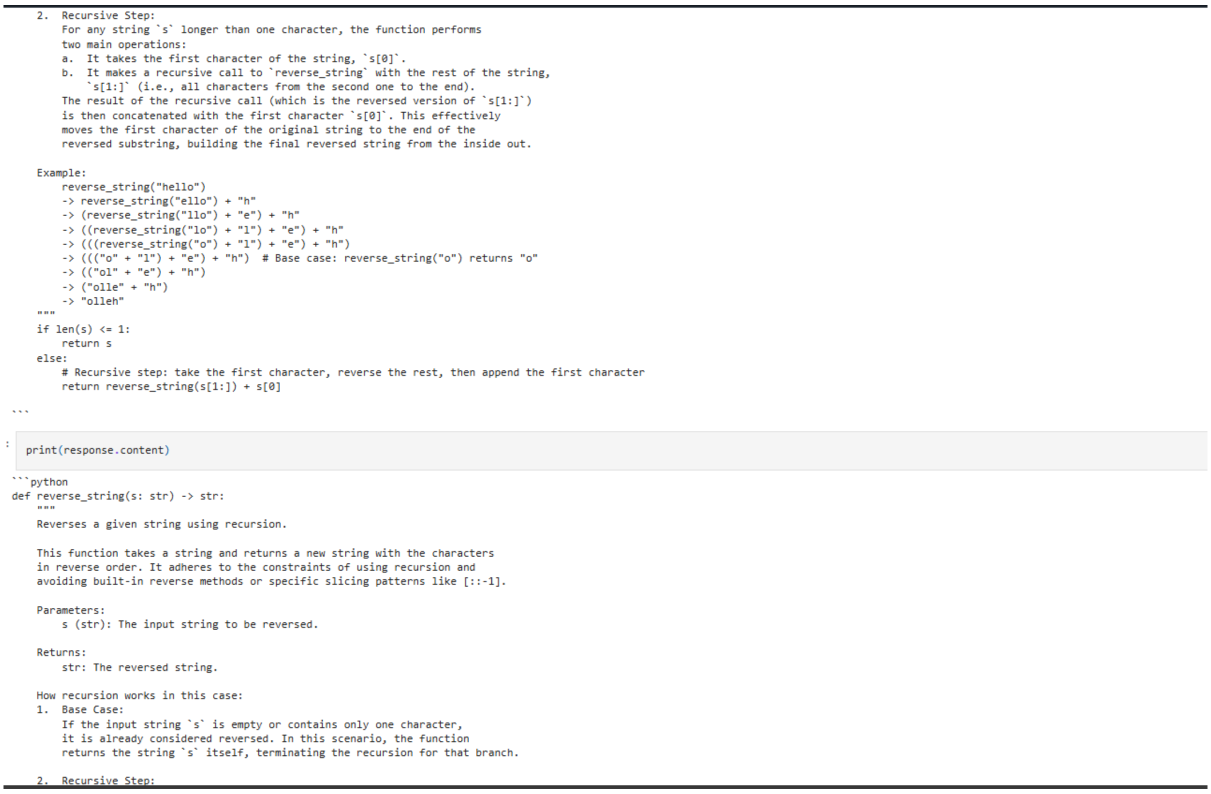
2_Prompt_Engineering_(1).ipynb

→Assignment

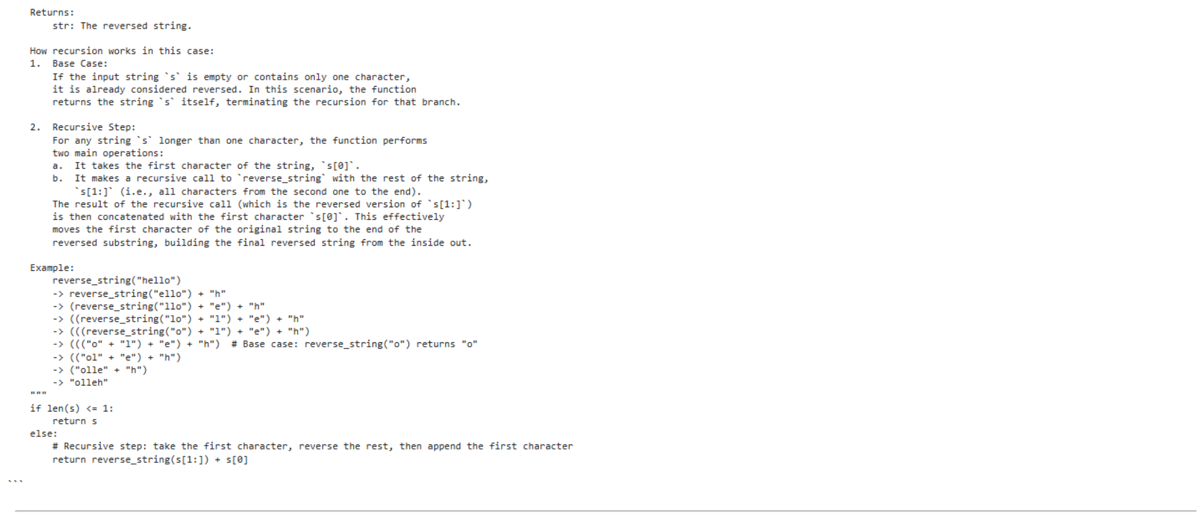


````
In [5]: assignment_prompt = """
        # Context
        You are a Senior Python Developer with 10 years of experience.

        # Objective
        Write a Python function that reverses a string.

        # Constraints
        1. You MUST use recursion.
        2. Do NOT use slicing (no [::-1]).
        3. Do NOT use built-in reverse methods.
        4. Code must be clean and readable.

        # Style
        Include detailed docstrings explaining:
        - What the function does
        - Parameters
        - Return value
        - How recursion works in this case

        # Output Format
        Return only valid Python code.
        Do not include explanations outside the code.
        """

        response = llm.invoke(assignment_prompt)
        print(response.content)
````

~~~
```python
def reverse_string(s: str) -> str:
    """
    Reverses a given string using recursion.

    This function takes a string and returns a new string with the characters
    in reverse order. It adheres to the constraints of using recursion and
    avoiding built-in reverse methods or specific slicing patterns like [::-1].

    Parameters:
        s (str): The input string to be reversed.

    Returns:
        str: The reversed string.

    How recursion works in this case:
    1.  Base Case:
        If the input string `s` is empty or contains only one character,
        it is already considered reversed. In this scenario, the function
        returns the string `s` itself, terminating the recursion for that branch.

    2.  Recursive Step:
        For any string `s` longer than one character, the function performs
        two main operations:
~~~



```
    2.  Recursive Step:
        For any string `s` longer than one character, the function performs
        two main operations:
        a.  It takes the first character of the string, `s[0]`.
        b.  It makes a recursive call to `reverse_string` with the rest of the string,
            `s[1:]` (i.e., all characters from the second one to the end).
        The result of the recursive call (which is the reversed version of `s[1:]`)
        is then concatenated with the first character `s[0]`. This effectively
        moves the first character of the original string to the end of the
        reversed substring, building the final reversed string from the inside out.

    Example:
        reverse_string("hello")
        -> reverse_string("ello") + "h"
        -> (reverse_string("llo") + "e") + "h"
        -> ((reverse_string("lo") + "l") + "e") + "h"
        -> (((reverse_string("o") + "l") + "e") + "h")
        -> ((("o" + "l") + "e") + "h")  # Base case: reverse_string("o") returns "o"
        -> (("ol" + "e") + "h")
        -> ("olle" + "h")
        -> "olleh"
    """
    if len(s) <= 1:
        return s
    else:
        # Recursive step: take the first character, reverse the rest, then append the first character
        return reverse_string(s[1:]) + s[0]
```
```

```
:
print(response.content)
~~~

```python
def reverse_string(s: str) -> str:
    """
    Reverses a given string using recursion.

    This function takes a string and returns a new string with the characters
    in reverse order. It adheres to the constraints of using recursion and
    avoiding built-in reverse methods or specific slicing patterns like [::-1].

    Parameters:
        s (str): The input string to be reversed.

    Returns:
        str: The reversed string.

    How recursion works in this case:
    1.  Base Case:
        If the input string `s` is empty or contains only one character,
        it is already considered reversed. In this scenario, the function
        returns the string `s` itself, terminating the recursion for that branch.

    2.  Recursive Step:
~~~



```
    Returns:
        str: The reversed string.

    How recursion works in this case:
    1. Base Case:
        If the input string `s` is empty or contains only one character,
        it is already considered reversed. In this scenario, the function
        returns the string `s` itself, terminating the recursion for that branch.

    2. Recursive Step:
        For any string `s` longer than one character, the function performs
        two main operations:
        a. It takes the first character of the string, `s[0]`.
        b. It makes a recursive call to `reverse_string` with the rest of the string,
            `s[1:]` (i.e., all characters from the second one to the end).
        The result of the recursive call (which is the reversed version of `s[1:]`)
        is then concatenated with the first character `s[0]`. This effectively
        moves the first character of the original string to the end of the
        reversed substring, building the final reversed string from the inside out.

    Example:
        reverse_string("hello")
        -> reverse_string("ello") + "h"
        -> (reverse_string("llo") + "e") + "h"
        -> ((reverse_string("lo") + "l") + "e") + "h"
        -> (((reverse_string("o") + "l") + "e") + "h")
        -> ((("o" + "l") + "e") + "h") # Base case: reverse_string("o") returns "o"
        -> (("ol" + "e") + "h")
        -> ("olle" + "h")
        -> "olleh"
    """
    if len(s) <= 1:
        return s
    else:
        # Recursive step: take the first character, reverse the rest, then append the first character
        return reverse_string(s[1:]) + s[0]
```
```
```