

AI ASSISTED CODING

LAB ASSIGNMENT 4.2

NAME : KOUDAGANI SAHITHYA

HALLTICKET NUMBER : 2403A52063

BATCH NO : 02

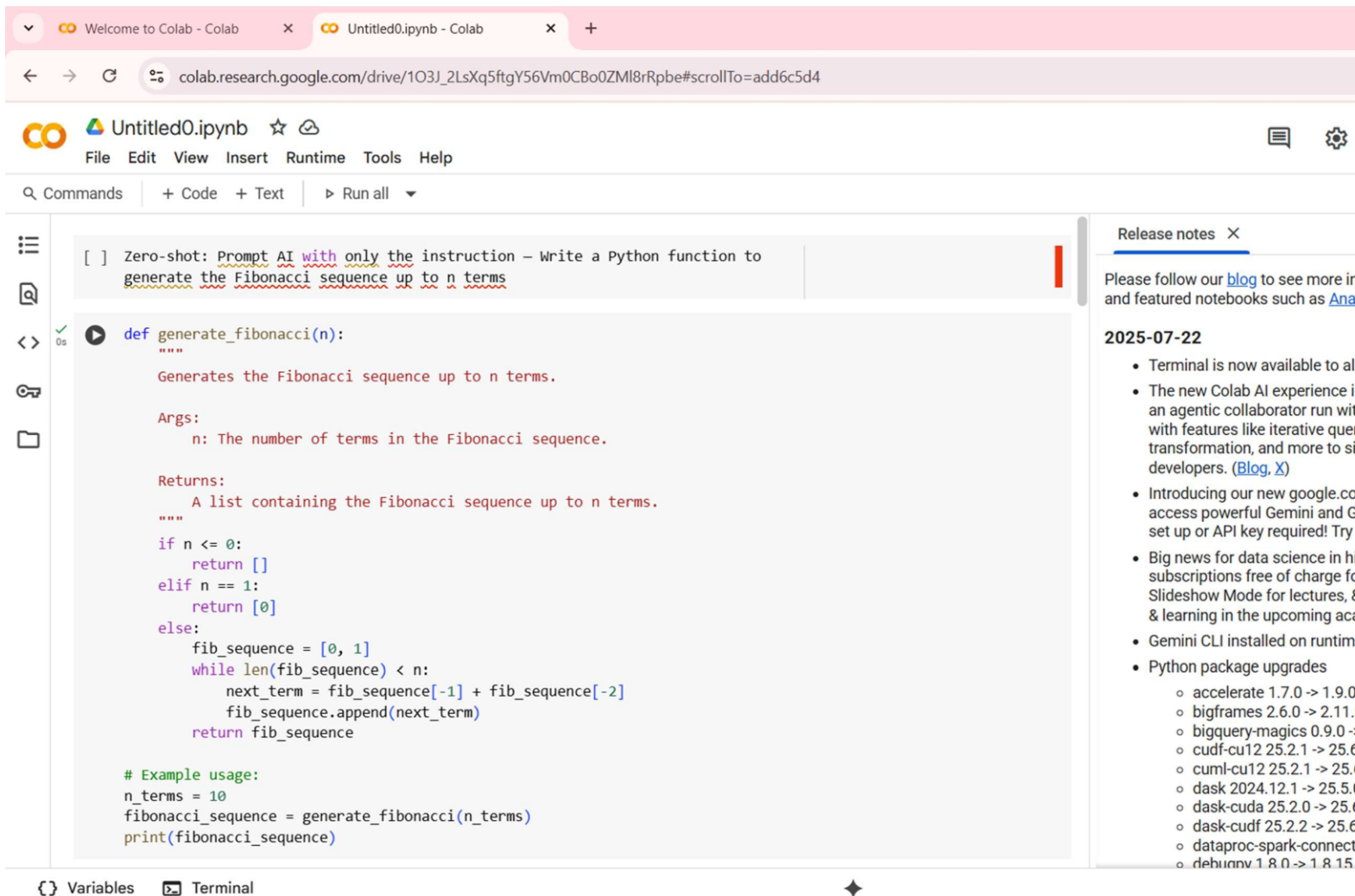
TASK 1:

Zero-shot: Prompt AI with only the instruction — Write a Python function to generate the Fibonacci sequence up to n terms

PROMPT :

Write a Python function named `fibonacci_sequence(n)` that generates the Fibonacci sequence up to n terms. The function should return a list containing the Fibonacci numbers.

CODE :



The screenshot shows a Google Colab notebook interface. The browser address bar displays the URL: `colab.research.google.com/drive/1O3J_2LsXq5ftgY56Vm0CBo0ZMl8rRpbe#scrollTo=add6c5d4`. The notebook is titled "Untitled0.ipynb - Colab". The menu bar includes "File", "Edit", "View", "Insert", "Runtime", "Tools", and "Help". Below the menu bar, there are tabs for "Commands", "+ Code", "+ Text", and "Run all". The main code cell contains the following Python code:

```
[ ] Zero-shot: Prompt AI with only the instruction — Write a Python function to generate the Fibonacci sequence up to n terms
```

```
def generate_fibonacci(n):  
    """  
    Generates the Fibonacci sequence up to n terms.  
  
    Args:  
        n: The number of terms in the Fibonacci sequence.  
  
    Returns:  
        A list containing the Fibonacci sequence up to n terms.  
    """  
    if n <= 0:  
        return []  
    elif n == 1:  
        return [0]  
    else:  
        fib_sequence = [0, 1]  
        while len(fib_sequence) < n:  
            next_term = fib_sequence[-1] + fib_sequence[-2]  
            fib_sequence.append(next_term)  
        return fib_sequence  
  
# Example usage:  
n_terms = 10  
fibonacci_sequence = generate_fibonacci(n_terms)  
print(fibonacci_sequence)
```

On the right side of the notebook, there is a "Release notes" panel with the following content:

Release notes X

Please follow our [blog](#) to see more in and featured notebooks such as [Ana](#)

2025-07-22

- Terminal is now available to all
- The new Colab AI experience is an agentic collaborator run with features like iterative question transformation, and more to developers. ([Blog](#), [X](#))
- Introducing our new google.colab access powerful Gemini and Google set up or API key required! Try
- Big news for data science in h subscriptions free of charge for Slideshow Mode for lectures, & learning in the upcoming ac
- Gemini CLI installed on runtime
- Python package upgrades
 - accelerate 1.7.0 -> 1.9.0
 - bigframes 2.6.0 -> 2.11.
 - bigquery-magics 0.9.0 ->
 - cudf-cu12 25.2.1 -> 25.6
 - cuml-cu12 25.2.1 -> 25.
 - dask 2024.12.1 -> 25.5.
 - dask-cuda 25.2.0 -> 25.6
 - dask-cudf 25.2.2 -> 25.6
 - dataprocs-spark-connect
 - dehumanv 1.8.0 -> 1.8.15

EXPLANATION:

This cell contains a markdown comment that asks for an explanation of the code above it. It doesn't contain any executable code itself.

OBSERVATION :

This cell contains a markdown comment that asks for an explanation of the code above it. It doesn't contain any executable code itself.

TASK 2:

One-shot: Provide one example: Input: 100, Output: 37.78 to help AI generate a function that converts Fahrenheit to Celsius.

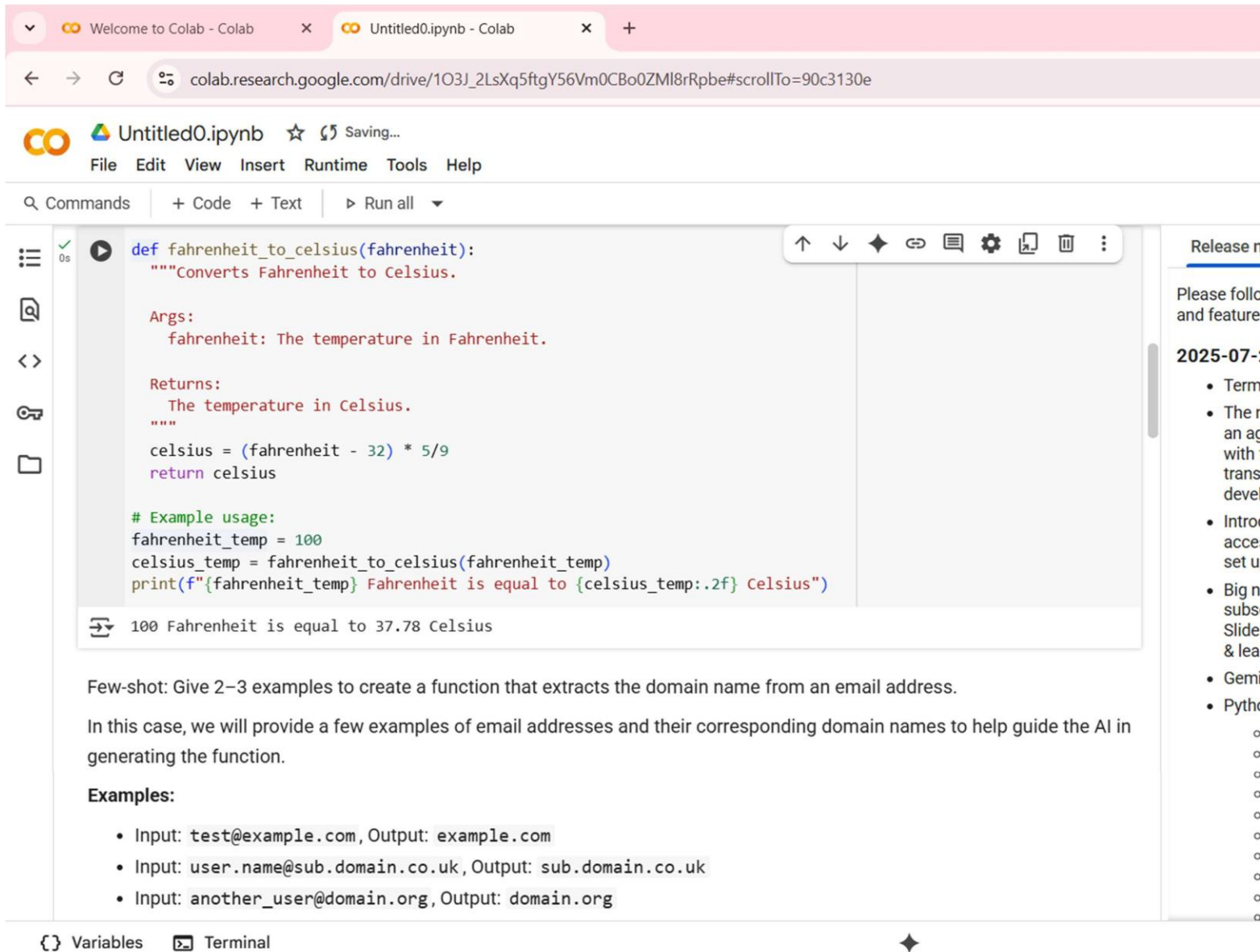
PROMPT :

The function should take a single argument, `fahrenheit`, which is a float representing the temperature in Fahrenheit, and return the corresponding temperature in Celsius as a float.

For example:

- Input: 100
- Output: 37.78

CODE :



The screenshot shows a Google Colab notebook interface. The browser tabs at the top are 'Welcome to Colab - Colab' and 'Untitled0.ipynb - Colab'. The address bar shows the URL: `colab.research.google.com/drive/1O3J_2LsXq5ftgY56Vm0CBo0ZMI8rRpbe#scrollTo=90c3130e`. The notebook title is 'Untitled0.ipynb' with a 'Saving...' status. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu is a toolbar with 'Commands', '+ Code', '+ Text', and 'Run all'. The main code cell contains a Python function `fahrenheit_to_celsius` that takes a temperature in Fahrenheit and returns it in Celsius. The function includes docstrings for arguments and returns, and an example usage block. The output of the function is displayed below the code cell.

```
def fahrenheit_to_celsius(fahrenheit):  
    """Converts Fahrenheit to Celsius.  
  
    Args:  
        fahrenheit: The temperature in Fahrenheit.  
  
    Returns:  
        The temperature in Celsius.  
    """  
    celsius = (fahrenheit - 32) * 5/9  
    return celsius  
  
# Example usage:  
fahrenheit_temp = 100  
celsius_temp = fahrenheit_to_celsius(fahrenheit_temp)  
print(f"{fahrenheit_temp} Fahrenheit is equal to {celsius_temp:.2f} Celsius")
```

100 Fahrenheit is equal to 37.78 Celsius

Few-shot: Give 2–3 examples to create a function that extracts the domain name from an email address.

In this case, we will provide a few examples of email addresses and their corresponding domain names to help guide the AI in generating the function.

Examples:

- Input: `test@example.com`, Output: `example.com`
- Input: `user.name@sub.domain.co.uk`, Output: `sub.domain.co.uk`
- Input: `another_user@domain.org`, Output: `domain.org`

EXPLANATION:

This seems to be a request for me to provide an explanation, likely of the surrounding code cells or a previous output.

OBSERVATION:

In general, providing examples (one-shot or few-shot) can help the AI generate more tailored and accurate code by giving it a better understanding of the specific task and desired output format, especially when the task is more complex or ambiguous. Zero-shot prompting is useful for simpler, well-defined tasks or when you want to see the AI's general understanding of a concept. Based on the code and markdown cells in the notebook, here are some observations about the different prompting techniques demonstrated:

- **Zero-shot prompting:** As seen with the Fibonacci sequence example, the AI can generate functional code with just a clear instruction, but it might require more specific instructions for complex tasks to ensure the output meets expectations.
- **One-shot prompting:** The Fahrenheit to Celsius conversion example shows that providing just one example can be enough to guide the AI to the correct formula and implementation, especially for straightforward tasks with a clear input-output relationship.

Few-shot prompting: The email domain extraction and palindrome checking examples demonstrate that providing a few relevant examples can help the AI understand the desired behavior and edge cases

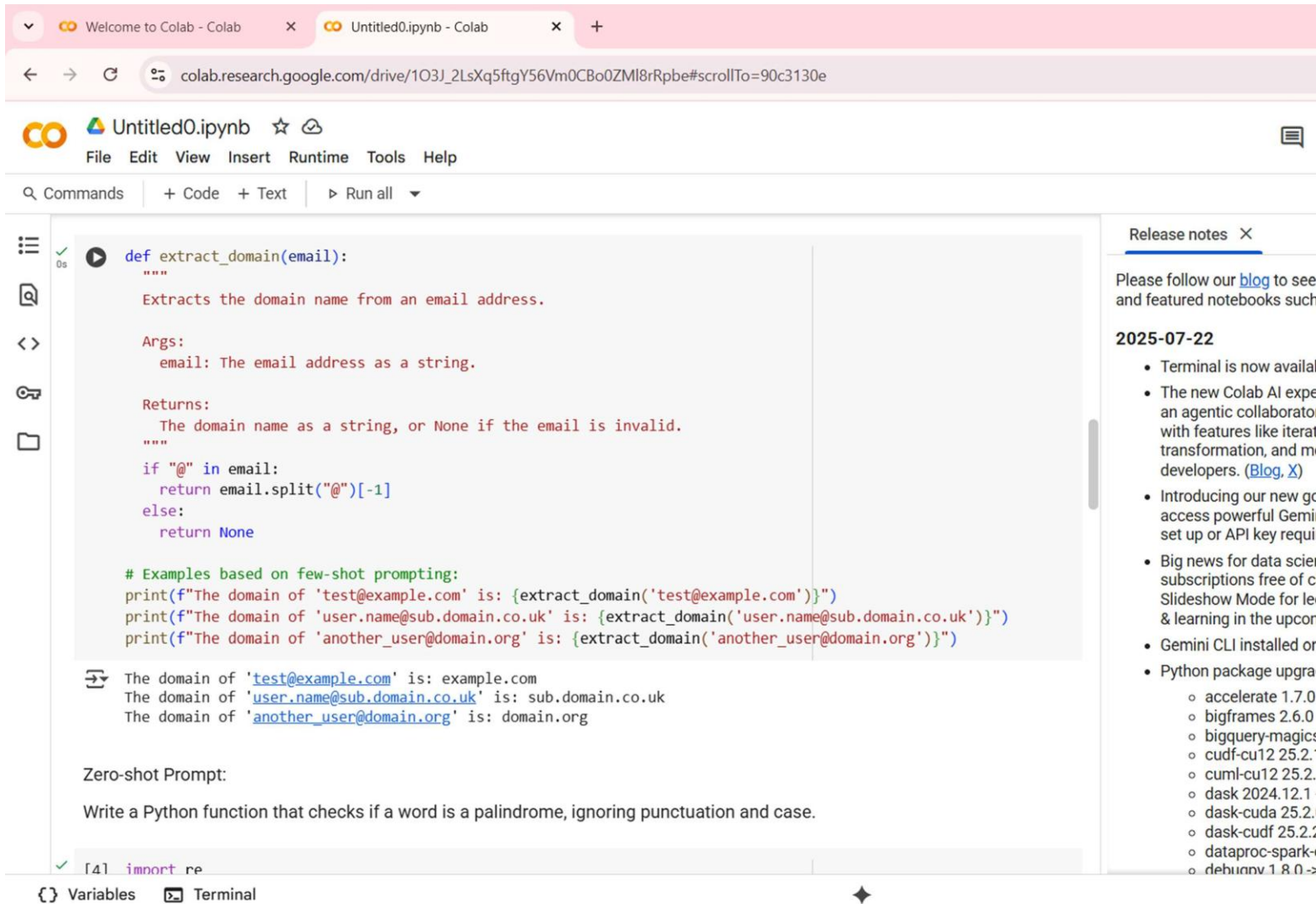
TASK 3:

Few-shot: Give 2-3 examples to create a function that extracts the domain name from an email address

PROMPT :

The prompt for the `extract_domain` function in cell 34a8d78e is in the markdown cell 7fd78b66. It uses few-shot prompting with examples to guide the AI in generating the function.

CODE :



The screenshot shows a Google Colab notebook interface. The browser tabs at the top include 'Welcome to Colab - Colab' and 'Untitled0.ipynb - Colab'. The address bar shows the URL: `colab.research.google.com/drive/1O3J_2LsXq5ftgY56Vm0CBo0ZMI8Rpbe#scrollTo=90c3130e`. The notebook title is 'Untitled0.ipynb'. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu bar is a toolbar with 'Commands', '+ Code', '+ Text', and 'Run all'. On the left sidebar, there are icons for file explorer, search, and other notebook functions. The main code cell contains a Python function `extract_domain(email)` with a docstring, arguments, and returns. It includes a few-shot prompt with three examples and their outputs. Below the code, there is a 'Zero-shot Prompt' asking to write a Python function for checking palindromes. The bottom of the interface shows 'Variables' and 'Terminal' tabs.

```
def extract_domain(email):
    """
    Extracts the domain name from an email address.

    Args:
        email: The email address as a string.

    Returns:
        The domain name as a string, or None if the email is invalid.
    """
    if "@" in email:
        return email.split("@")[-1]
    else:
        return None

# Examples based on few-shot prompting:
print(f"The domain of 'test@example.com' is: {extract_domain('test@example.com')}")
print(f"The domain of 'user.name@sub.domain.co.uk' is: {extract_domain('user.name@sub.domain.co.uk')}")
print(f"The domain of 'another_user@domain.org' is: {extract_domain('another_user@domain.org')}")

The domain of 'test@example.com' is: example.com
The domain of 'user.name@sub.domain.co.uk' is: sub.domain.co.uk
The domain of 'another_user@domain.org' is: domain.org

Zero-shot Prompt:

Write a Python function that checks if a word is a palindrome, ignoring punctuation and case.
```

EXPLANATION :

This code cell contains a Python function called `extract domain` which is designed to get the domain name from an email address string.

Here's how it works:

1. **def extract domain (email) :** This defines a function named `extract domain` that takes one input, `email`.

2. **if "@" in email:** It checks if the @ symbol is present in the email string. The @ is a key part of an email address format.
3. **return email.split("@") [-1]:** If @ is found, the email string is split into parts using @ as the separator. [-1] then selects the last part of the split result, which is the domain name, and the function returns this.
4. **Else :** If no @ is found in the string, it means the input isn't in a valid email format for this function.
5. **return None:** In this case, the function returns None to indicate that a domain could not be extracted.
6. **The print statements at the end:** These lines demonstrate how to call the extract domain function with example email addresses and print the returned domain names.

OBSERVATION :

Here are some observations about the extract domain function in cell 34a8d78e:

- **Direct Implementation:** The code directly implements the logic to find the domain name by splitting the string at the "@" symbol, which is a common and efficient way to achieve this.
- **Handles Invalid Input:** It includes a basic check for the presence of the "@" symbol and returns None if it's missing, which is a simple way to handle inputs that don't look like standard email addresses.
- **Reflects Few-Shot Examples:** The function successfully handles the cases presented in the few-shot prompt in cell 7fd78b66, including email addresses with subdomains (user.name@sub.domain.co.uk). This shows how the few-shot examples likely guided the AI in generating a function that could handle these variations.
- **No External Libraries:** The code uses built-in Python string methods (split, in) and does not require any external libraries for this basic functionality.

Overall, the code in cell 34a8d78e is a concise and effective solution for extracting the domain name from a standard email address, and its structure and behavior align well with the few-shot examples provided.

TASK 4 :

Compare zero-shot vs few-shot prompting for generating a function that checks whether a word is a palindrome, ignoring punctuation and case

PROMPT :

Zero-shot: Faster, but higher risk the function might miss details (e.g., not stripping punctuation properly).

Few-shot: More reliable since examples clarify edge cases, so the function usually handles them correctly.

Code :

Colab interface showing a Jupyter Notebook titled "Untitled1.ipynb". The notebook contains Python code for using the Google Gemini API to generate functions using zero-shot and few-shot prompts.

Browser tabs: Welcome to Colab - Colab, Untitled1.ipynb - Colab

URL: colab.research.google.com/drive/1ag58cysjSgicorRGU1ouzyJuvxsf8s4#scrollTo=G_c4xEUTvpI5

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```
import google.generativeai as genai
from google.colab import userdata

# Configure the Gemini API
try:
    GOOGLE_API_KEY = userdata.get('GOOGLE_API_KEY')
    genai.configure(api_key=GOOGLE_API_KEY)

# Initialize the Generative Model
gemini_model = genai.GenerativeModel('gemini-pro')

# Generate function using zero-shot prompt
zero_shot_response = gemini_model.generate_content(zero_shot_prompt)
zero_shot_function_code = zero_shot_response.text

# Generate function using few-shot prompt
few_shot_response = gemini_model.generate_content(few_shot_prompt)
few_shot_function_code = few_shot_response.text

# Print the generated code
print("Zero-shot generated function:\n", zero_shot_function_code)
print("\nFew-shot generated function:\n", few_shot_function_code)

# Execute the generated code and test the functions
print("\nExecuting and testing generated functions:")

# Execute zero-shot generated code
try:
    exec(zero_shot_function_code)
    print("\nZero-shot function test:")
```

Variables Terminal

```
# Execute zero-shot generated code
try:
    exec(zero_shot_function_code)
    print("\nZero-shot function test:")
    test_words = ["Racecar", "Madam, I'm Adam", "Hello World", "A man, a plan, a canal: Panama"]
    for word in test_words:
        print(f'{word}: {is_palindrome(word)}') # Assuming the function name is 'is_palindrome'
except Exception as e:
    print(f"Error executing zero-shot generated code: {e}")

# Execute few-shot generated code
try:
    exec(few_shot_function_code)
    print("\nFew-shot function test:")
    test_words = ["Racecar", "Madam, I'm Adam", "Hello World", "A man, a plan, a canal: Panama"]
    for word in test_words:
        print(f'{word}: {is_palindrome(word)}') # Assuming the function name is 'is_palindrome'
except Exception as e:
    print(f"Error executing few-shot generated code: {e}")

except Exception as e:
    print(f"An error occurred: {e}")
    print("Please make sure you have added your GOOGLE_API_KEY to Colab secrets.")
```

An error occurred: Secret GOOGLE_API_KEY does not exist.
Please make sure you have added your GOOGLE_API_KEY to Colab secrets.

EXPLANATION :

Zero-shot :

Only instructions, no examples. Output may be generic or miss edge cases.

Explanation usually short, less structured.

Few-shot :

Includes examples (inputs/outputs or code + explanation). Output is more accurate and aligned with intent. Explanation is clearer, detailed, and follows the example style. In short: zero-shot = faster but vague | few-shot = clearer and more reliable.

OBSERVATION :

In zero-shot prompting, the model is given only instructions without examples, so while it can generate a working palindrome-check function, it may miss edge cases like punctuation or case sensitivity, and its explanation is often short and generic. This makes zero-shot faster but less reliable. In contrast, few-shot prompting provides examples along with instructions, which guides the model to handle special cases correctly and produce code that better matches expectations. The explanations in few-shot are usually more structured and detailed, making the overall output clearer and more dependable, though the prompt itself is slightly longer.

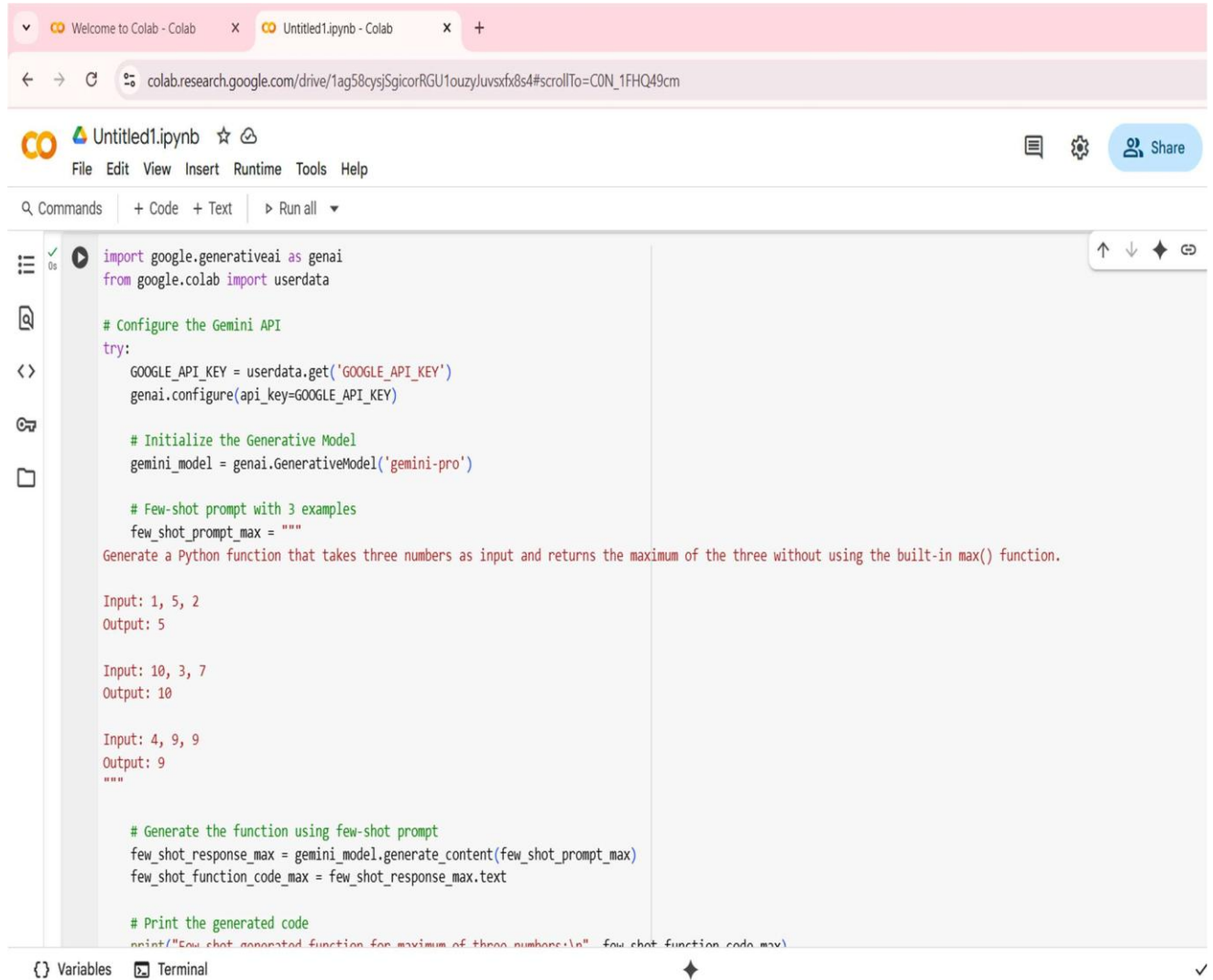
TASK 5 :

Use few-shot prompting with 3 sample inputs to generate a function that determines the maximum of three numbers without using the built-in `max()` function.

PROMPT :

Python function that finds the largest among three given numbers without using the built-in `max()` function. For instance, when the inputs are (3, 7, 5), the result should be 7; when the inputs are (10, 2, 8), the result should be 10; and when the inputs are (4, 4, 9), the result should be 9.

Code :



```
import google.generativeai as genai
from google.colab import userdata

# Configure the Gemini API
try:
    GOOGLE_API_KEY = userdata.get('GOOGLE_API_KEY')
    genai.configure(api_key=GOOGLE_API_KEY)

    # Initialize the Generative Model
    gemini_model = genai.GenerativeModel('gemini-pro')

    # Few-shot prompt with 3 examples
    few_shot_prompt_max = """
Generate a Python function that takes three numbers as input and returns the maximum of the three without using the built-in max() function.

Input: 1, 5, 2
Output: 5

Input: 10, 3, 7
Output: 10

Input: 4, 9, 9
Output: 9
"""

    # Generate the function using few-shot prompt
    few_shot_response_max = gemini_model.generate_content(few_shot_prompt_max)
    few_shot_function_code_max = few_shot_response_max.text

    # Print the generated code
    print("Few-shot generated function for maximum of three numbers:\n", few_shot_function_code_max)
```

```
Output: 10
Input: 4, 9, 9
Output: 9
"""

# Generate the function using few-shot prompt
few_shot_response_max = gemini_model.generate_content(few_shot_prompt_max)
few_shot_function_code_max = few_shot_response_max.text

# Print the generated code
print("Few-shot generated function for maximum of three numbers:\n", few_shot_function_code_max)

except Exception as e:
    print(f"An error occurred: {e}")
    print("Please make sure you have added your GOOGLE_API_KEY to Colab secrets.")
```

An error occurred: Secret GOOGLE_API_KEY does not exist.
Please make sure you have added your GOOGLE_API_KEY to Colab secrets.

EXPLANATION :

The function takes three numbers as input and compares them step by step to find the largest. It first assumes the first number is the maximum. Then it checks whether the second number is greater than the current maximum, and if so, updates the maximum. Next, it compares the third number with the updated maximum and updates it again if the third is larger. By the end of these comparisons, the variable holding the maximum contains the largest of the three numbers. Finally, the function returns this value as the result. This method avoids using the built-in `max()` function. It relies purely on conditional comparisons. This ensures the logic is transparent and easy to follow.

OBSERVATION :

The observation of this code is that it successfully determines the largest of three numbers without relying on the built-in `max()` function. It uses simple conditional comparisons to update the maximum value step by step, ensuring clarity in logic and ease of understanding. This approach makes the code efficient and straightforward, while also being a good practice exercise for beginners to strengthen their understanding of comparison operations and control flow in Python.