

« AI ASSISTED CODING

Task Description#1 (Classes)

- Use AI to complete a Student class with attributes and a method.
- Check output
- Analyze the code generated by AI tool

Expected Output#1

- Class with constructor and display_details() method

Prompt: Design and implement a Python class named Student. This class should have an initialization method that accepts and stores the student's name, roll number, and grade as attributes. Additionally, include a method within the class that, when called, prints out all the stored details of the student in a clear format.

Code :

```
▶ class Student:
    def __init__(self, name, roll_number, grade):
        """
        Initializes a Student object with name, roll number, and grade.

        Args:
            name (str): The name of the student.
            roll_number (str): The roll number of the student.
            grade (str): The grade of the student.
        """
        self.name = name
        self.roll_number = roll_number
        self.grade = grade

    def display_details(self):
        """
        Prints the details of the student in a clear format.
        """
        print("Student Details:")
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Grade: {self.grade}")
```

```
▶ # Get student details from user input
student_name = input("Enter student name: ")
student_roll_number = input("Enter student roll number: ")
student_grade = input("Enter student grade: ")

# Create a Student object
student1 = Student(student_name, student_roll_number, student_grade)

# Display student details
student1.display_details()
```

```
➞ Enter student name: Akhila
Enter student roll number: 2403a52010
Enter student grade: A
Student Details:
Name: Akhila
Roll Number: 2403a52010
Grade: A
```

Analysis of code :

The code defines a Python class called Student.

- **__init__(self, name, roll_number, grade):** This is the constructor of the class. It's called when you create a new Student object. It takes the student's name, roll_number, and grade as arguments and stores them as attributes of the object using self.
- **display_details(self):** This is a method within the Student class. When called on a Student object, it prints out the stored name, roll_number, and grade in a formatted way.

The code then demonstrates how to use this class:

1. It prompts the user to enter the student's name, roll number, and grade using the **input()** function.
2. It creates a new **Student** object named **student1**, passing the user's input as arguments to the constructor.
3. It calls the **display_details()** method on the **student1** object to print the entered information.

Observation :

- **Object-Oriented Structure:** The code effectively uses object-oriented programming principles by defining a class to represent a real-world entity (a student). This makes the code more organized and reusable.
- **Encapsulation:** The Student class encapsulates the data (name, roll number, grade) and the behavior (display_details) related to a student within a single unit.
- **Readability:** The method and variable names are clear and descriptive, making the code easy to understand.

- **Basic Functionality:** The code provides basic functionality for storing and displaying student information. It's a good starting point for a more complex student management system.
- **User Interaction:** It demonstrates how to interact with the user to get input and use it to create objects.

Task Description#2 (Loops)

- Prompt AI to complete a function that prints the first 10 multiples of a number using a loop.
- Analyze the generated code
- Ask AI to generate code using other controlled looping

Expected Output#2

- Correct loop-based implementation

Prompt: Write a Python function that takes a number as input and prints its first 10 multiples using a loop. After that, explain how the code works. Then, show me how to do the same thing but using a different type of loop.

Code :

```

▶ def print_multiples_while(number):
    """
    Prints the first 10 multiples of a given number using a while loop.

    Args:
        number: The number for which to print multiples.
    """
    print(f"Multiples of {number}:")
    i = 1
    while i <= 10:
        print(number * i)
        i += 1

# Get number from user input
try:
    user_number = int(input("Enter a number to see its first 10 multiples: "))
    print_multiples_while(user_number)
except ValueError:
    print("Invalid input. Please enter an integer.")

```

Output :

```
▶ # Get number from user input
try:
    user_number = int(input("Enter a number to see its first 10 multiples: "))
    print_multiples_while(user_number)
except ValueError:
    print("Invalid input. Please enter an integer.")
```

```
➡ Enter a number to see its first 10 multiples: 8
Multiples of 8:
8
16
24
32
40
48
56
64
72
80
```

Explanation of code:

This function, `print_multiples_for(number)`, does the following:

1. **`def print_multiples_for(number):`**: This line defines a function named `print_multiples_for` that takes one argument, `number`.
2. **`""" Docstring """`**: This is a docstring, which explains what the function does. It's good practice to include these for clarity.
3. **`print(f"Multiples of {number}:")`**: This line prints a header indicating which number's multiples are being displayed.
4. **`for i in range(1, 11):`**: This is a for loop.
 - a. `range(1, 11)` generates a sequence of numbers starting from 1 and going up to (but not including) 11. So, the values of `i` will be 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.
 - b. The loop will iterate 10 times, with `i` taking on each of these values in turn.
5. **`print(number * i)`**: Inside the loop, this line calculates the product of the input number and the current value of `i`. This gives you the multiple of the number. The result is then printed on a new line.

Observation :

- **while Loop Implementation:** This code achieves the same outcome as the previous for loop example, but it uses a while loop. This demonstrates that you can often accomplish the same task with different types of loops.
- **Manual Counter Management:** Unlike the for loop with `range()`, the while loop requires you to explicitly initialize a counter variable (`i = 1`) and increment it within the loop (`i += 1`). You also need a condition (`while i <= 10:`) to control when the loop stops.

- **User Input with Error Handling:** The code includes a section to get input from the user using input().
- **Function Reusability:** The print_multiples_while function is defined separately, making it reusable with different numbers.

Task Description#3 (Conditional Statements)

- Ask AI to write nested if-elif-else conditionals to classify age groups.
- Analyze the generated code
- Ask AI to generate code using other conditional statements

Expected Output#3

- Age classification function with appropriate conditions and with explanation

Prompt : Create a Python function that takes an age as input and uses nested if-elif-else statements to determine and print the age group (e.g., Child, Teenager, Adult, Senior). After that, please explain the code. Then, show me how to do the age classification using a different approach with conditional statements.

Code:

```

▶ def classify_age_nested_if(age):
    """
    Classifies age into groups using nested if-elif-else statements.

    Args:
        age (int): The age to classify.
    """
    if age is not None:
        if age < 0:
            print("Invalid age: Age cannot be negative.")
        elif age <= 12:
            print("Age group: Child")
        else:
            if age <= 19:
                print("Age group: Teenager")
            else:
                if age <= 64:
                    print("Age group: Adult")
                else:
                    print("Age group: Senior")
    else:
        print("Invalid input: Age cannot be None.")

```

```
# Get age from user input
try:
    user_age = int(input("Enter your age: "))
    classify_age_nested_if(user_age)
except ValueError:
    print("Invalid input. Please enter a valid integer for age.")
```

➞ Enter your age: 19
Age group: Teenager

Explanation of code:

This function, `classify_age_nested_if(age)`, does the following:

1. **def `classify_age_nested_if(age)`:** This line defines a function named `classify_age_nested_if` that takes one argument, `age`.
2. **""" Docstring """:** This is a docstring explaining the function's purpose.
3. **if `age is not None`:** This is the outermost condition, checking if the input age is not `None`. This is a basic check for a valid input.
4. **if `age < 0`:** If age is not `None`, this checks if the age is negative, indicating invalid input.
5. **elif `age <= 12`:** If the age is not negative, this checks if it's less than or equal to 12, classifying it as "Child".
6. **else:** If the age is greater than 12, the code enters a nested else block.
7. **if `age <= 19`:** Inside the first nested else, this checks if the age is less than or equal to 19, classifying it as "Teenager".
8. **else:** If the age is greater than 19, the code enters another nested else block.
9. **if `age <= 64`:** Inside the second nested else, this checks if the age is less than or equal to 64, classifying it as "Adult".
10. **else:** If the age is greater than 64, it's classified as "Senior".
11. **else: (corresponding to the outermost if):** If the initial check if age is not `None` is false, this else block is executed, indicating that the input age was `None`.

Observation :

- **Nested Conditional Structure:** The code uses `if-elif-else` statements nested within each other to handle different age ranges. This approach creates a clear hierarchy for the age classifications.
- **Specific Age Grouping:** The conditions are set up to define specific age brackets: negative ages (invalid), child (0-12), teenager (13-19), adult (20-64), and senior (65+).
- **Handling Invalid Input (Negative Age and None):** The code includes checks for negative age and whether the input age is `None`, demonstrating basic input validation.

- **User Input and Error Handling:** It incorporates getting age from the user via `input()` and uses a `try...except ValueError` block to gracefully handle cases where the user enters non-integer input.

Task Description#4 (For and While loops)

- Generate a `sum_to_n()` function to calculate sum of first n numbers
- Analyze the generated code
- Get suggestions from AI with other controlled looping

Prompt : Write a Python function called `sum_to_n` that takes an integer `n` as input and calculates the sum of all integers from 1 to `n` using a loop. After generating the code, please analyze it. Then, suggest how I could achieve the same sum using a different type of loop.

Code :

```
def sum_to_n(n):
    """
    Calculates the sum of integers from 1 to n using a for loop.

    Args:
        n (int): The upper limit of the range for summation.

    Returns:
        int: The sum of integers from 1 to n.
    """
    if n < 0:
        return "Input must be a non-negative integer." # Handle negative input

    total_sum = 0
    for i in range(1, n + 1):
        total_sum += i
    return total_sum

# Get input from user
try:
    user_n = int(input("Enter a positive integer to sum up to: "))
    result = sum_to_n(user_n)
    print(f"The sum of integers from 1 to {user_n} is: {result}")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

```
➞ Enter a positive integer to sum up to: 20
The sum of integers from 1 to 20 is: 210
```

Explanation of code :

This function, `sum_to_n(n)`, does the following:

1. **def sum_to_n(n)::** Defines a function named `sum_to_n` that takes one argument, `n`.
2. **if `n < 0`::** This is an input validation step. It checks if the input `n` is negative. If it is, the function returns a specific string indicating that the input must be a non-negative integer.
3. **total_sum = 0:** Initializes a variable `total_sum` to 0. This variable will store the cumulative sum.
4. **for `i` in range(1, n + 1)::** This is a for loop that iterates through a sequence of numbers.
 - a. `range(1, n + 1)` generates a sequence of integers starting from 1 and going up to (and including) `n`. For example, if `n` is 5, `range(1, 6)` will generate 1, 2, 3, 4, 5.
 - b. The loop will execute `n` times, with `i` taking on each value in the generated sequence.
5. **total_sum += i:** Inside the loop, in each iteration, the current value of `i` is added to `total_sum`. This accumulates the sum of the numbers from 1 to `n`.
6. **return total_sum:** After the loop finishes, the function returns the final calculated `total_sum`.

Observation:

- **Function Definition:** The code defines a clear function `sum_to_n(n)` that encapsulates the logic for calculating the sum.
- **for Loop for Summation:** It uses a for loop with `range(1, n + 1)` to iterate through the numbers from 1 to `n`, which is a standard and efficient way to handle such summations in Python.
- **Accumulator Variable:** The `total_sum = 0` variable acts as an accumulator, adding each number in the sequence to itself during the loop.
- **Input Validation:** The `if n < 0:` check is a good practice for handling invalid negative input, preventing unexpected results.

Task Description#5 (Class)

- Use AI to build a `BankAccount` class with `deposit`, `withdraw`, and `balance` methods.
- Analyze the generated code
- Add comments and explain code

Expected Output#5

- Python code with explanation

Prompt: Create a Python class named `BankAccount`. It should have methods for `deposit`, `withdraw`, and checking the balance. After you generate the code, please analyze it, add comments to explain each part, and provide a general explanation of how the class works.

Code:


```

▶ class BankAccount:
    """
    Represents a simple bank account with deposit, withdraw, and balance features.
    """
    def __init__(self, account_number, initial_balance=0):
        """
        Initializes a new BankAccount.

        Args:
            account_number (str): The unique account number.
            initial_balance (float): The initial balance of the account (default is 0).
        """
        self.account_number = account_number
        self.balance = initial_balance

    def deposit(self, amount):
        """
        Deposits funds into the account.

        Args:
            amount (float): The amount to deposit.

        Returns:
            str: A message indicating the success or failure of the deposit.
        """
        if amount > 0:
            self.balance += amount
            return f"Deposit of ${amount:.2f} successful. New balance: ${self.balance:.2f}"
        else:
            return "Invalid deposit amount. Amount must be positive."

    def withdraw(self, amount):
        """
        Withdraws funds from the account.

        Args:
            amount (float): The amount to withdraw.

        Returns:
            str: A message indicating the success or failure of the withdrawal.
        """
        if amount > 0:
            if self.balance >= amount:
                self.balance -= amount
                return f"Withdrawal of ${amount:.2f} successful. New balance: ${self.balance:.2f}"
            else:
                return "Insufficient funds."
        else:
            return "Invalid withdrawal amount. Amount must be positive."

```



```
def get_balance(self):
    """
    Gets the current balance of the account.

    Returns:
        float: The current balance.
    """
    return self.balance

# Example Usage with User Input:
account_num = input("Enter account number: ")
initial_bal = float(input("Enter initial balance: "))

my_account = BankAccount(account_num, initial_bal)

while True:
    print("\nChoose an action:")
    print("1. Deposit")
    print("2. Withdraw")
    print("3. Check Balance")
    print("4. Exit")

    choice = input("Enter choice (1-4): ")

    choice = input("Enter choice (1-4): ")

    if choice == '1':
        try:
            deposit_amount = float(input("Enter amount to deposit: "))
            print(my_account.deposit(deposit_amount))
        except ValueError:
            print("Invalid input. Please enter a valid number for the amount.")
    elif choice == '2':
        try:
            withdraw_amount = float(input("Enter amount to withdraw: "))
            print(my_account.withdraw(withdraw_amount))
        except ValueError:
            print("Invalid input. Please enter a valid number for the amount.")
    elif choice == '3':
        print(f"Current balance: ${my_account.get_balance():.2f}")
    elif choice == '4':
        print("Exiting.")
        break
    else:
        print("Invalid choice. Please enter a number between 1 and 4.")
```

Output:

```
Enter account number: 240352010
Enter initial balance: 5000
→
Choose an action:
1. Deposit
2. Withdraw
3. Check Balance
4. Exit
Enter choice (1-4): 2
Enter amount to withdraw: 2000
Withdrawal of $2000.00 successful. New balance: $3000.00

Choose an action:
1. Deposit
2. Withdraw
3. Check Balance
4. Exit
Enter choice (1-4): 3
Current balance: $3000.00

Choose an action:
1. Deposit
2. Withdraw
3. Check Balance
4. Exit
Enter choice (1-4): 4
Exiting.
```

Code Analysis with Comments:

- **class BankAccount::** This line declares the beginning of a new class named BankAccount.
- **""" Docstring """**: This is a docstring that provides a brief description of the class's purpose.
- **def __init__(self, account_number, initial_balance=0)::** This is the constructor method. It's automatically called when you create a new BankAccount object.
 - self: Refers to the instance of the class being created.
 - account_number: An argument to pass the unique account number when creating an object.
 - initial_balance=0: An optional argument for the starting balance, defaulting to 0 if not provided.
 - self.account_number = account_number: Stores the provided account number as an attribute of the object.
 - self.balance = initial_balance: Stores the initial balance as an attribute of the object.
- **def deposit(self, amount)::** This method handles depositing money.
 - self: Refers to the instance of the class.

- amount: The amount of money to deposit.
- if amount > 0:: Checks if the deposit amount is positive.
- self.balance += amount: Adds the positive amount to the current balance.
- return f"...": Returns a confirmation message with the new balance.
- else:: If the amount is not positive, returns an error message.
- **def withdraw(self, amount)::** This method handles withdrawing money.
 - self: Refers to the instance of the class.
 - amount: The amount of money to withdraw.
 - if amount > 0:: Checks if the withdrawal amount is positive.
 - if self.balance >= amount:: Checks if there are sufficient funds in the account.
 - self.balance -= amount: Subtracts the withdrawal amount from the balance.
 - return f"...": Returns a confirmation message with the new balance.
 - else:: If insufficient funds, returns an error message.
 - else:: If the amount is not positive, returns an error message.
- **def get_balance(self)::** This method returns the current balance.
 - self: Refers to the instance of the class.
 - return self.balance: Returns the value of the balance attribute.

Observation :

- **Object-Oriented Design:** The code effectively uses a class to model a real-world concept (a bank account), demonstrating object-oriented principles like encapsulation.
- **Methods for Operations:** The class includes well-defined methods (deposit, withdraw, get_balance) that represent the actions you can perform on a bank account.
- **State Management:** The balance attribute within each BankAccount object maintains the state of that specific account, independent of other accounts.
- **Input Validation:** The deposit and withdraw methods include checks to ensure the transaction amount is positive. The withdraw method also checks for sufficient funds.
- **User Interaction Loop:** The example usage includes a while loop that provides a menu-driven interface for the user to interact with the bank account object.