

## Assignment 3.1

Name:K Pavani

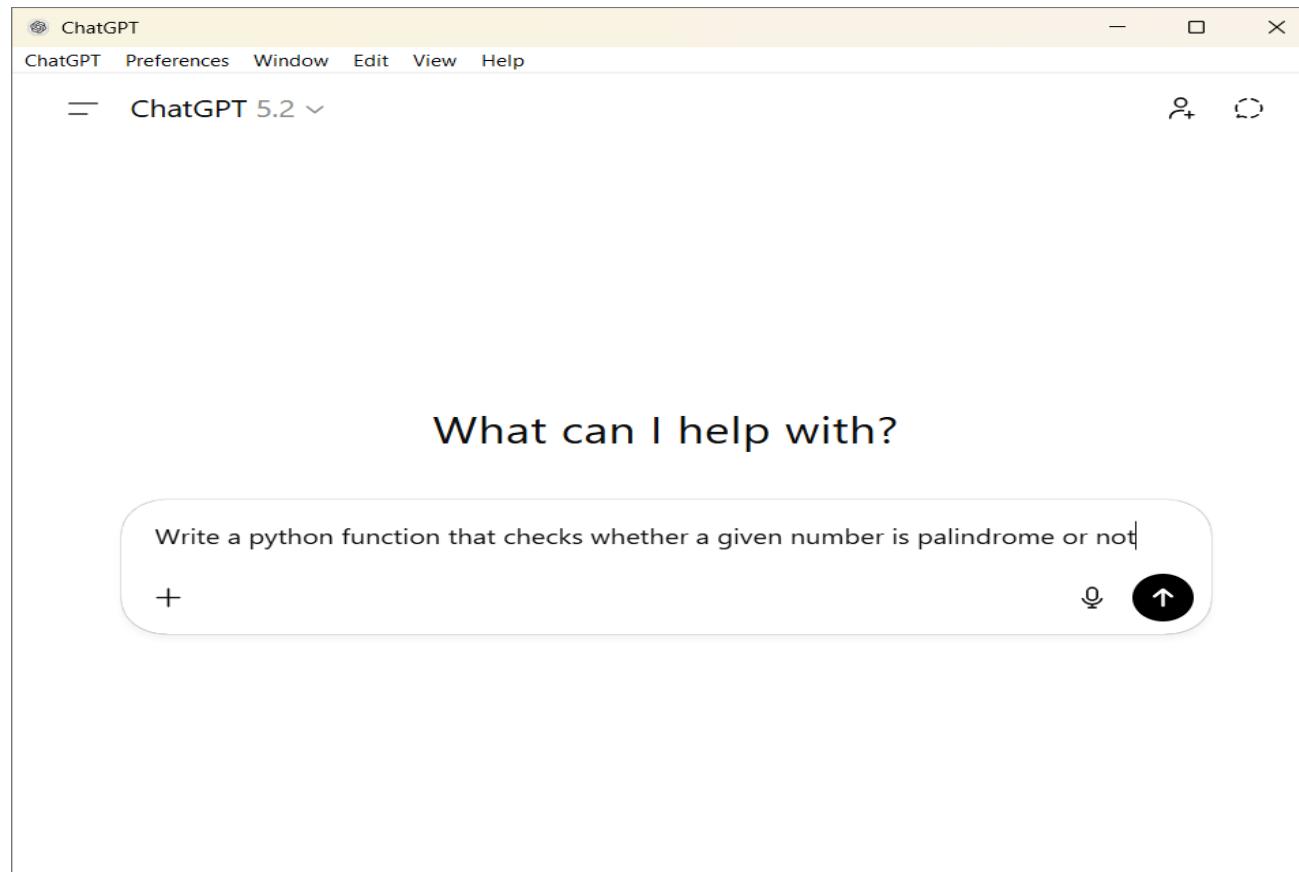
Hall-ticket:2303A51494

Batch:27

### Question 1: Zero-Shot Prompting (Palindrome Number Program)

#### Prompt:

Write a python program that checks whether a given number is palindrome or not



#### Code and Output:

```
[4]  Os
▶ def is_palindrome(num):
    original = num
    reverse = 0

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

    return original == reverse

print(is_palindrome(121))
print(is_palindrome(1254))
print(is_palindrome(12221))
```

```
▼ ... True
    False
    True
```

#### Justification:

- It accurately checks if the number reads the same forward and backward by reversing it.
- It works for any integer, positive or negative, depending on how you define a numeric palindrome.
- The code is straightforward and easy to read and understand

#### Question 2: One-Shot Prompting (Factorial Calculation)

##### Prompt:

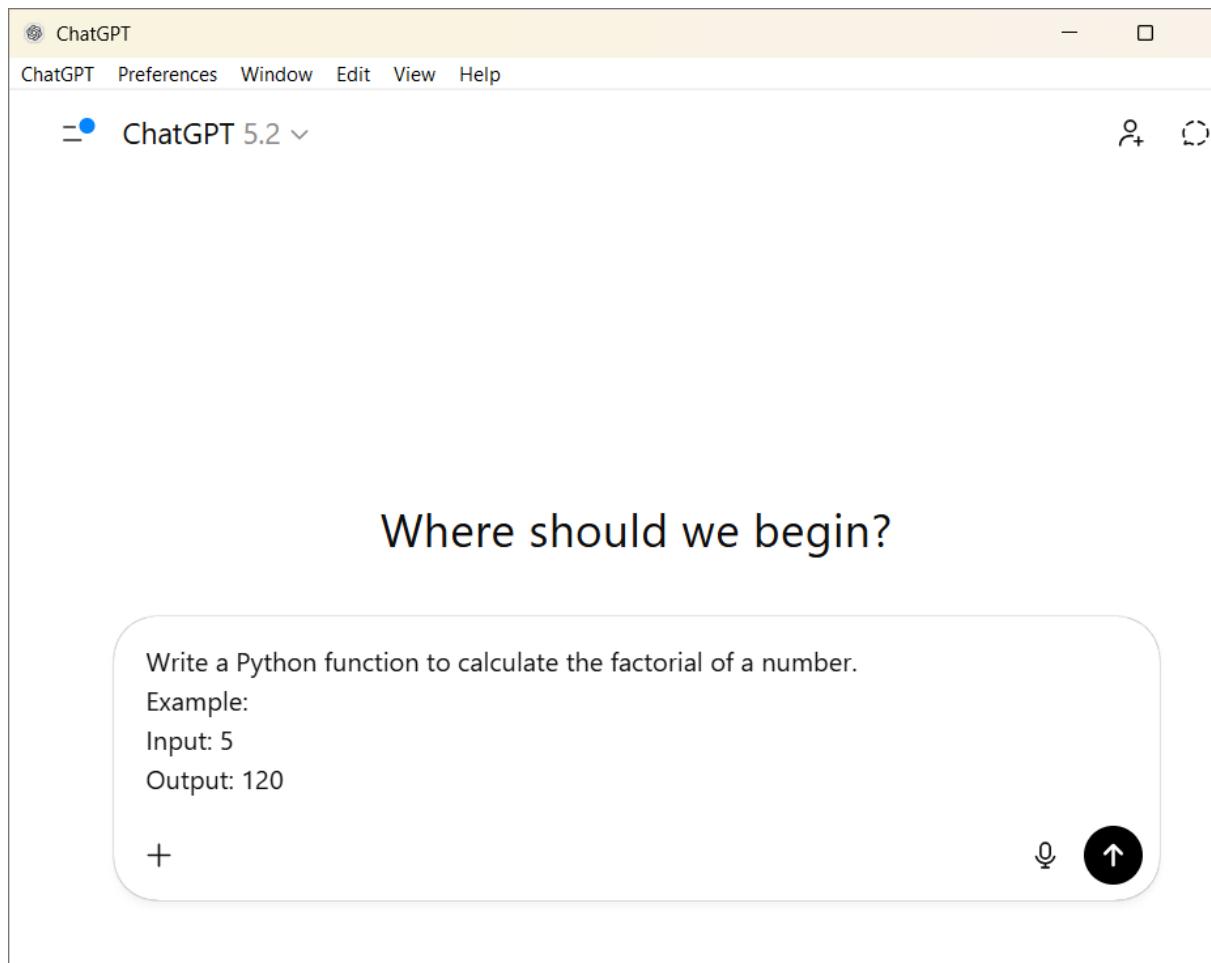
write a python program that calculate the factorial of a number

Example:

Input: 5

Output: 120

#### Code and Output:



The screenshot shows a code editor interface with a code cell. The code cell has a status bar indicating '[5] 0s'. The code itself is:

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

# Example
print(factorial(5)) # Output: 120
```

At the bottom of the code cell, there is a line starting with '...', followed by '120'.

### Explanation:

- The function `factorial(n)` is used to find the factorial of a number.
- `result` starts as 1 to store the multiplication value.
- The loop multiplies numbers from 1 to `n` with `result`.
- For input 5, it calculates  $1 \times 2 \times 3 \times 4 \times 5 = 120$ .

- For a negative number, the loop does not run, so the code does not handle negatives correctly.

Aspect	Zero-Shot Prompt	Zero-Shot Prompt
Prompt length	Short and generic	Slightly longer
Error Probability	Higher	Lower
Code structure	Basic	Well-Structured
Interpretation by AI	Can be ambiguous	Clear due to example
Reusability of output	Limited	High

### Question3: Few-Shot Prompting (Armstrong Number Check)

**Prompt:** Input: 153 → Armstrong Number

Input: 370 → Armstrong Number

Input: 123 → Not an Armstrong Number

Using the above examples, generate a Python function to check whether a given number is an Armstrong number.

Ready when you are.

Input: 153 → Armstrong Number

Input: 370 → Armstrong Number

Input: 123 → Not an Armstrong Number

Using the above examples, generate a Python function to check whether a given number is an Armstrong number.

+

0

↑

**Code and Output:**

```
[7] ✓ 0s
▶ def is_armstrong(n):
    temp = n
    digits = len(str(n))
    total = 0

    while temp > 0:
        digit = temp % 10
        total += digit ** digits
        temp //= 10

    return total == n

# Example checks
print(is_armstrong(153)) # True → Armstrong Number
print(is_armstrong(370)) # True → Armstrong Number
print(is_armstrong(123)) # False → Not an Armstrong Number

▼ ...
  ... True
  ... True
  ... False
```

### Explanation:

- Providing multiple examples (153, 370, 123) helps the AI infer the correct logical pattern for Armstrong numbers.
- The examples guide the code to use digit extraction and power calculation instead of hard-coded values
- The structure becomes generalized, working for any number of digits.
- Overall correctness increases because the model validates logic across varied output
- Special cases such as 0 are correctly identified as Armstrong numbers.

### Question 4: Context-Managed Prompting (Optimized Number Classification)

#### Prompt:

You are an expert python developer write an optimized python program that classifies a number as prime composite or neither

constraints:

validate input

handle negative numbers

optimize loop for performance

How can I help, Pavani?

You are an expert python developer write an optimized python program that classifies a number as prime composite or neither

constraints:

- validate input
- handle negative numbers
- optimize loop for performance



## Code and output:

```
[8]  ✓ 0s
  def classify_number(n):
      # Input validation
      if not isinstance(n, int):
          return "Invalid input: Please enter an integer"

      # Handle negative numbers, 0, and 1
      if n <= 1:
          return "Neither Prime nor Composite"

      # 2 is the only even prime
      if n == 2:
          return "Prime"

      # Eliminate even numbers
      if n % 2 == 0:
          return "Composite"

      # Check divisibility up to √n (optimized)
      limit = int(n ** 0.5) + 1
      for i in range(3, limit, 2):
          if n % i == 0:
              return "Composite"

      return "Prime"

print(classify_number(-7))    # Neither Prime nor Composite
print(classify_number(0))     # Neither Prime nor Composite
print(classify_number(1))     # Neither Prime nor Composite
print(classify_number(2))     # Prime
print(classify_number(9))     # Composite
print(classify_number(17))    # Prime
print(classify_number(2.5))   # Invalid input

...
*** Neither Prime nor Composite
Neither Prime nor Composite
Neither Prime nor Composite
Prime
Composite
Prime
Invalid input: Please enter an integer
```

## Explanation:

- Input validation ensures only integers are processed, preventing incorrect classifications.

- Numbers less than or equal to 1 (including negatives) are correctly labeled as neither prime nor composite
- The number 2 is handled as a special case since it is the only even prime number.
- Even numbers greater than 2 are immediately classified as composite to reduce unnecessary checks
- Only odd divisors are tested, cutting the number of iterations nearly in half for better performance

#### Question 5: Zero-Shot Prompting (Perfect Number Check)

**Prompt:** Write a Python function that checks whether a given number is a perfect number.

What can I help with?

| Write a Python function that checks whether a given number is a perfect number.

+

0



**Code with output:**

```
[9] Os
  def is_perfect_number(n):
    # Validate input
    if not isinstance(n, int) or n <= 0:
        return False

    divisor_sum = 1 # 1 is a proper divisor for all n > 1

    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            divisor_sum += i
            if i != n // i:
                divisor_sum += n // i

    return n > 1 and divisor_sum == n
print(is_perfect_number(6))    # True
print(is_perfect_number(28))   # True
print(is_perfect_number(12))   # False
print(is_perfect_number(1))    # False
print(is_perfect_number(-6))   # False
```

```
✓ ... True
True
False
False
False
```

### Explanation:

- The function first validates the input to ensure it is a positive integer, since perfect numbers are defined only for positive integers.
- It initializes the divisor sum with 1 because 1 is a proper divisor of every number greater than 1.
- The loop runs only up to the square root of the number to reduce unnecessary iterations and improve efficiency.
- When a divisor is found, both the divisor and its corresponding pair are added to the sum to avoid missing factors.
- Finally, the function checks whether the sum of proper divisors is equal to the original number to determine if it is perfect.

### Question 6: Few-Shot Prompting (Even or Odd Classification with Validation)

#### Prompt:

Input: 8 → Even Input: 15 → Odd Input: 0 → Even Using the above examples, write a Python program that determines whether a number is even or odd with proper input validation

## Ready when you are.

Input: 8 → Even

Input: 15 → Odd

Input: 0 → Even

Using the above examples, write a Python program that determines whether a number is even or odd with proper input validation



### Code with output:

```
[10] ✓ Os
  def check_even_odd(num):
      if not isinstance(num, int):
          return "Invalid input: Please enter an integer"

      if num % 2 == 0:
          return "Even"
      else:
          return "Odd"

# Test cases based on given examples
print("Input: 8 →", check_even_odd(8))
print("Input: 15 →", check_even_odd(15))
print("Input: 0 →", check_even_odd(0))

# Invalid input test
print("Input: 3.5 →", check_even_odd(3.5))

...
*** Input: 8 → Even
      Input: 15 → Odd
      Input: 0 → Even
      Input: 3.5 → Invalid input: Please enter an integer
```

### Explanation:

- The function checks whether the input is an integer to ensure valid input.
- Modulus operator % is used to determine divisibility by 2.
- Numbers divisible by 2 are classified as Even.
- Numbers not divisible by 2 are classified as Odd.
- The program safely handles zero and invalid inputs.

