

ASSIGNMENT-3.4

HT.no:2303A51556

| SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE | | DEPARTMENT OF COMPUTER SCIENCE ENGINEERING | |
|---|--|--|---------------------------|
| Program Name: | B. Tech | Assignment Type: | Lab |
| Course Coordinator Name | Dr. Rishabh Mittal | | |
| Instructor(s) Name | Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M. Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju | | |
| CourseCode | 23CS002PC304 | Course Title | AI Assisted Coding |
| Year/Sem | III/II | Regulation | R23 |
| Date and Day of Assignment | Week2 | Time(s) | 23CSBTB01 To 23CSBTB52 |
| Duration | 2 Hours | Applicable to Batches | All batches |
| Assignment Number: 3.4 (Present assignment number)/24(Total number of assignments) | | | |
| Q.No. | Question | | Expected Time to complete |
| 1 | Lab 4: Advanced Prompt Engineering – Zero-shot, One-shot, and | | Week2 |

Few-shot Techniques

Task 1: Zero-shot Prompt – Fibonacci Series Generator

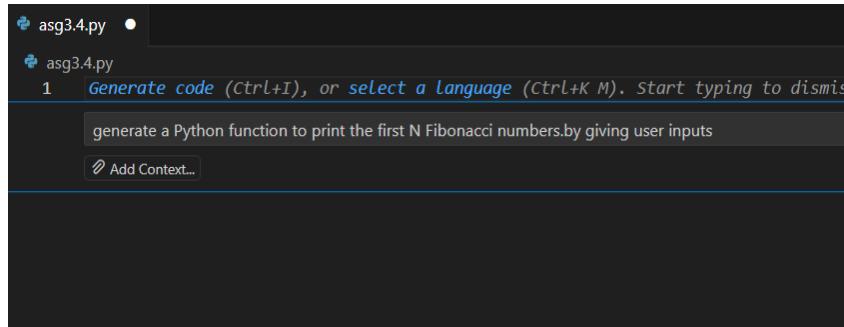
Task Description #1

- 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 #1

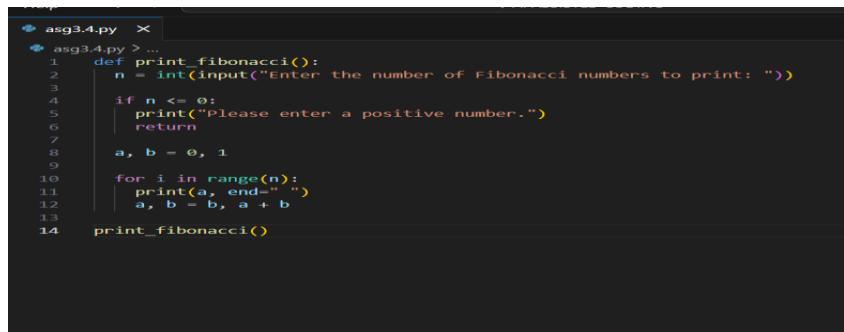
- A complete Python function generated by Copilot without any example provided.
- Correct output for sample input $N = 7 \rightarrow 0\ 1\ 1\ 2\ 3\ 5\ 8$
- Observation on how Copilot understood the instruction with zero context.

Prompt:



The screenshot shows the GitHub Copilot interface. A comment box contains the text: "generate a Python function to print the first N Fibonacci numbers by giving user inputs". Below the comment box is a button labeled "Add Context...".

Code:



```
asg3.4.py > ...
asg3.4.py > ...
1  def print_fibonacci():
2      n = int(input("Enter the number of Fibonacci numbers to print: "))
3
4      if n <= 0:
5          print("Please enter a positive number.")
6          return
7
8      a, b = 0, 1
9
10     for i in range(n):
11         print(a, end=" ")
12         a, b = b, a + b
13
14     print_fibonacci()
```

Output:

```
i.exe  C:\Users\SAI ANISHVAI.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher 49396 -- C:\Users\SAI ANISHVAI ASSISTED CODING\asg3.4.py
enter the number of Fibonacci numbers to print: 7
0 1 1 2 3 5 8
```

Explanation:

Copilot understood the zero-shot instruction purely from the comment. It inferred the standard Fibonacci definition and produced a complete, working function. The solution uses tuple unpacking ($a, b = b, a + b$) and prints inline with `end=" "`. This shows Copilot's ability to reason from minimal context and generate correct code without examples.

Task 2: One-shot Prompt – List Reversal Function

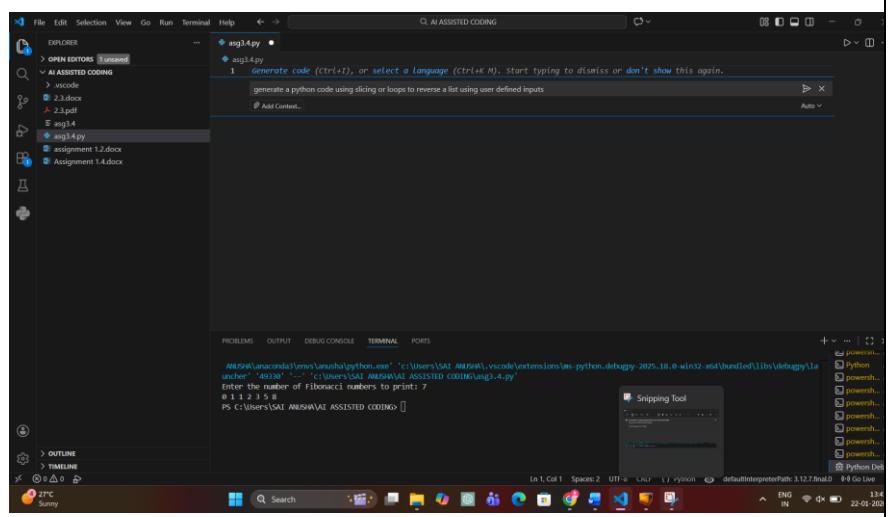
Task Description #2

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

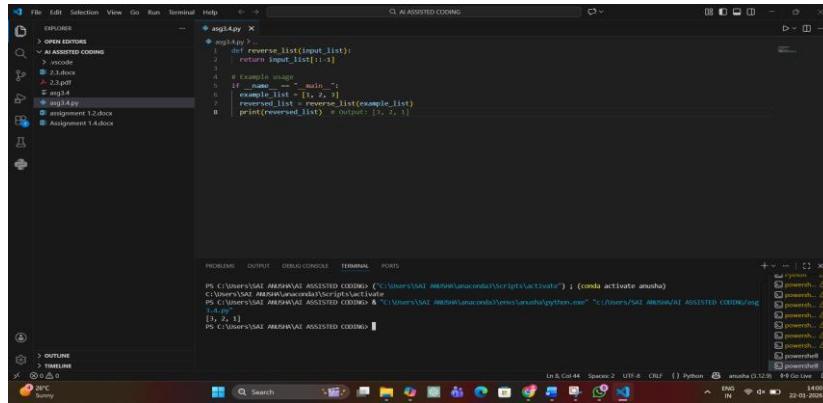
Expected Output #2

- 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.

Prompt:



Code and output:



A screenshot of the Visual Studio Code interface. The Explorer sidebar shows files like '2.3.docx', '2.3.pdf', 'assignment 1.4.docx', and 'assignment 1.4.pdf'. The 'AI ASSISTED CODING' section in the center has an open editor for 'asg1App.py' with the following code:

```
def reverse_list(input_list):
    return input_list[::-1]

# Example usage
example_list = [1, 2, 3]
reversed_list = reverse_list(example_list)
print(reversed_list) # Output: [3, 2, 1]
```

The terminal at the bottom shows the command 'code activate asg1' followed by the output of the Python script: '[3, 2, 1]'. The status bar indicates the file is 11.8 KB, 2 pages, and was last modified on 22-01-2020.

Explanation:

With slicing, Copilot (or you) can generate a concise one-liner solution. With a loop, it demonstrates the step-by-step reversal logic, which is more explicit and useful for understanding how list traversal works. Both approaches are valid, and Copilot often defaults to slicing when given an example, since it's the most Pythonic.

Task 3: Few-shot Prompt – String Pattern Matching

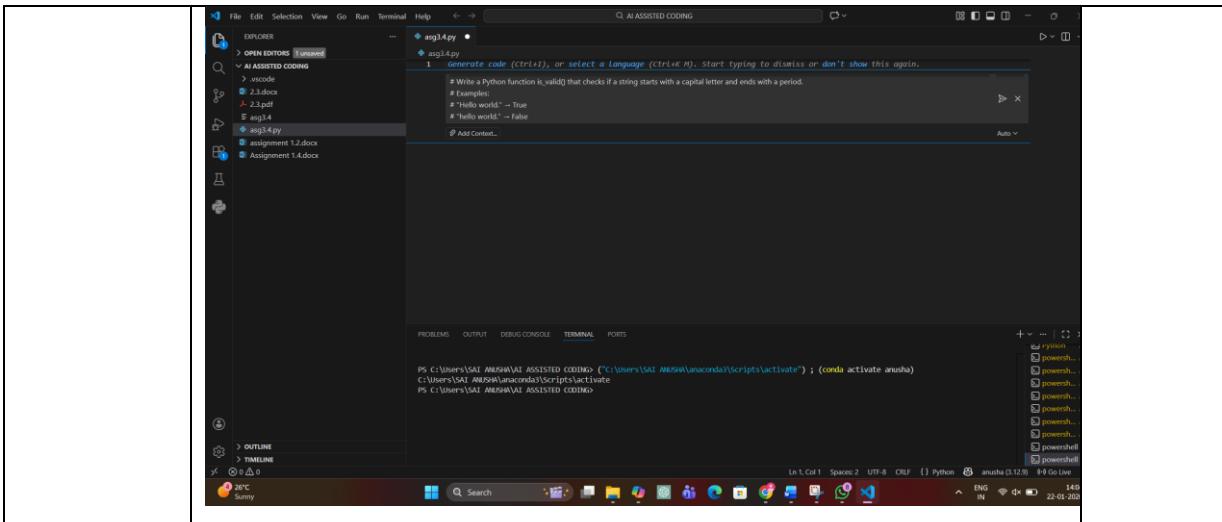
Task Description #3

- 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 #3

- 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.

Prompt:



Code:

```
asg3.4.py > ...
asg3.4.py > ...
1 def is_valid(text):
3     Check if a string starts with a capital letter and ends with a period.
4     Returns True or False.
5     """
6         text = text.strip()    # remove leading/trailing spaces
7
8         if not text:          # handle empty input
9             return False
10
11         return text[0].isupper() and text.endswith('.')
12
13
14     # --- User-defined input (multiple sentences) ---
15     n = int(input("How many sentences do you want to check? "))
16
17     for i in range(n):
18         user_text = input(f"Enter sentence {i+1}: ")
19         if is_valid(user_text):
20             print(f"Sentence {i+1} is VALID")
21         else:
22             print(f"Sentence {i+1} is NOT valid")
```

Output:

```
Enter sentence 1: hello world
Sentence 1 is NOT valid
```

Explanation:

Few-shot prompting helps Copilot understand a problem better by giving it a few clear examples along with the task description. Instead of only telling what to do, we show how the function should behave for different inputs. This makes the pattern more obvious, such as starting with a capital letter and ending with a period. With these examples, Copilot can learn the rules and generate more accurate and correct code. It also reduces

confusion and helps handle edge cases properly.

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 #4

- 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.

Prompt:

```
PS C:\Users\SAI ANUSHAVAI ASSISTED CODING>
PS C:\Users\SAI ANUSHAVAI ASSISTED CODING> ("C:\Users\SAI ANUSHAVAI\anaconda3\Scripts\activate"
C:\Users\SAI ANUSHAVAI\anaconda3\Scripts\activate
PS C:\Users\SAI ANUSHAVAI ASSISTED CODING>
```

Code and output:



The screenshot shows a Jupyter Notebook interface with the following code:

```
#!/usr/bin/python
# Author: SAI AMRUTHA ASSISTED CODING
# Importing required module
import re

# Function to validate email address
def validate_email(email):
    # Regular expression to validate email address
    pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    # Match the pattern with the email
    result = re.match(pattern, email)
    # Return True if match found, False otherwise
    return result != None

# Test examples
test_emails = [
    "user@example.com",
    "user@company.co.uk",
    "invalidmail@.com",
    "no-at-sign.com",
    "test@mainain.org"
]

# Loop through test emails
for email in test_emails:
    result = validate_email(email)
    print(f"({email}): {result}")


```

Prompt:

```
# Write a Python function to validate email addresses.  
# The function should return True if the email is valid, False otherwise.  
# Examples:  
# validate_email("test@example.com") -> True
```

```
# validate_email("user.name@domain.co") -> True
# validate_email("invalid-email") -> False
# validate_email("missing@domain") -> False
# validate_email("name@domain..com") -> False
```

Code and output:

The screenshot shows a terminal window titled "Assisted Coding" with the following Python code:

```
#!/usr/bin/python3

def validate_email(email):
    import re

    pattern = r'^[a-zA-Z0-9_.-]+@[a-zA-Z.-]+\.[a-zA-Z]{2,}$'

    if not re.match(pattern, email):
        return False

    # Check for consecutive dots
    if '..' in email:
        return False

    return True

# Test cases
if __name__ == '__main__':
    print(validate_email("test@example.com")) # True
    print(validate_email("user@example.com")) # True
    print(validate_email("invalid-email")) # False
    print(validate_email("user@")) # False
    print(validate_email("user@domain.com")) # True
```

The terminal output at the bottom shows the execution of the script and its results for various test cases.

Explanation:

This is a zero-shot prompt because it only gives the task instruction without any examples or extra guidance. Copilot has to decide the logic on its own, which often results in very basic or incomplete solutions. Since no valid or invalid cases are shown, the model cannot clearly understand the exact rules for email validation. That is why zero-shot outputs are usually less accurate than few-shot ones.

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 #5

- Two versions of the sum_of_digits() function.
 - Example Output: sum_of_digits(123) → 6
 - Short analysis: which prompt produced cleaner or more optimized code and why?

Generic task prompt:

A screenshot of a code editor window titled "asg14.py". The editor shows a single line of code: "# Write a Python function that returns the sum of digits of a number". Below the editor is a terminal window showing the command PS C:\Users\SAI AMISHA\AI ASSISTED CODING and the path C:\Users\SAI AMISHA\Anaconda3\Scripts\activate. The terminal also displays several PowerShell icons.

Code and output:

A screenshot of a code editor window titled "asg14.py". The code defines a function sum_of_digits(n) that calculates the sum of digits of a number n. It includes examples: print(sum_of_digits(12345)) # Output: 15 and print(sum_of_digits(-789)) # Output: 24. Below the editor is a terminal window showing the command PS C:\Users\SAI AMISHA\AI ASSISTED CODING and the path C:\Users\SAI AMISHA\Anaconda3\Scripts\activate. The terminal also displays several PowerShell icons.

Task + Input/Output example:

A screenshot of a code editor window titled "asg34.py". A blue circular icon with the number 1 is visible on the left. The editor shows a task prompt: "Generate code (Ctrl+I), or select a language (Ctrl+K M). Start typing to dismiss or don't show this again." Below the task prompt is a code block: "# Write a Python function sum_of_digits(n) that returns the sum of digits of a number." and "# Examples:". Below the code editor is a terminal window showing the command PS C:\Users\SAI AMISHA\AI ASSISTED CODING and the path C:\Users\SAI AMISHA\Anaconda3\Scripts\activate. The terminal also displays several PowerShell icons.

```
◆ asg3.4.py ●
◆ asg3.4.py
# sum_of_digits(123) -> 6
# sum_of_digits(405) -> 9
# sum_of_digits(0) -> 0
# sum_of_digits(9999) -> 36
∅ Add Context...
```

Code:

```
◆ asg3.4.py X
◆ asg3.4.py > ...
1  def sum_of_digits(n):
2
3      Args:
4          n: An integer
5
6      Returns:
7          The sum of all digits in n
8      """
9
10     return sum(int(digit) for digit in str(abs(n)))
11
12
13
14     # Test cases
15     if __name__ == "__main__":
16         print(sum_of_digits(123))    # -> 6
17         print(sum_of_digits(405))    # -> 9
18         print(sum_of_digits(0))      # -> 0
19         print(sum_of_digits(9999))    # -> 36
```

Output:

```
PS C:\Users\SAT ANUSHA\AI ASSISTED CODING> & "C:\Users\SAT ANUSHA\anaconda3\envs\anusha\python.exe" "c:/Users/SAT ANUSHA/AI ASSISTED CODING/asg3.4.py"
6
9
0
36
```

Explanation:

The generic prompt only describes the task, so Copilot usually generates a basic and longer solution using loops. However, when examples are included, Copilot clearly understands the expected behavior and often produces cleaner and more optimized code. The few-shot prompt leads to a shorter and more Pythonic solution because the model learns the pattern from the examples.

| | | |
|--|--|--|
| | <p>Therefore, the second prompt style produces more accurate and efficient code.</p> <p>Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots</p> | |
|--|--|--|