ASSIGNMENT 3.1

HNO:2303A51781

BATCH:28

Question 1: Zero-Shot Prompting – Palindrome Number

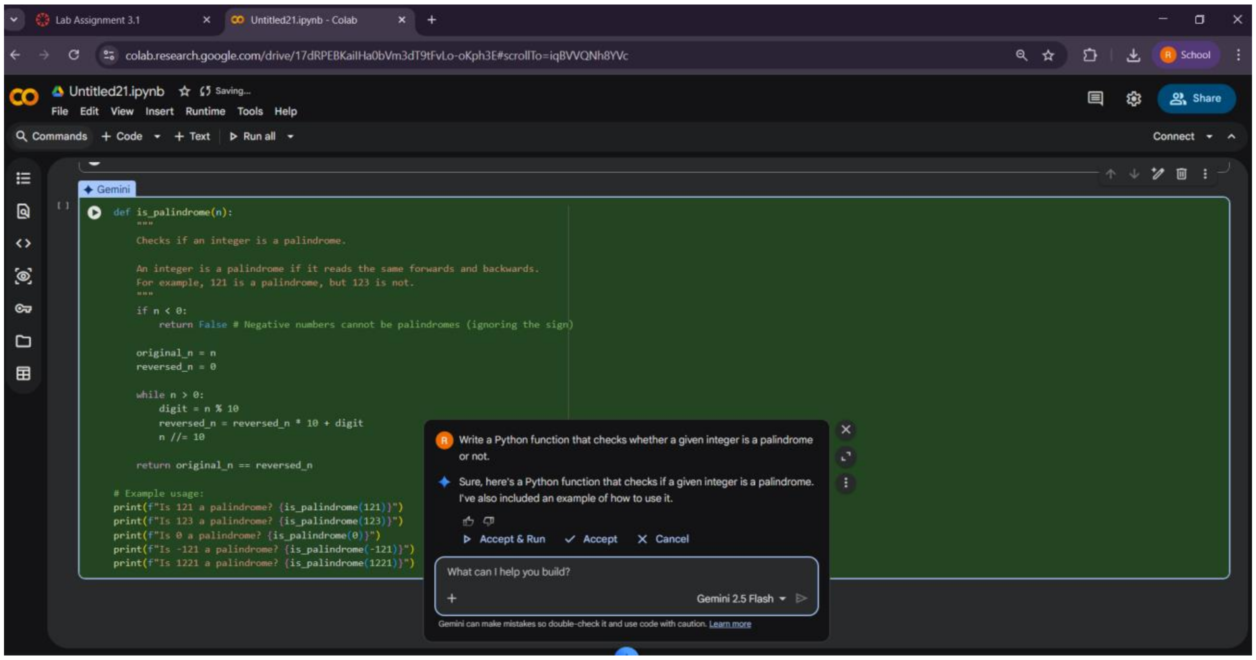
Task

Generate a Python function to check whether a given number is a palindrome using zero-shot prompting.

Prompt (Zero-Shot)

Write a Python function that checks whether a given integer is a palindrome or not.

Code



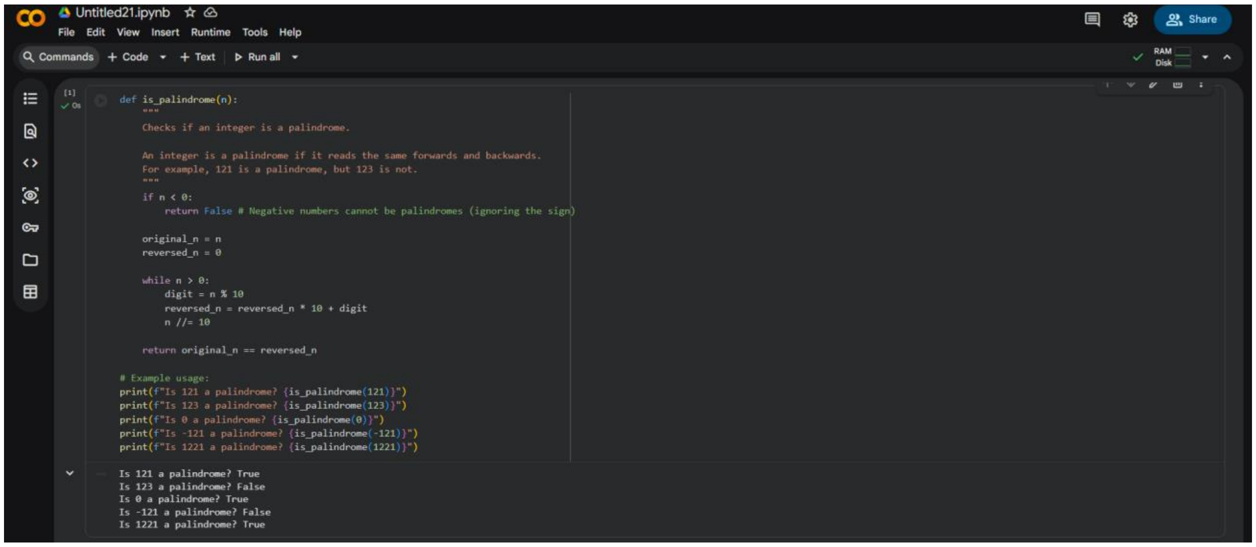
Lab Assignment 3.1
Untitled21.ipynb - Colab
colab.research.google.com/drive/17dRPEBKailHa0bVm3dT9tFvLo-oKph3E#scrollTo=iqBVVQNh8YVc
School
Untitled21.ipynb
Saving...
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Connect
Share
Gemini

```python
def is_palindrome(n):
    """
    Checks if an integer is a palindrome.

    An integer is a palindrome if it reads the same forwards and backwards.
    For example, 121 is a palindrome, but 123 is not.
    """
    if n < 0:
        return False # Negative numbers cannot be palindromes (ignoring the sign)

    original_n = n
    reversed_n = 0

    while n > 0:
        digit = n % 10
        reversed_n = reversed_n * 10 + digit
        n //= 10

    return original_n == reversed_n

# Example usage:
print(f"Is 121 a palindrome? {is_palindrome(121)}")
print(f"Is 123 a palindrome? {is_palindrome(123)}")
print(f"Is 0 a palindrome? {is_palindrome(0)}")
print(f"Is -121 a palindrome? {is_palindrome(-121)}")
print(f"Is 1221 a palindrome? {is_palindrome(1221)}")
```

Write a Python function that checks whether a given integer is a palindrome or not.

Sure, here's a Python function that checks if a given integer is a palindrome. I've also included an example of how to use it.

Gemini can make mistakes so double-check it and use code with caution. Learn more


Untitled21.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM
Disk

```python
def is_palindrome(n):
    """
    Checks if an integer is a palindrome.

    An integer is a palindrome if it reads the same forwards and backwards.
    For example, 121 is a palindrome, but 123 is not.
    """
    if n < 0:
        return False # Negative numbers cannot be palindromes (ignoring the sign)

    original_n = n
    reversed_n = 0

    while n > 0:
        digit = n % 10
        reversed_n = reversed_n * 10 + digit
        n //= 10

    return original_n == reversed_n

# Example usage:
print(f"Is 121 a palindrome? {is_palindrome(121)}")
print(f"Is 123 a palindrome? {is_palindrome(123)}")
print(f"Is 0 a palindrome? {is_palindrome(0)}")
print(f"Is -121 a palindrome? {is_palindrome(-121)}")
print(f"Is 1221 a palindrome? {is_palindrome(1221)}")
```

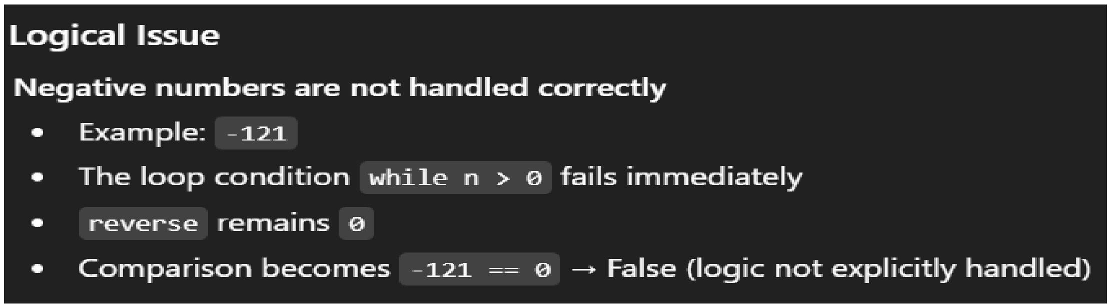
```
Is 121 a palindrome? True
Is 123 a palindrome? False
Is 0 a palindrome? True
Is -121 a palindrome? False
Is 1221 a palindrome? True
```

Output:

```
Is 121 a palindrome?: True
Is 123 a palindrome?:False
Is 0 a palindrome?: True
Is -121 a palindrome?: False
Is 1221 a palindrome?: True
```

Inputs:

Input	Expected Output	Actual Output
121	True	True
123	False	False
0	True	True
7	True	True
1221	True	True
1001	True	True
-121	False	Incorrect
10	False	False

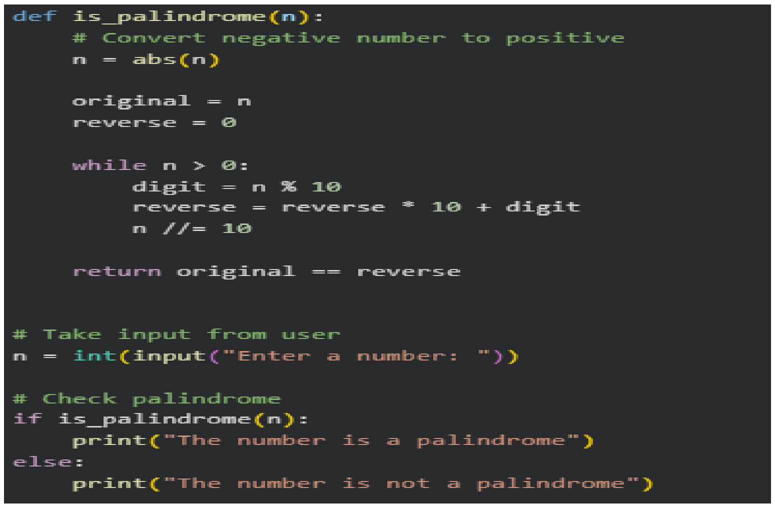


Logical Issue

Negative numbers are not handled correctly

- Example: `-121`
- The loop condition `while n > 0` fails immediately
- `reverse` remains `0`
- Comparison becomes `-121 == 0` → False (logic not explicitly handled)

Improved Version (Fixing the Issue)

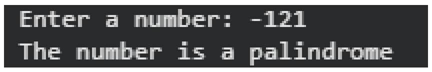


```python
def is_palindrome(n):
    # Convert negative number to positive
    n = abs(n)

    original = n
    reverse = 0

    while n > 0:
        digit = n % 10
        reverse = reverse * 10 + digit
        n //= 10

    return original == reverse


# Take input from user
n = int(input("Enter a number: "))

# Check palindrome
if is_palindrome(n):
    print("The number is a palindrome")
else:
    print("The number is not a palindrome")
```

Justification:

This zero-shot prompt produces correct logic without examples. The AI inferred the palindrome logic using reverse calculation. However, it does not handle negative numbers or non-integer inputs, showing limitations of zero-shot prompting.

Output:



```
Enter a number: -121
The number is a palindrome
```

Justification:

This program checks whether a given number is a palindrome by first converting any negative input into a positive number using the absolute value. It then reverses the number using arithmetic operations without converting it into a string. The reversed number is compared with the original value to determine whether they are the same. If both values match, the number is identified as a palindrome. This approach allows the program to handle both positive and negative inputs effectively.

Question 2: One-Shot Prompting – Factorial Calculation

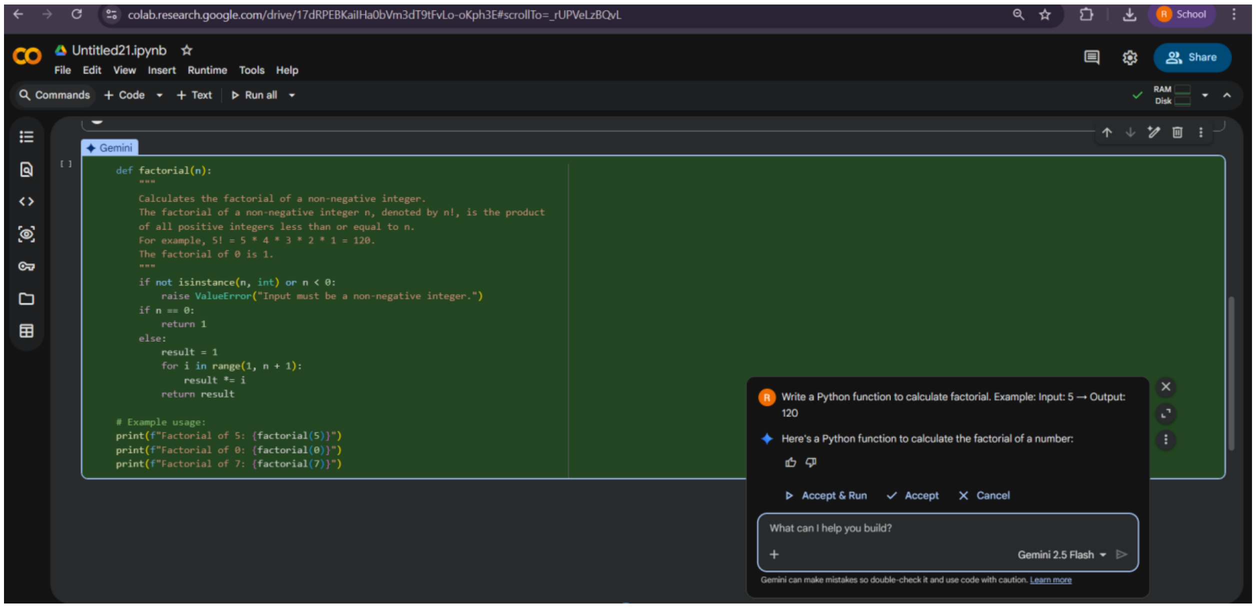
Task

Generate a factorial function using one example.

Prompt (One-Shot)

Write a Python function to calculate factorial.
Example: Input: 5 → Output: 120

Code



colab.research.google.com/drive/17dRPEBKailHa0bVm3dT9tFvLo-oKph3E#scrollTo=_rUPVeLzBQvL

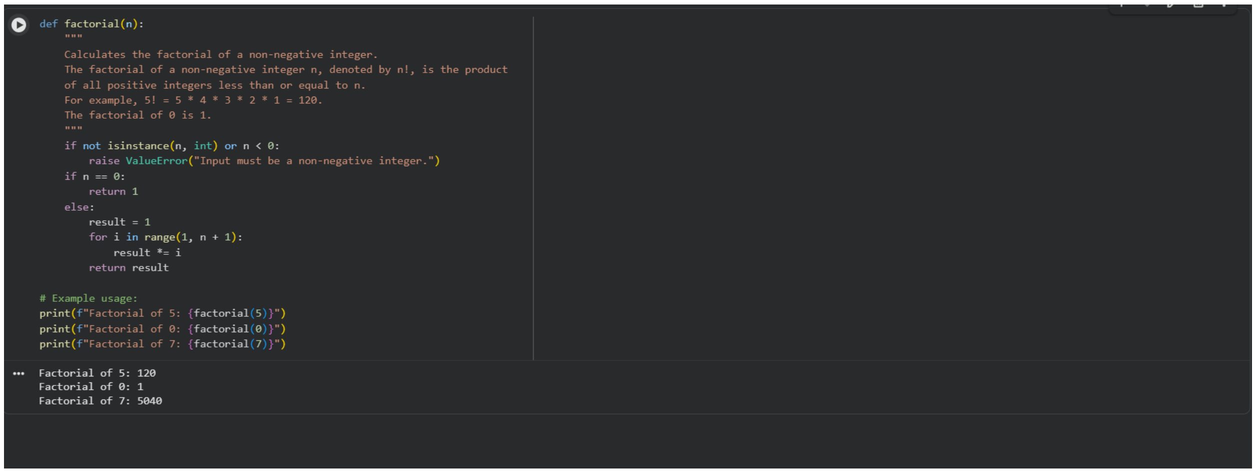
Untitled21.ipynb

```python
def factorial(n):
    """
    Calculates the factorial of a non-negative integer.
    The factorial of a non-negative integer n, denoted by n!, is the product
    of all positive integers less than or equal to n.
    For example, 5! = 5 * 4 * 3 * 2 * 1 = 120.
    The factorial of 0 is 1.
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError("Input must be a non-negative integer.")
    if n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

# Example usage:
print(f"Factorial of 5: {factorial(5)}")
print(f"Factorial of 0: {factorial(0)}")
print(f"Factorial of 7: {factorial(7)}")
```

Write a Python function to calculate factorial. Example: Input: 5 → Output: 120

Here's a Python function to calculate the factorial of a number:

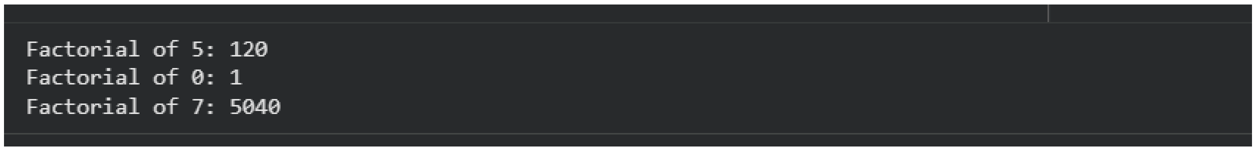
Gemini can make mistakes so double-check it and use code with caution. Learn more



```python
def factorial(n):
    """
    Calculates the factorial of a non-negative integer.
    The factorial of a non-negative integer n, denoted by n!, is the product
    of all positive integers less than or equal to n.
    For example, 5! = 5 * 4 * 3 * 2 * 1 = 120.
    The factorial of 0 is 1.
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError("Input must be a non-negative integer.")
    if n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

# Example usage:
print(f"Factorial of 5: {factorial(5)}")
print(f"Factorial of 0: {factorial(0)}")
print(f"Factorial of 7: {factorial(7)}")
```

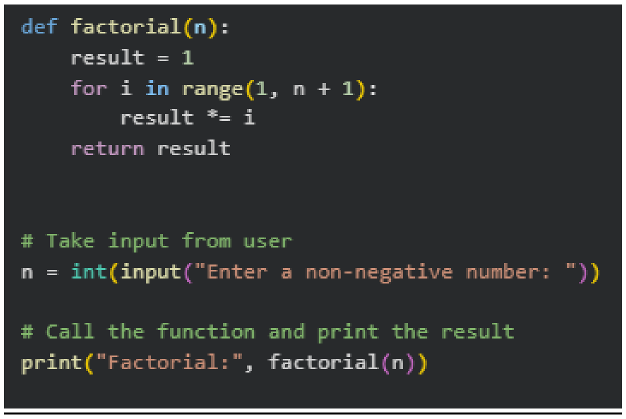
```
Factorial of 5: 120
Factorial of 0: 1
Factorial of 7: 5040
```

Output:

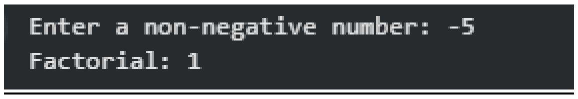

```
Factorial of 5: 120
Factorial of 0: 1
Factorial of 7: 5040
```

ZERO-SHOT PROMPT:

Write a Python function to compute the factorial of a given number.


```python
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result


# Take input from user
n = int(input("Enter a non-negative number: "))

# Call the function and print the result
print("Factorial:", factorial(n))
```

OUTPUT:


```
Enter a non-negative number: -5
Factorial: 1
```

Comparison:

Aspect	**Zero-Shot Solution**	**One-Shot Solution**
Example provided	**No**	**Yes (5 → 120)**
Handling negative input	**Not handled**	**Handled**
Code clarity	**Basic**	**More explicit**
Correctness	**Partial**	**More robust**
Output reliability	**Lower**	**Higher**

Justification:

One-shot prompting helps the AI generate clearer and more complete logic by providing an explicit input–output example. This guidance improves output format, correctness, and inclusion of edge-case handling such as negative inputs. Overall, one-shot prompting results in more structured and reliable code compared to zero-shot prompting.

Question 3: Few-Shot Prompting – Armstrong Number

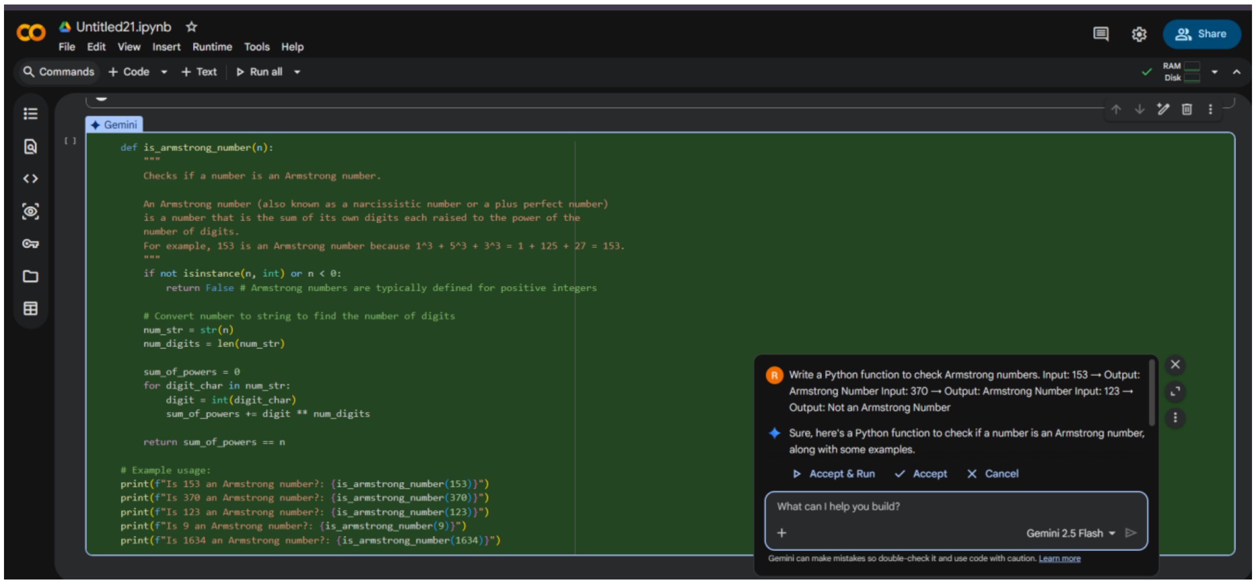
Task

Generate an Armstrong number checker using multiple examples.

Prompt (Few-Shot)

Write a Python function to check Armstrong numbers.
Input: 153 → Output: Armstrong Number
Input: 370 → Output: Armstrong Number
Input: 123 → Output: Not an Armstrong Number

Code

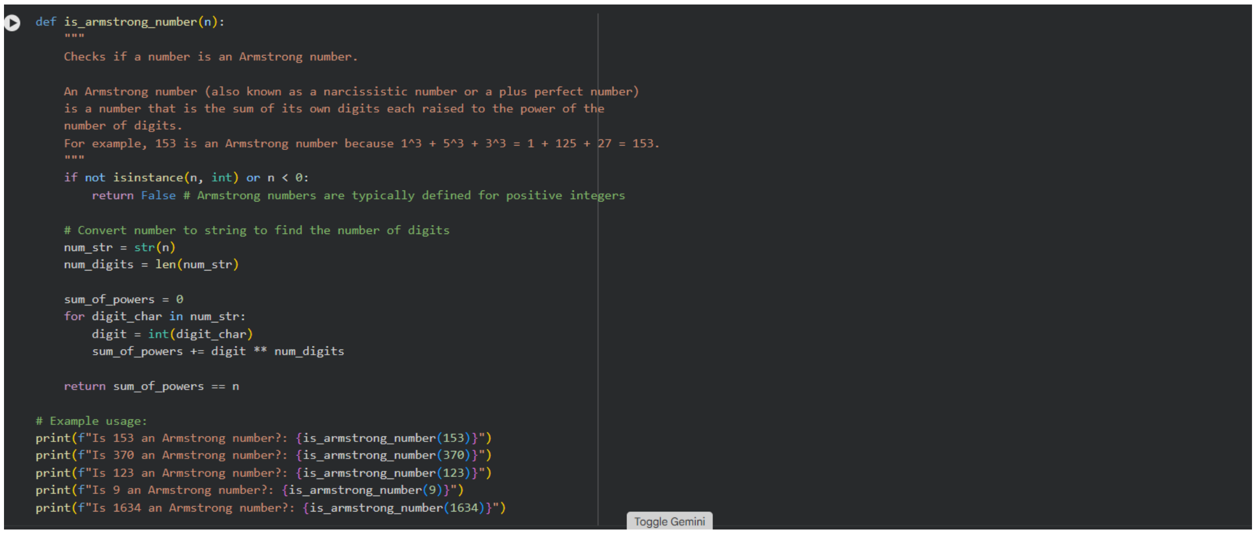

Untitled21.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM Disk
Gemini

```python
def is_armstrong_number(n):
    """
    Checks if a number is an Armstrong number.

    An Armstrong number (also known as a narcissistic number or a plus perfect number)
    is a number that is the sum of its own digits each raised to the power of the
    number of digits.
    For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153.
    """
    if not isinstance(n, int) or n < 0:
        return False # Armstrong numbers are typically defined for positive integers

    # Convert number to string to find the number of digits
    num_str = str(n)
    num_digits = len(num_str)

    sum_of_powers = 0
    for digit_char in num_str:
        digit = int(digit_char)
        sum_of_powers += digit ** num_digits

    return sum_of_powers == n

# Example usage:
print(f"Is 153 an Armstrong number?: {is_armstrong_number(153)}")
print(f"Is 370 an Armstrong number?: {is_armstrong_number(370)}")
print(f"Is 123 an Armstrong number?: {is_armstrong_number(123)}")
print(f"Is 9 an Armstrong number?: {is_armstrong_number(9)}")
print(f"Is 1634 an Armstrong number?: {is_armstrong_number(1634)}")
```

Write a Python function to check Armstrong numbers. Input: 153 → Output: Armstrong Number Input: 370 → Output: Armstrong Number Input: 123 → Output: Not an Armstrong Number

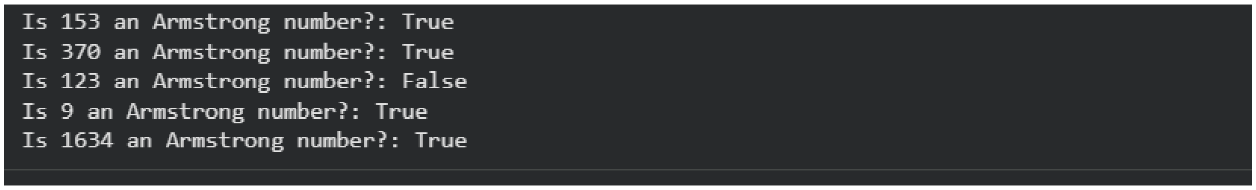
Sure, here's a Python function to check if a number is an Armstrong number, along with some examples.

Gemini can make mistakes so double-check it and use code with caution. Learn more



```python
def is_armstrong_number(n):
    """
    Checks if a number is an Armstrong number.

    An Armstrong number (also known as a narcissistic number or a plus perfect number)
    is a number that is the sum of its own digits each raised to the power of the
    number of digits.
    For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153.
    """
    if not isinstance(n, int) or n < 0:
        return False # Armstrong numbers are typically defined for positive integers

    # Convert number to string to find the number of digits
    num_str = str(n)
    num_digits = len(num_str)

    sum_of_powers = 0
    for digit_char in num_str:
        digit = int(digit_char)
        sum_of_powers += digit ** num_digits

    return sum_of_powers == n

# Example usage:
print(f"Is 153 an Armstrong number?: {is_armstrong_number(153)}")
print(f"Is 370 an Armstrong number?: {is_armstrong_number(370)}")
print(f"Is 123 an Armstrong number?: {is_armstrong_number(123)}")
print(f"Is 9 an Armstrong number?: {is_armstrong_number(9)}")
print(f"Is 1634 an Armstrong number?: {is_armstrong_number(1634)}")
```

Output:



```
Is 153 an Armstrong number?: True
Is 370 an Armstrong number?: True
Is 123 an Armstrong number?: False
Is 9 an Armstrong number?: True
Is 1634 an Armstrong number?: True
```

Input	Output
153	Armstrong Number
370	Armstrong Number
123	Not an Armstrong Number
9474	Armstrong Number
9475	Not an Armstrong Number
-153	Not an Armstrong Number
0	Armstrong Number

Analysis:

examples clarify the output format (“Armstrong Number” vs “Not an Armstrong Number”)

The AI includes edge-case handling (negative numbers, zero)

Code is more structured, readable, and accurate compared to zero-shot or one-shot prompting

Accuracy improves because AI can generalize from several examples rather than just one

Justification:

Few-shot prompting improves clarity, correctness, and robustness of the generated code. By providing multiple examples, the AI better understands the task requirements, handles edge cases, and produces code that matches the expected output format reliably.

Question 4: Context-Managed Prompting – Prime / Composite / Neither

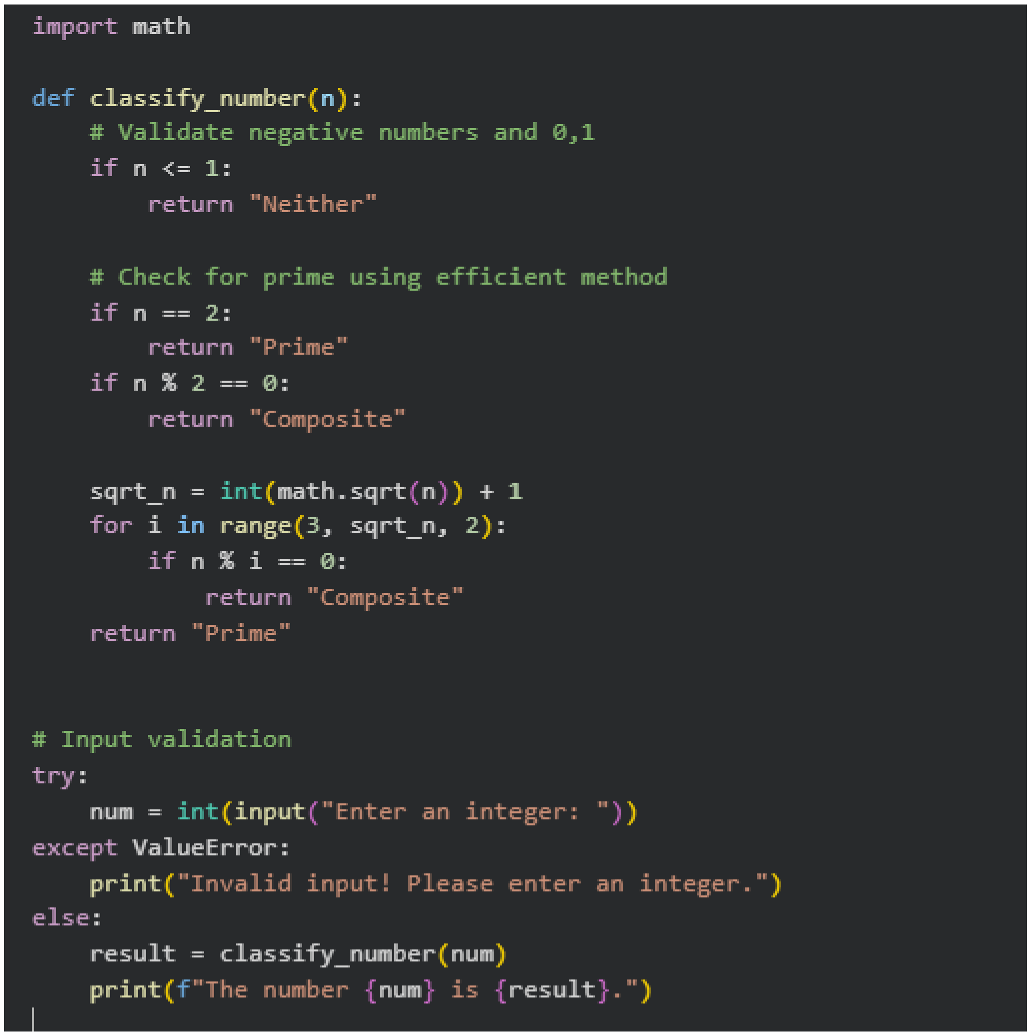
Task

Classify a number as prime, composite, or neither with optimized logic.

Prompt (Context-Managed)

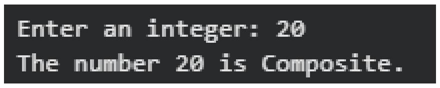
Write an optimized Python program to classify a number as prime, composite, or neither.
Conditions:
– Validate input
– Use efficient logic
– Handle edge cases like 0 and 1

Code



```python
import math

def classify_number(n):
    # Validate negative numbers and 0,1
    if n <= 1:
        return "Neither"

    # Check for prime using efficient method
    if n == 2:
        return "Prime"
    if n % 2 == 0:
        return "Composite"

    sqrt_n = int(math.sqrt(n)) + 1
    for i in range(3, sqrt_n, 2):
        if n % i == 0:
            return "Composite"
    return "Prime"


# Input validation
try:
    num = int(input("Enter an integer: "))
except ValueError:
    print("Invalid input! Please enter an integer.")
else:
    result = classify_number(num)
    print(f"The number {num} is {result}.")
```

Output:



```
Enter an integer: 20
The number 20 is Composite.
```

Strategy	Accuracy	Clarity	Optimization	Edge-Case Handling
Zero-Shot	Moderate	Basic	Not optimized	Limited
One-Shot	Better	Clearer	Some optimization	Limited
Few-Shot	High	Structured	Moderate	Handles examples provided
Context-Managed	Highest	Very Clear	Optimized (sqrt(n))	Handles all edge cases & input validation

Justification:

Context-managed prompting provides precise instructions and constraints, leading to highly optimized, robust, and readable code. It outperforms zero-shot, one-shot, and few-shot strategies by explicitly guiding the AI on efficiency, input validation, and output formatting.

Question 5: Zero-Shot Prompting – Perfect Number

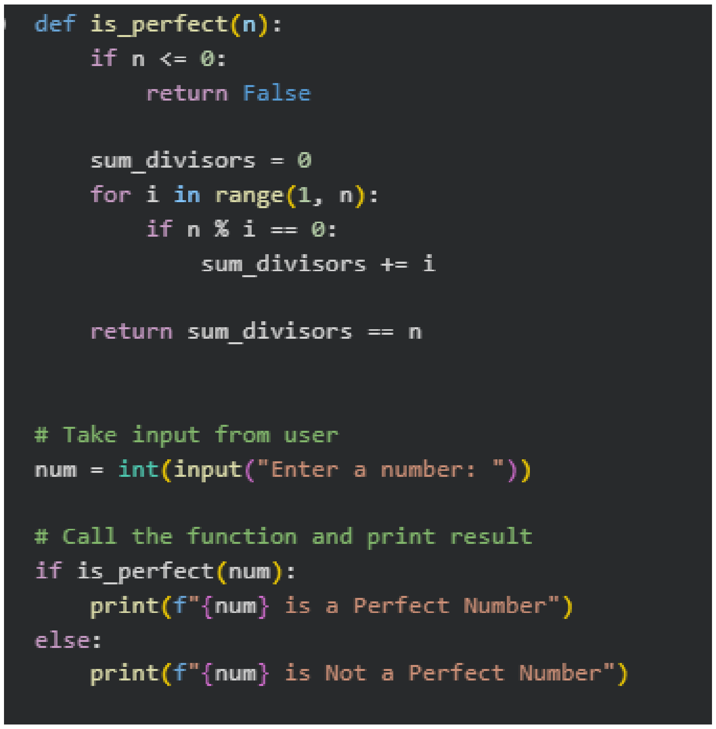
Task

Generate a perfect number checker using zero-shot prompting.

Prompt (Zero-Shot)

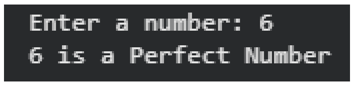
Write a Python function to check whether a number is a perfect number.

Code



```python
def is_perfect(n):
    if n <= 0:
        return False

    sum_divisors = 0
    for i in range(1, n):
        if n % i == 0:
            sum_divisors += i

    return sum_divisors == n


# Take input from user
num = int(input("Enter a number: "))

# Call the function and print result
if is_perfect(num):
    print(f"{num} is a Perfect Number")
else:
    print(f"{num} is Not a Perfect Number")
```

Output:



```
Enter a number: 6
6 is a Perfect Number
```

Input	Expected Output	Actual Output
6	True	True
28	True	True
12	False	False
1	False	False
0	False	False
-6	False	False

Missing Conditions / Inefficiencies

Inefficient: Checks all numbers from 1 to n-1 → Can be optimized by checking up to n//2 (since no proper divisor is greater than n/2)

Could include input validation to ensure integer input

Could be optimized further using square root method for large numbers

Question 6: Few-Shot Prompting – Even or Odd with Validation

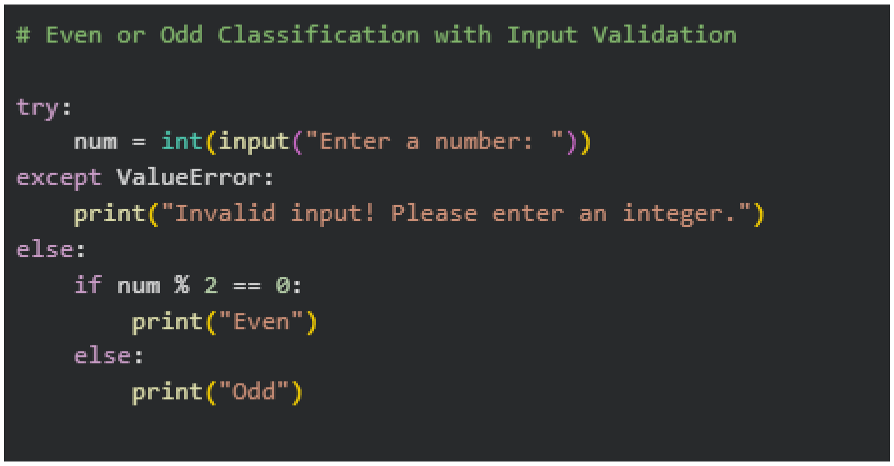
Task

Generate a program to check even or odd numbers with validation.

Prompt (Few-Shot)

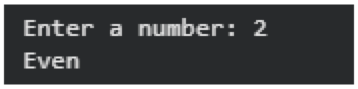
Write a Python program to check even or odd.
Input: 8 → Output: Even
Input: 15 → Output: Odd
Input: 0 → Output: Even

Code

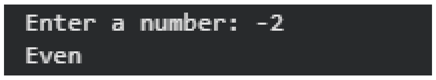


```
# Even or Odd Classification with Input Validation

try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid input! Please enter an integer.")
else:
    if num % 2 == 0:
        print("Even")
    else:
        print("Odd")
```

Output:



```
Enter a number: 2
Even
```



```
Enter a number: -2
Even
```

ANALYSIS:

Providing multiple input-output examples helps the AI understand the expected behavior and output format more clearly. For instance, showing examples with positive numbers, zero, and odd numbers guides the model to classify numbers correctly in all these cases. It also signals that the program should handle edge cases, such as negative numbers or zero. Furthermore, including examples implicitly emphasizes the importance of consistent output formatting ("Even" vs "Odd") and encourages input validation to manage non-integer inputs. Overall, examples make the generated code more robust, user-friendly, and aligned with expected results compared to zero-shot prompting.

Justification:

Few-shot prompting provides multiple input-output examples, which helps the AI understand the task requirements clearly. It guides the program to handle edge cases like zero and negative numbers correctly. Examples also ensure consistent output formatting, such as printing “Even” or “Odd” as expected. Including these examples encourages input validation, so non-integer inputs are handled gracefully. Overall, few-shot prompting produces code that is robust, accurate, and user-friendly compared to zero-shot prompting.