ASSIGNMENT - 6.3

Shaik Sabina
2303A51303
Batch-05

TASK1

Prompt: Generate a Python Student class with attributes name, roll_number, and branch.
Include a constructor (__init__) and a display_details() method.
Create a sample object and display the student details.
Add clean and readable code.

Code:

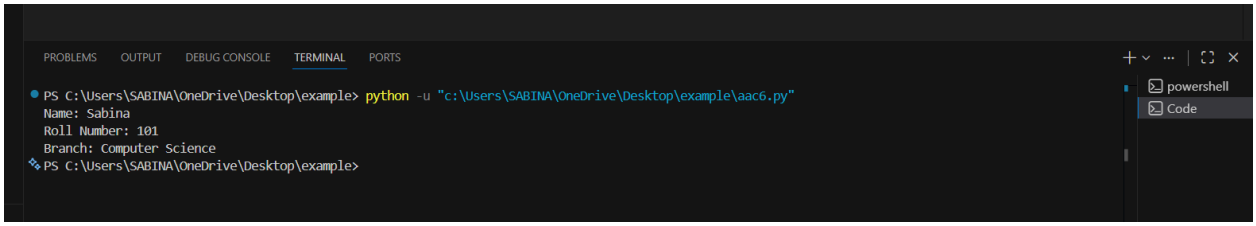
```
class Student:
    def __init__(self, name, roll_number, branch):
        self.name = name
        self.roll_number = roll_number
        self.branch = branch

    def display_details(self):
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Branch: {self.branch}")


# Create a sample object and display details
student = Student("Sabina", 101, "Computer Science")
student.display_details()
```

Code Explanation: This code defines a `Student` class to store student-related information.
The constructor initializes name, roll number, and branch when an object is created.
The `display_details()` method prints the stored data in a readable format.
An object is created to demonstrate how the class works.

Output:



Description:The AI-generated code correctly defines a Student class using a constructor to initialize attributes.
The `display_details()` method prints student information in a clear format.
Object creation and method invocation are correctly demonstrated.
The code is simple, readable, and suitable for beginner-level applications.

TASK-2

Prompt:Write a Python function to print the first 10 multiples of a number using a for loop.

Code:

```
class Student:
    def __init__(self, name, roll_number, branch):
        self.name = name
        self.roll_number = roll_number
        self.branch = branch

    def display_details(self):
        print("Name:", self.name)
        print("Roll Number:", self.roll_number)
        print("Branch:", self.branch)


# Object creation (ALWAYS outside the class)
student = Student("Sabina", 101, "Computer Science")
student.display_details()
def print_multiples(num):
    for i in range(1, 11):
        print(f"{num} × {i} = {num * i}")

print_multiples(5)
```

```
print_multiples(7)
```

Code Explanation:

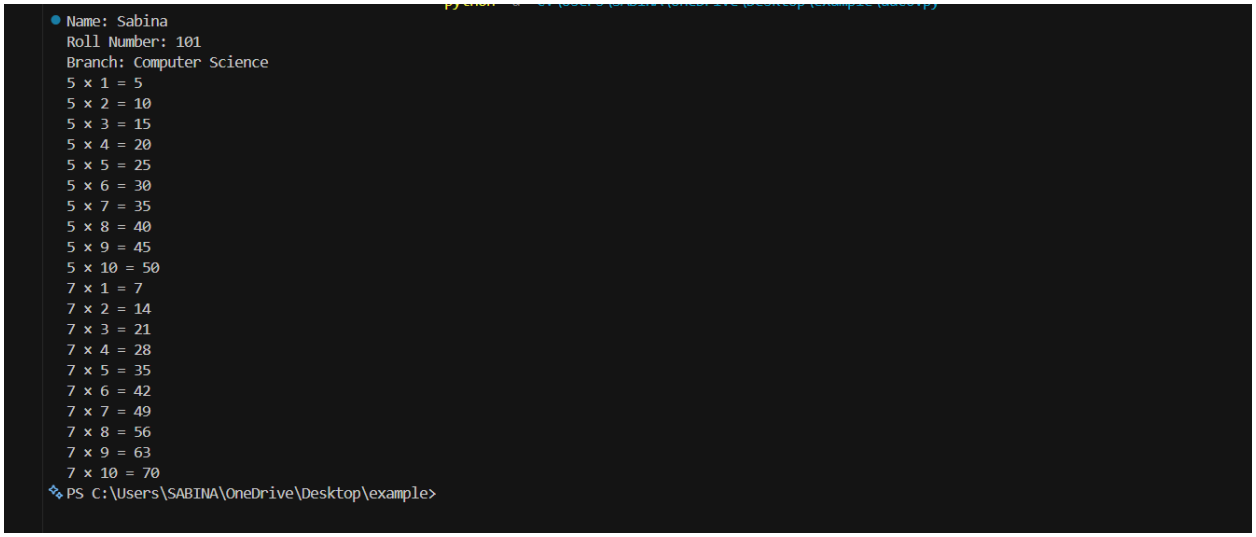
Student Class Explanation

The `Student` class is used to store student details such as name, roll number, and branch.
The constructor initializes these values when an object is created.
The `display_details()` method prints the stored information.
The object is created outside the class to avoid execution errors.

Multiples Function Explanation

The `print_multiples()` function uses a loop to print the first 10 multiples of a given number.The loop runs from 1 to 10 and multiplies the number with the counter.
Formatted output improves readability.
This function can be reused for any input number.

Output:



Description:

The `for` loop implementation is concise and ideal when the number of iterations is known.

The `while` loop provides more control over loop variables but requires manual incrementing.

Both approaches correctly generate the first 10 multiples.

The AI-generated logic is correct and easy to understand

TASK-3

Prompt:Create a Python function using nested if-elif-else statements to classify age into child, teenager, adult, and senior.Provide a simplified alternative implementation.

Code

```
def classify_age(age):
    if age < 0:
        return "Invalid age"
    elif age < 13:
        return "Child"
    elif age < 18:
        return "Teenager"
    elif age < 60:
        return "Adult"
    else:
        return "Senior"


# Test the function
```

```
print(classify_age(5))

print(classify_age(15))

print(classify_age(30))

print(classify_age(70))
```

Code Explanation:The function checks age using ordered conditional statements.

Each condition represents a specific age range.

Only one condition executes due to the if-elif-else structure.

This ensures accurate and clear age classification.

Alternative Method Code

```
# Simplified alternative using a dictionary and conditional logic
def classify_age_simplified(age):
    age_groups = [(13, "Child"), (18, "Teenager"), (60, "Adult")]
    for threshold, category in age_groups:
        if age < threshold:
            return category
    return "Senior" if age >= 0 else "Invalid age"

# Test simplified version
print(classify_age_simplified(5))
print(classify_age_simplified(15))
print(classify_age_simplified(30))
print(classify_age_simplified(70))t=int(input())

```

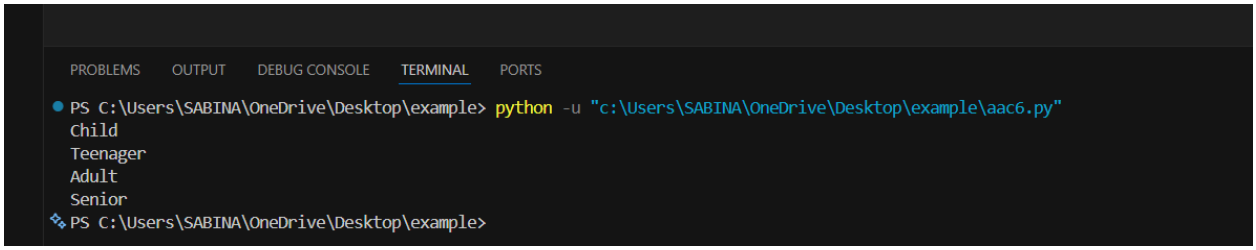
Code Explanation:This version uses simplified conditions without explicit ranges.

The function checks higher age limits first.

Once a condition is met, the function returns immediately.

It improves readability while maintaining correct results.

Output:



Description:

The nested if-elif-else structure clearly defines mutually exclusive age groups.

Conditions are ordered logically to avoid incorrect classification.

The alternative approach improves readability using structured conditions.

Both implementations correctly classify age values.

TASK-4

Prompt: Write a Python function sum_to_n() to calculate the sum of first n natural numbers using a for loop.

Also suggest an alternative using a while loop or mathematical formula.

Code

```
# Function to calculate sum of first n natural numbers using for loop

def sum_to_n(n):

    total = 0

    for i in range(1, n + 1):

        total += i

    return total
```

```
# Test the function

print(sum_to_n(5))     # Output: 15

print(sum_to_n(10))    # Output: 55
```

Code explanation:

The function initializes a variable to store the sum.

A for loop adds each number from 1 to n.

The final sum is returned after the loop completes.

This method is easy to understand and widely used.

Alternative Method

```
# Alternative using while loop

def sum_to_n_while(n):

    total = 0

    i = 1

    while i <= n:

        total += i

        i += 1

    return total



# Alternative using mathematical formula (Gauss formula)

def sum_to_n_formula(n):

    return n * (n + 1) // 2
```

```
# Test alternatives

print(sum_to_n_while(5))       # Output: 15

print(sum_to_n_formula(10))    # Output: 55
```

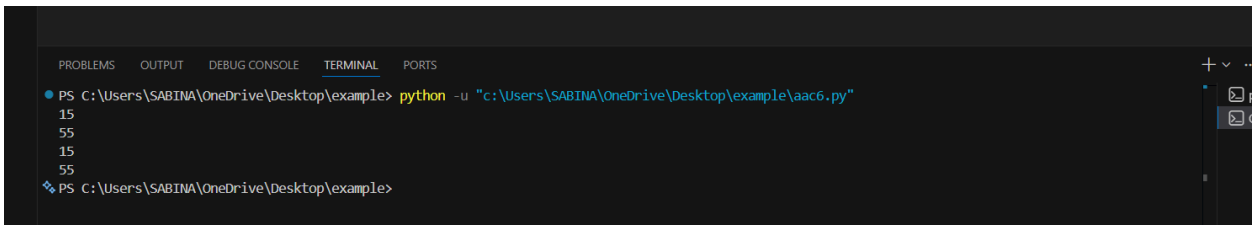
Code explanation:This method uses a mathematical formula to calculate the sum.

It avoids looping and executes in constant time.

The formula is efficient for large values of n.

This approach is optimal in terms of performance.

Output



Description:The for-loop method is straightforward and easy to understand.

The while-loop version offers similar logic with explicit control.

The mathematical formula is the most efficient with constant time complexity.

All approaches produce correct results for valid inputs.

TASK-5

Prompt:Generate a Python BankAccount class with deposit(), withdraw(), and check_balance() methods.

Include comments and demonstrate usage with sample operations.

Code

aac6.py > BankAccount > withdraw

```python
# BankAccount class
class BankAccount:
    """A simple bank account class with deposit, withdraw, and balance checking."""

    def __init__(self, account_holder, initial_balance=0):
        """Initialize account with holder name and optional initial balance."""
        self.account_holder = account_holder
        self.balance = initial_balance

    def deposit(self, amount):
        """Deposit money into the account."""
        if amount <= 0:
            print("Deposit amount must be positive.")
            return False
        self.balance += amount
        print(f"Deposited ${amount}. New balance: ${self.balance}")
        return True

    def withdraw(self, amount):
        """Withdraw money from the account."""
        if amount <= 0:
            print("Withdrawal amount must be positive.")
            return False
        if amount > self.balance:
            print(f"Insufficient funds. Current balance: ${self.balance}")
            return False
        self.balance -= amount
        print(f"Withdrew ${amount}. New balance: ${self.balance}")
        return True

    def check_balance(self):
        """Check current account balance."""
        print(f"Account holder: {self.account_holder}")
        print(f"Current balance: ${self.balance}")
        return self.balance

# Demonstrate usage
account = BankAccount("Alice", 100)
account.check_balance()
account.deposit(50)
account.withdraw(30)
account.withdraw(200)  # This will fail
account.check_balance()

```

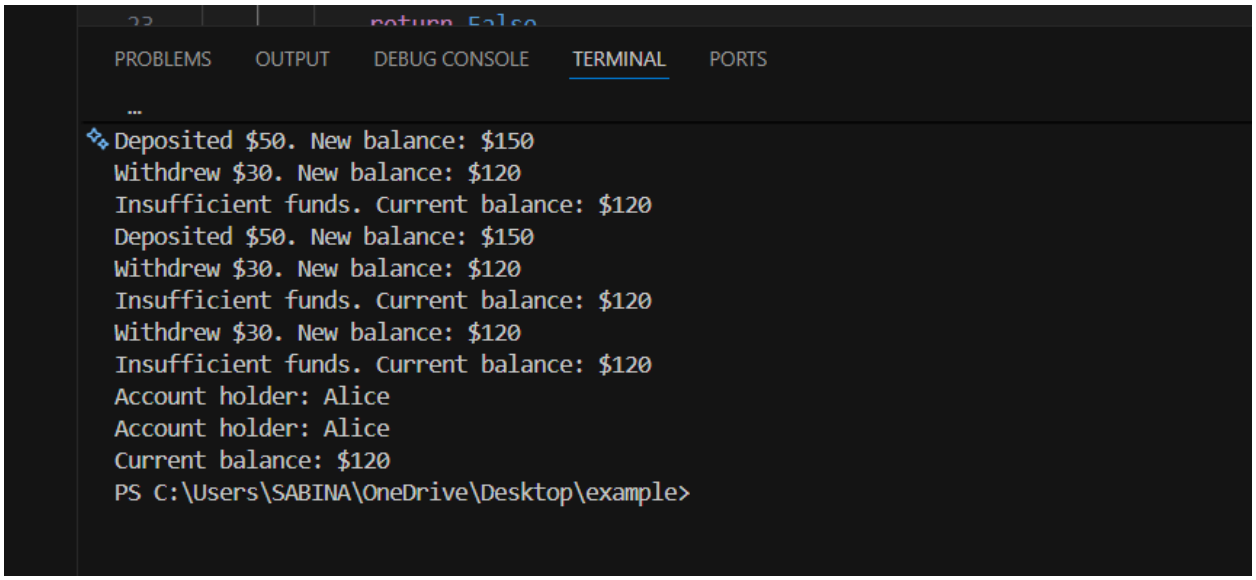
Code explanation:This class represents a simple bank account system.

The constructor initializes account holder name and balance.

Deposit and withdraw methods update the balance safely.

The `check_balance()` method displays the current balance.

Output



Description:The BankAccount class is well-structured with clearly defined methods.
Balance validation is correctly handled during withdrawal.
Comments improve code readability and understanding.
The AI-generated solution is suitable for a basic banking simulation.