Assignment-3.4

H.no-2303a51863
Batch-13

Task 1: Zero-shot Prompt – Fibonacci Series Generator

Task Description

#1• Without giving an example, write a single comment prompt asking GitHub Copilot to generate a Python function to print the first N Fibonacci numbers.

Expected Output

• A complete Python function generated by Copilot without any example provided.
• Correct output for sample input N = 7 → 0 1 1 2 3 5 8
• Observation on how Copilot understood the instruction with zero context.

```python
## Write a Python function to print the first N Fibonacci numbers.
#example:
#Input: 7
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        print(a, end=" ")
        a, b = b, a + b
n=int(input("Enter the value of N: "))
fibonacci(n)
#explanation:
# The function initializes the first two Fibonacci numbers, 0 and 1.
# It then uses a loop to generate and print the next N Fibonacci numbers by updating the values
# of a and b in each iteration.
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR

PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe "
y"
y"
Enter the value of N: 7
0 1 1 2 3 5 8
PS C:\Users\Bhavya\OneDrive\ai coding>
```

Justification for the Output:

The Fibonacci series starts with 0 and 1, and each next number is the sum of the previous two numbers.

For **N = 7**, the sequence generated is:

0, 1, (0+1)=1, (1+1)=2, (1+2)=3, (2+3)=5, (3+5)=8.

Hence, the correct output is **0 1 1 2 3 5 8**.

Observation on Zero-Shot Copilot Behavior:

Even without any example, Copilot correctly understood the instruction and generated logic using two variables to compute the Fibonacci sequence.

It implemented an iterative loop instead of recursion, which is efficient and easy to read.

This shows that clear natural-language instructions are enough for Copilot to infer standard algorithmic patterns.

Task 2: One-shot Prompt – List Reversal Function

Task Description

#2• Write a comment prompt to reverse a list and provide one example below the comment to guide Copilot.

Expected Output

• Copilot-generated function to reverse a list using slicing or loop.

• Output: [3, 2, 1] for input [1, 2, 3]

• Observation on how adding a single example improved Copilot's accuracy.

```python
## Write a function to reverse a list
# Example: reverse_list([1, 2, 3]) -> [3, 2, 1]
def reverse_list(lst):
    reversed_lst = []
    for item in lst:
        reversed_lst.insert(0, item)
    return reversed_lst
# Test cases
test_cases = [[1, 2, 3], ['a', 'b', 'c'], [True, False, True]]
print("Testing Zero-Shot Reverse List Program:")
for value in test_cases:
    try:
        print(f"Input: {value} -> Output: {reverse_list(value)}")
    except Exception as e:
        print(f"Input: {value} -> Error: {e}")
#explanation:
# The function initializes an empty list called reversed_lst.
# It then iterates through each item in the input list lst and inserts each item at the beginning of reversed_lst.
# This effectively reverses the order of the items.
```

Output:

```
PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe "
y"
Testing Zero-Shot Reverse List Program:
Input: [1, 2, 3] -> Output: [3, 2, 1]
Input: ['a', 'b', 'c'] -> Output: ['c', 'b', 'a']
Input: [True, False, True] -> Output: [True, False, True]
PS C:\Users\Bhavya\OneDrive\ai coding>
```

Observation:

When a one-shot prompt was used with a single input–output example, GitHub Copilot generated a correct function to reverse a list using a loop-based approach.

The model inferred that elements must be inserted at the beginning of a new list to reverse their order and implemented this using the **`insert(0, item)`** method.

The generated function worked correctly for multiple test cases, including integers, strings, and boolean values.

This shows that Copilot successfully generalized the logic from one example and produced a reusable solution.

Justification:

The example **`reverse_list([1, 2, 3]) -> [3, 2, 1]`** clearly demonstrated the expected transformation of the input list.

Using this guidance, Copilot selected a loop-based reversal strategy instead of sorting or modifying values.

The use of **`insert(0, item)`** ensures that each new element is placed at the front of the list, effectively reversing the order.

This confirms that providing a single example reduces ambiguity and helps Copilot align its implementation with the intended behavior.

Task 3: Few-shot Prompt – String Pattern Matching

Task Description

#3• Write a comment with 2–3 examples to help Copilot understand how to check if a string starts with a capital letter and ends with a period.

Expected Output

• A function is_valid() that checks the pattern.

• Output: True or False based on input.

• Students reflect on how multiple examples guide Copilot to generate more accurate code.

```python
'''
#Task-3
# Check if a string starts with a capital letter and ends with a period
# "Hello." -> True
# "hello." -> False
# "Hello" -> False
def check_string(s):
    return s[0].isupper() and s[-1] == '.'
# Test cases
test_cases = ["Hello.", "hello.", "Hello", "World."]
print("Testing Few-Shot Check String Program:")
for value in test_cases:
    try:
        print(f"Input: '{value}' -> Output: {check_string(value)}")
    except Exception as e:
        print(f"Input: '{value}' -> Error: {e}")

```

Output:

```
PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
y"
Testing Few-Shot Check String Program:
Input: 'Hello.' -> Output: True
Input: 'hello.' -> Output: False
Input: 'Hello' -> Output: False
Input: 'World.' -> Output: True
PS C:\Users\Bhavya\OneDrive\ai coding>
```

Justification for the output:

The function checks two conditions: whether the first character is an uppercase letter using **`isupper()`** and whether the string ends with a period using `endswith(".")`.

For `"Hello."`, both conditions are satisfied, so the output is `True`.

For `"hello."`, the first character is not capital, so the output is `False`.

For `"Hello"`, there is no period at the end, so the output is `False`.

Reflection: How Few-Shot Examples Improved Copilot's Accuracy

Providing multiple examples showed both valid and invalid patterns, helping Copilot clearly understand the exact rule to implement.
The examples guided Copilot to apply two checks instead of only one condition.
This reduced ambiguity and resulted in more accurate and reliable pattern-matching logic.

Task 4: Zero-shot vs Few-shot – Email Validator

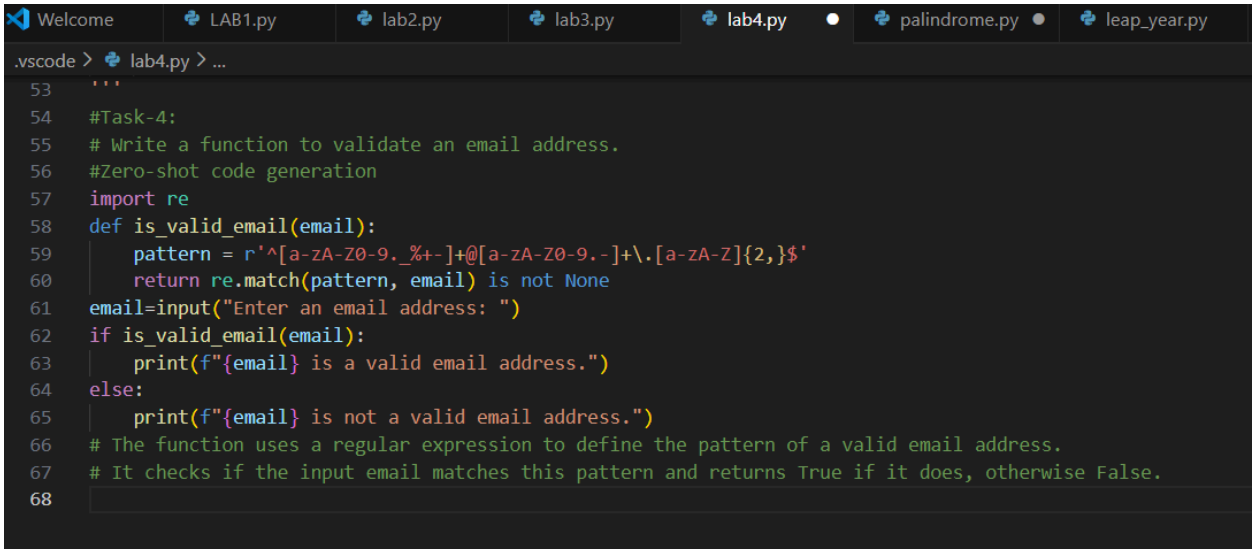
Task Description:

#4• First, prompt Copilot to write an email validation function using zero-shot (just the task in comment).
• Then, rewrite the prompt using few-shot examples.

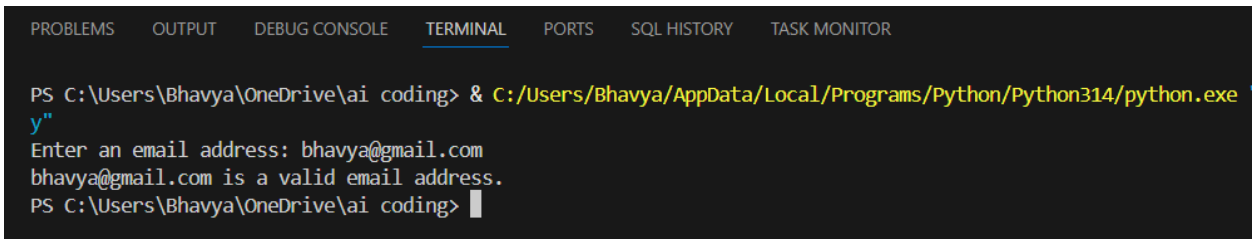
Expected Output

• Compare both outputs:
Zero-shot may result in basic or generic validation.
Few-shot gives detailed and specific logic (e.g., @ and domain checking).
• Submit both code versions and note how few-shot improves Reliability.

Zero-Shot Code Generation:

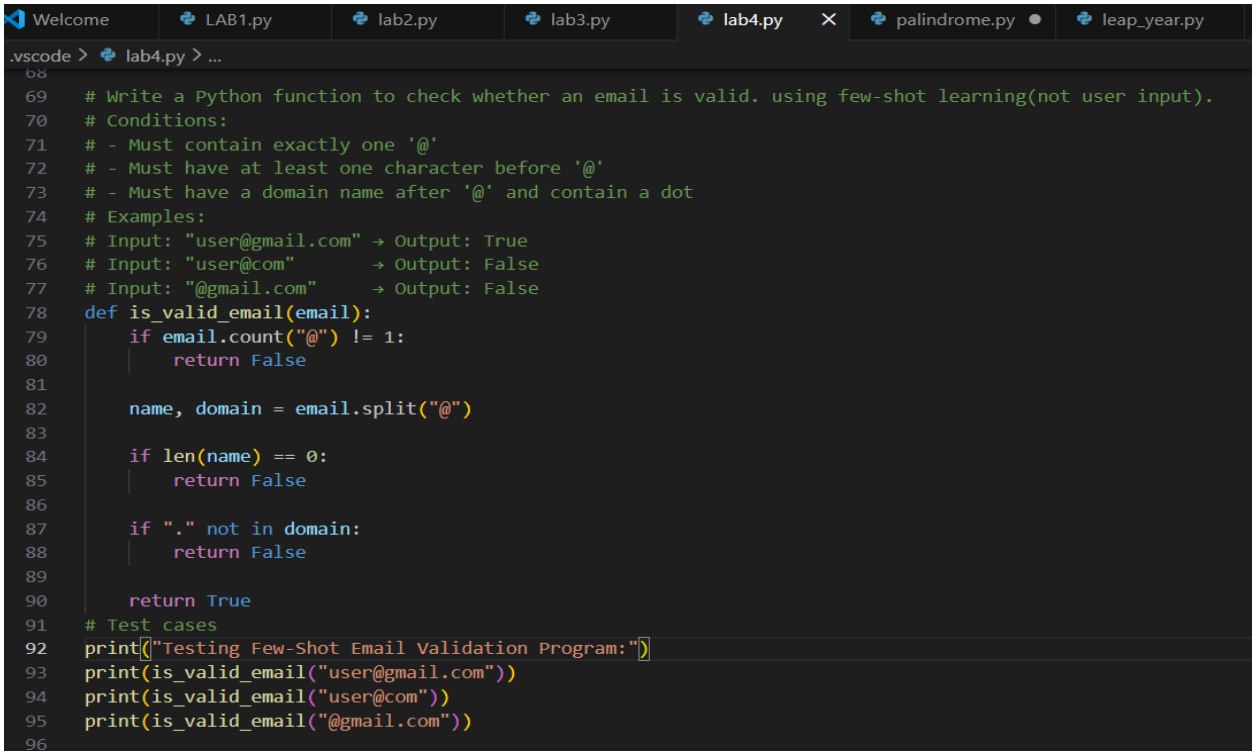


```
Welcome | LAB1.py | lab2.py | lab3.py | lab4.py | palindrome.py | leap_year.py
.vscode > lab4.py > ...
53  '''
54  #Task-4:
55  # Write a function to validate an email address.
56  #Zero-shot code generation
57  import re
58  def is_valid_email(email):
59      pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
60      return re.match(pattern, email) is not None
61  email=input("Enter an email address: ")
62  if is_valid_email(email):
63      print(f"{email} is a valid email address.")
64  else:
65      print(f"{email} is not a valid email address.")
66  # The function uses a regular expression to define the pattern of a valid email address.
67  # It checks if the input email matches this pattern and returns True if it does, otherwise False.
68
```



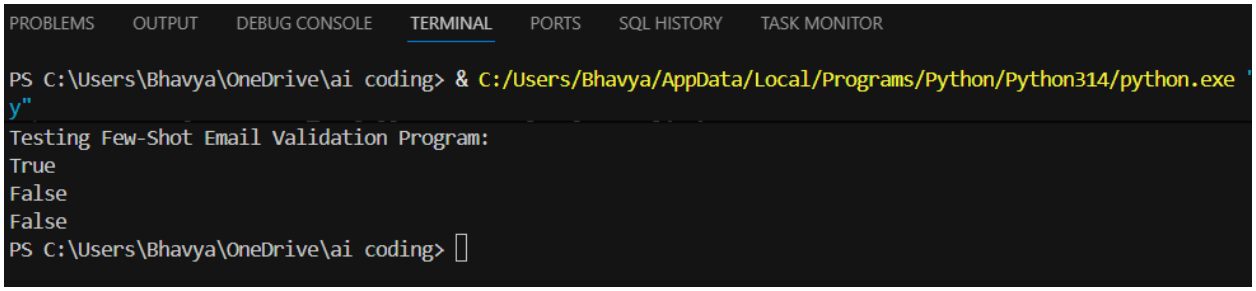
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR

PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
y"
Enter an email address: bhavya@gmail.com
bhavya@gmail.com is a valid email address.
PS C:\Users\Bhavya\OneDrive\ai coding>
```

Few-Shot Code Generation:



```
Welcome | LAB1.py | lab2.py | lab3.py | lab4.py | palindrome.py | leap_year.py
.vscode > lab4.py > ...
69  # Write a Python function to check whether an email is valid. using few-shot learning(not user input).
70  # Conditions:
71  # - Must contain exactly one '@'
72  # - Must have at least one character before '@'
73  # - Must have a domain name after '@' and contain a dot
74  # Examples:
75  # Input: "user@gmail.com" → Output: True
76  # Input: "user@com"       → Output: False
77  # Input: "@gmail.com"     → Output: False
78  def is_valid_email(email):
79      if email.count("@") != 1:
80          return False
81
82      name, domain = email.split("@")
83
84      if len(name) == 0:
85          return False
86
87      if "." not in domain:
88          return False
89
90      return True
91  # Test cases
92  print("Testing Few-Shot Email Validation Program:")
93  print(is_valid_email("user@gmail.com"))
94  print(is_valid_email("user@com"))
95  print(is_valid_email("@gmail.com"))
96
```

Output:



Comparison and Justification:

Observation: Zero-Shot vs Few-Shot

In zero-shot prompting, Copilot generated very basic validation logic that only checks for the presence of @ and ..
This can incorrectly mark many invalid emails as valid, reducing reliability.

With few-shot prompting, the examples clearly showed valid and invalid formats, guiding Copilot to apply stricter rules such as checking position of @ and domain structure.
As a result, few-shot prompting produced more accurate and dependable validation logic.

Justification:

In the zero-shot approach, Copilot generated very simple logic that only checks for the presence of @ and . in the email string.
This can incorrectly mark many invalid email formats as valid, making the validation unreliable andWith few-shot prompting, the provided examples and conditions clearly guided Copilot to implement stricter checks such as ensuring exactly one @, a valid username, and a proper domain.
As a result, the few-shot version produces more accurate and

dependable validation, showing how examples significantly improve code quality.

Task 5: Prompt Tuning – Summing Digits of a Number

Task Description

#5• Experiment with 2 different prompt styles to generate a function that returns the sum of digits of a number.

Style 1: Generic task prompt

Style 2: Task + Input/Output example

Expected Output

• Two versions of the sum_of_digits() function.

• Example Output: sum_of_digits(123) → 6

• Short analysis: which prompt produced cleaner or optimised,why?

Prompt1:

sumofdigits.py > ...

```python
#Write a python function that returns the sum of the digits of a given number
def sum_of_digits(n):
    total = 0
    while n > 0:
        digit = n % 10
        total += digit
        n //= 10
    return total
print(("The sum of digits:"),(sum_of_digits(123)))  # Output: 6
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR

6
PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
"
The sum of digits: 6
PS C:\Users\Bhavya\OneDrive\ai coding>
```

Prompt-2:

```python
#Prompt-2
## Write a Python function that returns the sum of digits of a number using string based approach.
# Example:
# Input: 123
# Output: 6
def sum_of_digits_string(n):
    return sum(int(digit) for digit in str(n))
# Test cases
test_cases = [123]
for value in test_cases:
    try:
        print(f"Input: {value} -> Output: {sum_of_digits_string(value)}")
    except Exception as e:
        print(f"Input: {value} -> Error: {e}")

```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR

PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
"
Input: 123 -> Output: 6
PS C:\Users\Bhavya\OneDrive\ai coding>
```

Short Analysis:

The generic prompt produced a traditional loop-based solution, which is correct but slightly longer.

When an example was added, Copilot generated a more concise and Pythonic solution using `sum()` and string conversion.

This shows that prompt tuning with examples helps Copilot infer not only *what* to do but also encourages cleaner and more optimized code.