

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	Academic Year: 2025-2026
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

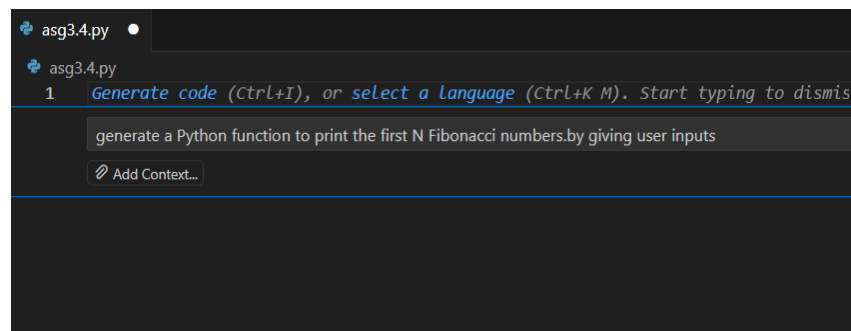
Task Description #1

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

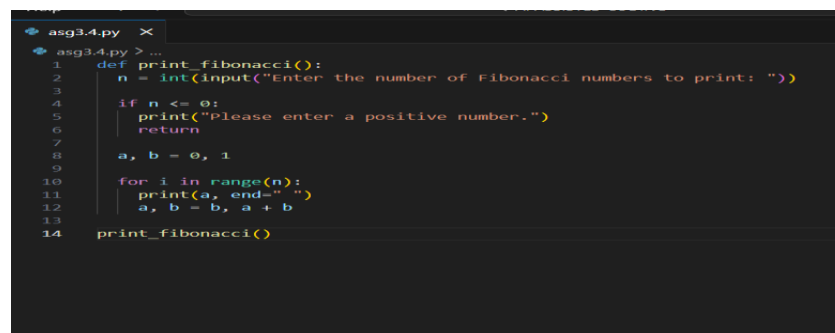
Expected Output #1

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

Prompt:



Code:



Output:

```
File Edit Selection View Go Run Terminal Help
C:\Users\SAT\Anaconda\extensions\ms-python.debugpy-2025.18.0-win32-x64\out\lib\debugpy\launcher 40938
ANISHAJAI 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

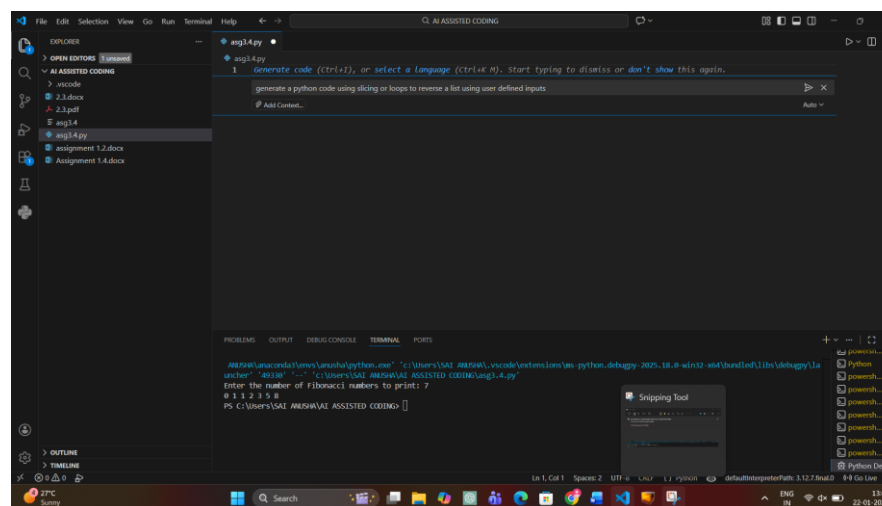
Task Description #2

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

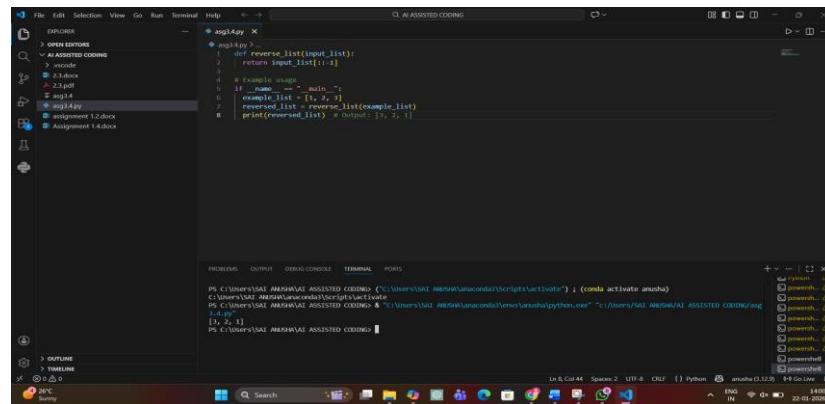
Expected Output #2

- 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:



```
def reverse_list(input_list):  
    return input_list[::-1]  
  
# Example usage  
if __name__ == "__main__":  
    example_list = [1, 2, 3]  
    reversed_list = reverse_list(example_list)  
    print(reversed_list) # Output: [3, 2, 1]
```

```
PS C:\Users\SAG\ANISHVAI ASSISTED CODING> (conda activate anusha)  
C:\Users\SAG\ANISHVAI ASSISTED CODING> python  
PS C:\Users\SAG\ANISHVAI ASSISTED CODING> C:\Users\SAG\ANISHVAI ASSISTED CODING> sing  
1-4.py  
[3, 2, 1]
```

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

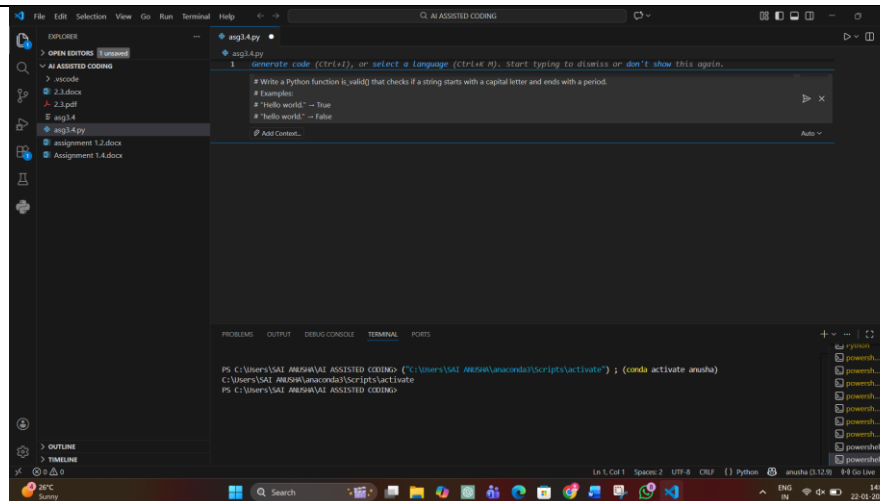
Task Description #3

- 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:

```

1 def is_valid(text):
2     """
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

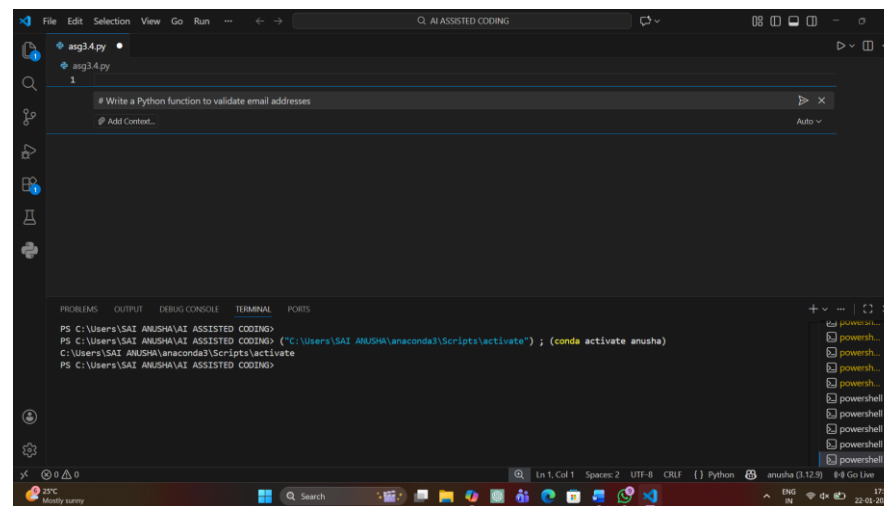
- Compare both outputs:

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:



```
# Write a Python function to validate email addresses

PS C:\Users\SAI ANUSHA\AI ASSISTED CODING>
PS C:\Users\SAI ANUSHA\AI ASSISTED CODING> ("C:\Users\SAI ANUSHA\anaconda3\Scripts\activate"); (conda activate anusha)
C:\Users\SAI ANUSHA\anaconda3\Scripts\activate
PS C:\Users\SAI ANUSHA\AI ASSISTED CODING>
```

Code and output:

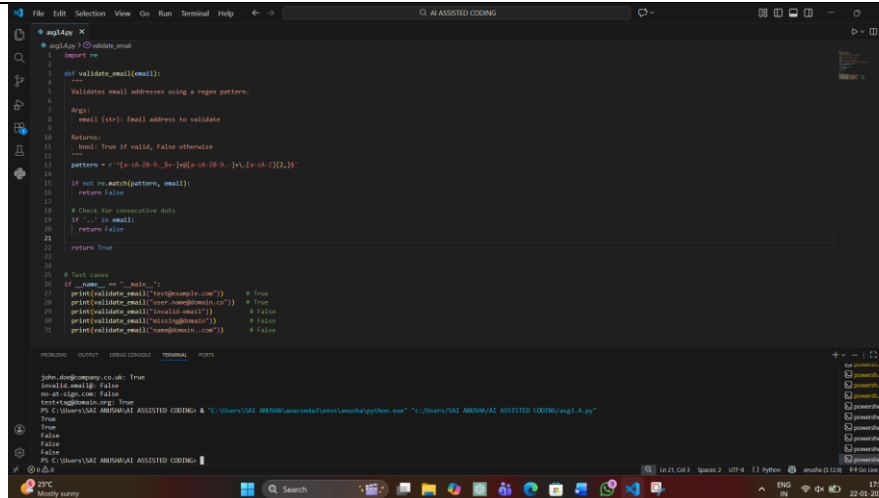
```
File Edit Selection View Go Run Terminal Help AI ASSISTED CODING
asg3.4.py X
1 import re
2
3 def validate_email(email):
4     """
5     Validator on email address using regex pattern.
6
7     Args:
8     email (str): The email address to validate
9
10    Returns:
11    bool: True if valid, False otherwise
12    """
13    pattern = r"^[a-z0-9]+([a-z0-9-]+)?@[a-z0-9-]+(\.[a-z]{2,4})$"
14    return re.match(pattern, email) is not None
15
16 # Test examples
17 if __name__ == "__main__":
18     test_emails = [
19         "example.com",
20         "john.doe@example.co.uk",
21         "invalid_email",
22         "too.at.sign.co.uk",
23         "testtag@domain.org"
24     ]
25
26 for email in test_emails:
27     result = validate_email(email)
28     print(f"{email}: {result}")
29
PROBLEMS OUTPUT OUTPUT CONSOLE TERMINAL PORTS
PS C:\Users\SAL\ANNOUAI\ASSISTED CODING> (conda activate amoha)
C:\Users\SAL\ANNOUAI\ASSISTED CODING> cd C:\Users\SAL\ANNOUAI\ASSISTED CODING\asg3.4.py
PS C:\Users\SAL\ANNOUAI\ASSISTED CODING> python asg3.4.py
example.com: True
john.doe@example.co.uk: True
invalid_email: False
too.at.sign.co.uk: False
testtag@domain.org: True
PS C:\Users\SAL\ANNOUAI\ASSISTED CODING>
```

Prompt:

```
asg3.4.py
asg3.4.py
1
# 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
Add Context...

asg3.4.py
asg3.4.py
1
# validate_email("user.name@domain.co") -> True
# validate_email("invalid-email") -> False
# validate_email("missing@domain") -> False
# validate_email("name@domain.com") -> False
Add Context...
```

Code and output:



```
1 # validate_email.py
2 import re
3
4 def validate_email(email):
5     """
6     Validates email addresses using a regex pattern.
7
8     Args:
9         email (str): Email address to validate
10
11     Returns:
12         bool: True if valid, False otherwise
13     """
14     pattern = r'^[a-zA-Z0-9_!@#$%^&*~.-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
15     if not re.match(pattern, email):
16         return False
17     # Check for common mistakes
18     if '..' in email:
19         return False
20     return True
21
22 # Test cases
23 if __name__ == '__main__':
24     print(validate_email('test@example.com')) # True
25     print(validate_email('user@domain.co.uk')) # True
26     print(validate_email('invalid_email')) # False
27     print(validate_email('invalid@domain')) # False
28     print(validate_email('user@domain.com')) # True
```

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.

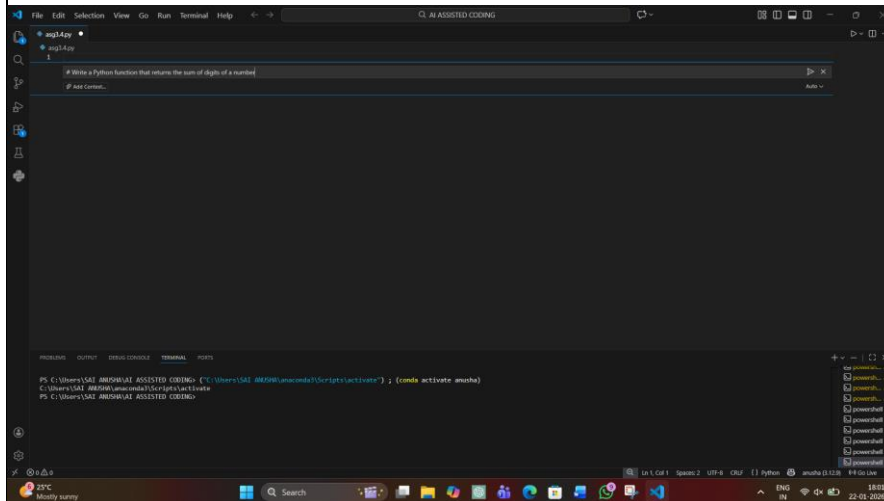
Style 1: Generic task prompt

Style 2: Task + Input/Output example

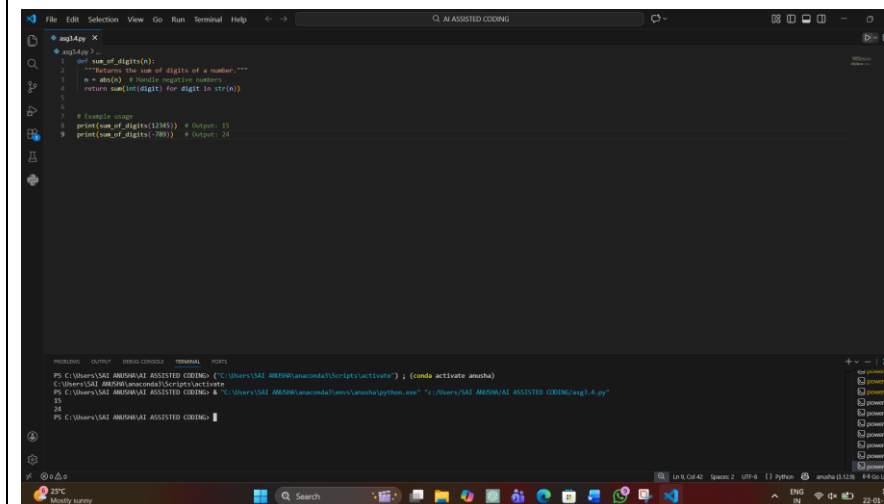
Expected Output #5

- 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?

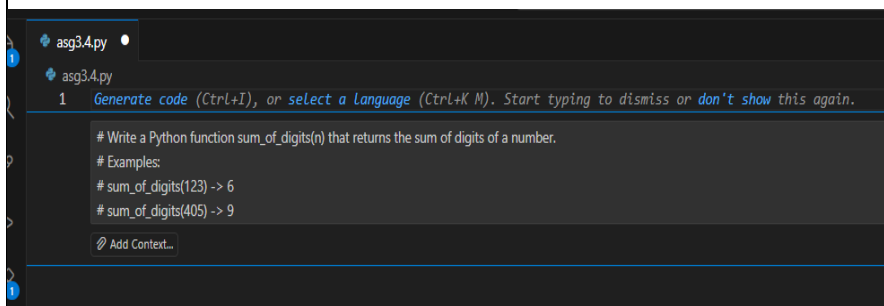
Generic task prompt:



Code and output:



Task + Input/Output example:



```
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
4
5     Args:
6     | n: An integer
7
8     Returns:
9     | The sum of all digits in n
10    """
11    return sum(int(digit) for digit in str(abs(n)))
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\SAI_AMISHA\AI_ASSISTED_CODING> & "C:\Users\SAI_AMISHA\anaconda3\envs\anusha\python.exe" "C:\Users\SAI_AMISHA\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>	
--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--