

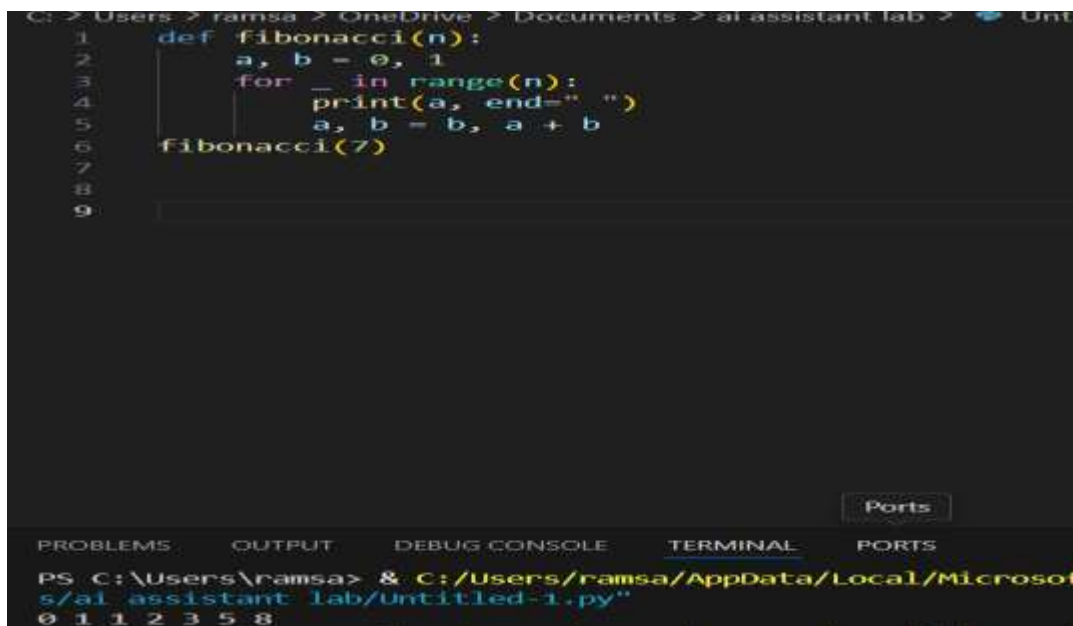
B.YASHWANTH KUMAR
ROLL NO: 2503A51L47
BATCH:20

TASK1

- PROMPT GIVEN: Write a Python function that prints the first N Fibonacci numbers.
- Python function generated by Copilot without any example provided.

```
def fibonacci(n):  
    a, b = 0, 1  
    for _ in range(n):  
        print(a, end=" ")  
        a, b = b, a + b
```

- output for sample input N = 7.



The screenshot shows a code editor with a dark theme. The top part displays the Python code for the fibonacci function. Below the code, the function is called with the argument 7. The bottom part of the screenshot shows the output of the program, which is the sequence of the first 7 Fibonacci numbers: 0 1 1 2 3 5 8. The editor interface includes a terminal at the bottom with the command prompt and the file path.

```
1 def fibonacci(n):  
2     a, b = 0, 1  
3     for _ in range(n):  
4         print(a, end=" ")  
5         a, b = b, a + b  
6 fibonacci(7)  
7  
8  
9
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\ramsa> & C:/Users/ramsa/AppData/Local/Microsoft/ai assistant lab/Untitled-1.py
0 1 1 2 3 5 8

- Observation on how Copilot understood the instruction with zero context.

When provided with the single-line comment prompt, GitHub Copilot correctly interpreted the request and generated a complete Python function to print the first N Fibonacci numbers. Despite receiving no examples or contextual code, Copilot inferred the standard iterative approach, initialized the first two Fibonacci values (0 and 1), and used a loop to generate the sequence. For the test input $N = 7$, the function produced the expected output 0 1 1 2 3 5 8, demonstrating that Copilot understood the mathematical logic of the Fibonacci series purely from the natural language instruction.

TASK2

- PROMPT GIVEN: One-shot Prompt – List Reversal Function..

```
def reverse_list_slice(lst):  
    return lst[::-1]
```

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

```
1 def reverse_list_slice(lst):  
2     return lst[::-1]  
3 print(reverse_list_slice([1,2,3])) |
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\ramsa> & C:/Users/ramsa/AppData/Local/Microsoft  
s/ai assistant lab/Untitled-1.py"  
[3, 2, 1]  
PS C:\Users\ramsa>
```

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

When a user provides even **one concrete example**, Copilot shifts from general guessing to **targeted generation**. That example acts as an anchor, helping Copilot:

- **Clarify ambiguity:** Abstract instructions like “reverse a list” could be interpreted in multiple ways. But when paired with an example like $[1, 2, 3] \rightarrow [3, 2, 1]$, Copilot instantly understands the expected behavior.
- **Match output format:** The example sets expectations for structure, style, and formatting—whether it's code, text, or data.
- **Reduce error margin:** With a reference point, Copilot avoids common mistakes like reversing in-place when a new list is expected, or misinterpreting the data type.

TASK3

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.

- THE CODE:

```
if __name__ == "__main__":
    test_cases = [
        "Hello world.",
        "This is a test.",
        "hello world.",
        "Hello world",
        "Hi.",
        "no period",
        "Another Example."
    ]
    for s in test_cases:
        print(f"is_valid({s!r}) -> {is_valid(s)}")
```

- OUTPUT:

```
PS C:\Users\ramsa> & C:/Users/ramsa/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/ramsa/Downloads/WEB DESIGNING/ailab3.py"
is_valid('Hello world.') -> True
is_valid('This is a test.') -> True
is_valid('hello world.') -> False
is_valid('Hello world') -> False
is_valid('Hi.') -> True
is_valid('no period') -> False
is_valid('Another Example.') -> True
```

- **Students reflect on how multiple examples guide Copilot to generate more accurate code.**
 1. By providing multiple examples, Copilot can better understand the intended pattern and logic for the function.
 2. This helps Copilot distinguish between valid and invalid cases, resulting in more accurate code generation.

3. For instance, Copilot learns that the string must start with a capital letter and end with a period, and can avoid common mistakes.
4. The more examples given, the more reliable and precise Copilot's suggestions become..

TASK 4

- **First, prompt Copilot to write an email validation function using zero-shot (just the task in comment).**

```
def is_valid_email(email):  
    import re  
    pattern = r'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'  
    return re.match(pattern, email) is not None
```

- **rewrite the prompt using few-shot examples.**

```
if __name__ == "__main__":  
    test_cases = [  
        "test@example.com",  
        "hello.world@domain.co.in",  
        "invalidemail.com",  
        "user@.com",  
        "abc@domain"  
    ]  
    for email in test_cases:  
        print(f"{email}: {is_valid_email(email)}")
```

- **Output:**

```
PS C:\Users\ramsa> & C:/Users/ramsa/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/ramsa/Downloads/WEB DESIGNING/aiLab3.4.py"  
test@example.com: True  
hello.world@domain.co.in: True  
invalidemail.com: False  
user@.com: False  
abc@domain: False  
□
```

TASK 5

- Experiment with 2 different prompt styles to generate a function that returns the sum of digits of a number.
- Two versions of the sum of digits() function.

1.

```
def sum_of_digits_generic(n):  
    return sum(int(d) for d in str(abs(n)))
```

2.

```
def sum_of_digits(n):  
    total = 0  
    for digit in str(abs(n)):  
        total += int(digit)  
    return total
```

- Example Output: sum of digits(123) → 6..

Program:

```
def sum_of_digits_example(n):  
    total = 0  
    for digit in str(abs(n)):  
        total += int(digit)  
    return total  
  
if __name__ == "__main__":  
    test_cases = [123]  
    print("Generic prompt results:")  
    for num in test_cases:  
        print(f"sum_of_digits_generic({num}) -> {sum_of_digits_generic(num)}")  
    print("\nExample prompt results:")  
    for num in test_cases:  
        print(f"sum_of_digits_example({num}) -> {sum_of_digits_example(num)}")
```

Output:

```
PS C:\Users\ransa> & C:\Users\ransa\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/Users/ransa/Downloads/WEB DESIGNING/ai1ab3.5.py"
Generic prompt results:
sum_of_digits_generic(123) -> 6

Example prompt results:
sum_of_digits_example(123) -> 6
```

- **Short analysis: which prompt produced cleaner or more optimized code and why?**

1.The generic task prompt produced a concise and optimized solution using Python's built-in sum and generator expression:

2. return sum(int(d) for d in str(abs(n)))

3.This is both clean and efficient for this simple problem.

4.The input/output example prompt led to a more explicit, step-by-step approach using a loop and accumulator variable.

5.While both are correct, the generic prompt is more Pythonic and compact, whereas the example-based prompt is easier to understand for beginners.

6.For simple tasks, generic prompts often yield more optimized code. For complex logic, examples help Copilot generate more reliable and specific solutions.