

## ASSIGNMENT\_6.3

Name:Ch.Jyothika

Ht.no:2303A51280

Batch:05

### Task Description #1: Classes (Student Class)

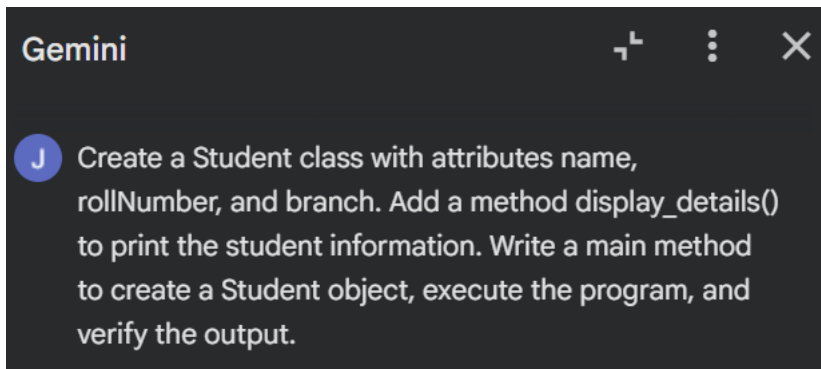
#### Scenario

You are developing a simple student information management module.

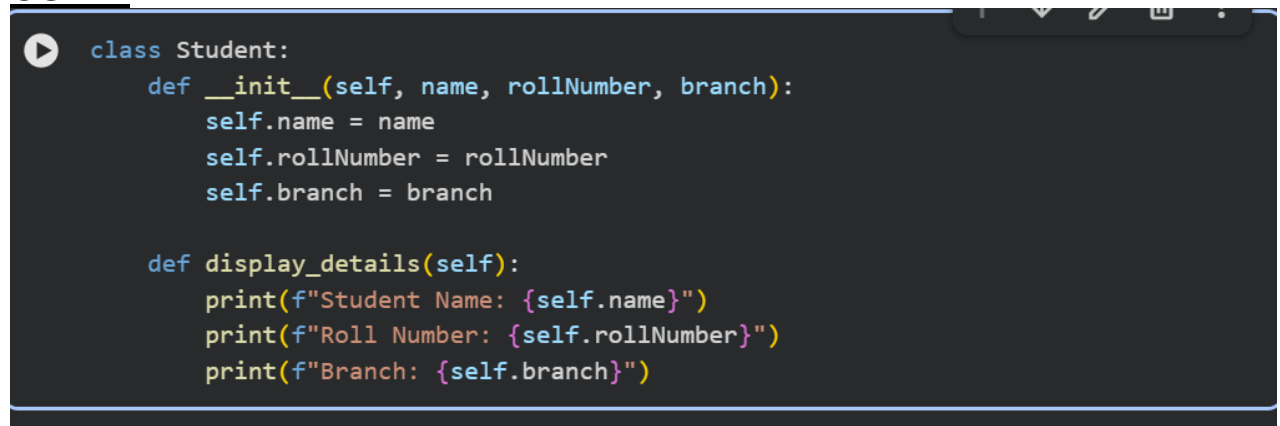
#### Task

- Use an AI tool (GitHub Copilot / Cursor AI / Gemini) to complete a Student class

#### PROMPT:



#### CODE:



#### OUTPUT:

```
# Create a Student object
student1 = Student("Alice Smith", "CS101", "Computer Science")

# Call the display_details method
student1.display_details()

... Student Name: Alice Smith
    Roll Number: CS101
    Branch: Computer Science
```

### EXPLANATION:

- A class named Student is created to represent student information.
- The `__init__()` constructor is defined and is called automatically when a Student object is created.
- The constructor initializes the student's name, roll number, and branch.
- A method `display_details()` is defined to print the student details.
- A Student object is created by passing values for name, roll number, and branch.
- The `display_details()` method is called using the object.
- The student information is displayed on the console.

## Task 2: Loops (Multiples of a Number)

### PROMPT:

Gemini

J Generate a python function that prints the first 10 multiples of a given number using a for loop. Analyze the loop logic. Then generate the same functionality using a while loop and compare both approaches.

✦ Certainly! I will generate Python functions to print the first 10 multiples of a given number using both `for` and `while` loops. I'll also include an analysis of their logic and a comparison of the two approaches.

### CODE:

```
def print_multiples_for_loop(number):  
    """  
    Prints the first 10 multiples of a given number using a for loop.  
  
    Args:  
        number (int): The number for which to print multiples.  
    """  
    for i in range(1, 11): # Iterate from 1 to 10 (inclusive)  
        multiple = number * i  
        print(f"{number} * {i} = {multiple}")
```

### OUTPUT:

▶ print\_multiples\_for\_loop(5)

```
... 5 * 1 = 5  
     5 * 2 = 10  
     5 * 3 = 15  
     5 * 4 = 20  
     5 * 5 = 25  
     5 * 6 = 30  
     5 * 7 = 35  
     5 * 8 = 40  
     5 * 9 = 45  
     5 * 10 = 50
```

## **EXPLANATION**

- A function is defined that accepts a number as input.
- A loop is used to repeat the process 10 times.
- In each iteration, the input number is multiplied by the loop counter.
- The result of the multiplication gives one multiple of the number.
- Each multiple is printed on the console.
- The loop continues until the first 10 multiples are displayed.
- When the loop ends, the program stops execution.

## **Task 3: Conditional Statements (Age Classification)**

### **PROMPT:**

**J** Generate an age classification program using nested if-elif-else statements. Analyze the correctness and clarity of the conditions. Then rewrite the same classification using an alternative conditional structure such as simplified conditions or a dictionary-based approach.

## CODE:

```
def classify_age_alternative(age):
    """
    Classifies an age into categories (Child, Teenager, Adult, Senior)
    using an alternative conditional structure (simple if-elif-else).

    Args:
        age (int): The age to classify.

    Returns:
        str: The age category or an error message if the age is invalid.
    """
    if age < 0:
        return 'Invalid age: Age cannot be negative.'
    elif age <= 12:
        return 'Child'
    elif age <= 19:
        return 'Teenager'
    elif age <= 64:
        return 'Adult'
    else:
        return 'Senior'

# Test cases as requested
test_ages = [-5, 7, 15, 30, 70]
print("Classifying ages using alternative conditional structure function:")
for age in test_ages:
    category = classify_age_alternative(age)
    print(f"Age {age}: {category}")
```

## OUTPUT:

```
•• Classifying ages using alternative conditional structure function:
   Age -5: Invalid age: Age cannot be negative.
   Age 7: Child
   Age 15: Teenager
   Age 30: Adult
   Age 70: Senior
```

## **EXPLANATION:**

- A function is defined that accepts age as an input value.
- The program checks the age using if-elif-else conditions.
- If the age is less than a certain value, it is classified as a child.
- If the age falls in the next range, it is classified as a teenager.
- If the age is in the adult range, it is classified as an adult.
- If the age is above the adult range, it is classified as a senior.
- Only one condition is executed because once a condition is true, the remaining checks are skipped.
- The function returns or prints the appropriate age group.

## **Task 4: For and While Loops (Sum of First n Numbers)**

## **PROMPT:**

**J** Generate a Python function to compute the sum of the first n natural numbers using a for loop. Analyze the generated code and verify the output with sample inputs. Then suggest an alternative implementation using a while loop or a mathematical formula and compare both approaches.

## CODE:

```
def sum_natural_numbers_for_loop(n):  
    """  
    Computes the sum of the first n natural numbers using a for loop.  
  
    Args:  
        n (int): The number of natural numbers to sum.  
  
    Returns:  
        int: The sum of the first n natural numbers.  
    """  
    if n < 0:  
        return "Error: n must be a non-negative integer."  
    total_sum = 0  
    for i in range(1, n + 1):  
        total_sum += i  
    return total_sum  
  
# Test cases to verify the function  
print(f"Sum of first 5 natural numbers: {sum_natural_numbers_for_loop(5)}")  
print(f"Sum of first 0 natural numbers: {sum_natural_numbers_for_loop(0)}")  
print(f"Sum of first 1 natural numbers: {sum_natural_numbers_for_loop(1)}")  
print(f"Sum of first 10 natural numbers: {sum_natural_numbers_for_loop(10)}")  
print(f"Sum of first -3 natural numbers: {sum_natural_numbers_for_loop(-3)}")
```

# Expected: 15  
# Expected: 0  
# Expected: 1  
# Expected: 55  
# Expected: Error message

## OUTPUT:

```
Sum of first 5 natural numbers: 15  
Sum of first 0 natural numbers: 0  
Sum of first 1 natural numbers: 1  
Sum of first 10 natural numbers: 55  
Sum of first -3 natural numbers: Error: n must be a non-negative integer.
```

## **EXPLANATION**

- A function named `sum_to_n()` is defined to calculate the sum of natural numbers.
- The function takes an integer `n` as input.
- A variable (for example, `total`) is initialized to 0 to store the sum.
- A for loop runs from 1 to `n`.
- In each iteration, the current number is added to `total`.
- After the loop finishes, `total` contains the sum of the first `n` natural numbers.
- The function returns or prints the final sum.
- The output is displayed for the given sample input.

## **Task 5: Classes (Bank Account Class)**

### **PROMPT:**

**J** Create a Python class named `BankAccount` with attributes for account holder name and balance. Add methods `deposit()`, `withdraw()`, and `check_balance()`. Show deposit and withdrawal operations with updated balance.



## CODE:

```
class BankAccount:
    def __init__(self, account_holder, initial_balance=0.0):
        self.account_holder = account_holder
        if initial_balance < 0:
            print("Initial balance cannot be negative. Setting to 0.")
            self.balance = 0.0
        else:
            self.balance = initial_balance
        print(f"Account for {self.account_holder} created with initial balance: ${self.balance:.2f}")

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount:.2f}. New balance: ${self.balance:.2f}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif amount > self.balance:
            print("Insufficient funds. Current balance: ${self.balance:.2f}")
        else:
            self.balance -= amount
            print(f"Withdrew ${amount:.2f}. New balance: ${self.balance:.2f}")

    def check_balance(self):
        print(f"Current balance for {self.account_holder}: ${self.balance:.2f}")
```

## OUTPUT:

```
# Create a BankAccount object
my_account = BankAccount("John Doe", 1000.00)

# Check initial balance
my_account.check_balance()

# Perform a deposit
my_account.deposit(500.50)

# Perform a withdrawal
my_account.withdraw(200.25)

# Check balance after operations
my_account.check_balance()

# Attempt an invalid withdrawal (insufficient funds)
my_account.withdraw(1500.00)

# Attempt an invalid deposit (negative amount)
my_account.deposit(-100.00)

# Check final balance
my_account.check_balance()

... Account for John Doe created with initial balance: $1000.00
Current balance for John Doe: $1000.00
Deposited $500.50. New balance: $1500.50
Withdrew $200.25. New balance: $1300.25
Current balance for John Doe: $1300.25
Insufficient funds. Current balance: ${self.balance:.2f}
Deposit amount must be positive.
Current balance for John Doe: $1300.25
```

## EXPLANATION:

- A class named BankAccount is created to represent a bank account.
- The constructor (`__init__`) initializes the account holder's name and the starting balance.
- The `deposit()` method is used to add a given amount to the current balance.
- The `withdraw()` method subtracts an amount from the balance after checking if sufficient funds are available.
- If the balance is not enough, the withdrawal is not allowed and a message is shown.
- The `check_balance()` method displays the current account balance.
- An object of the BankAccount class is created to perform banking operations.
- Deposit and withdrawal methods are called, and the updated balance is shown after each operation

