AI Assisted Coding

Lab Assignment 6.3

Name : V Sai Krishnachaithanya **Date : 04/02/2026**

Hall Ticket no : 2303A52154

<u>Task 1:</u> 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.

• The class should include attributes such as name, roll number, and branch.

• Add a method display_details() to print student information.

• Execute the code and verify the output.

• Analyze the code generated by the AI tool for correctness and clarity.

Expected Output #1

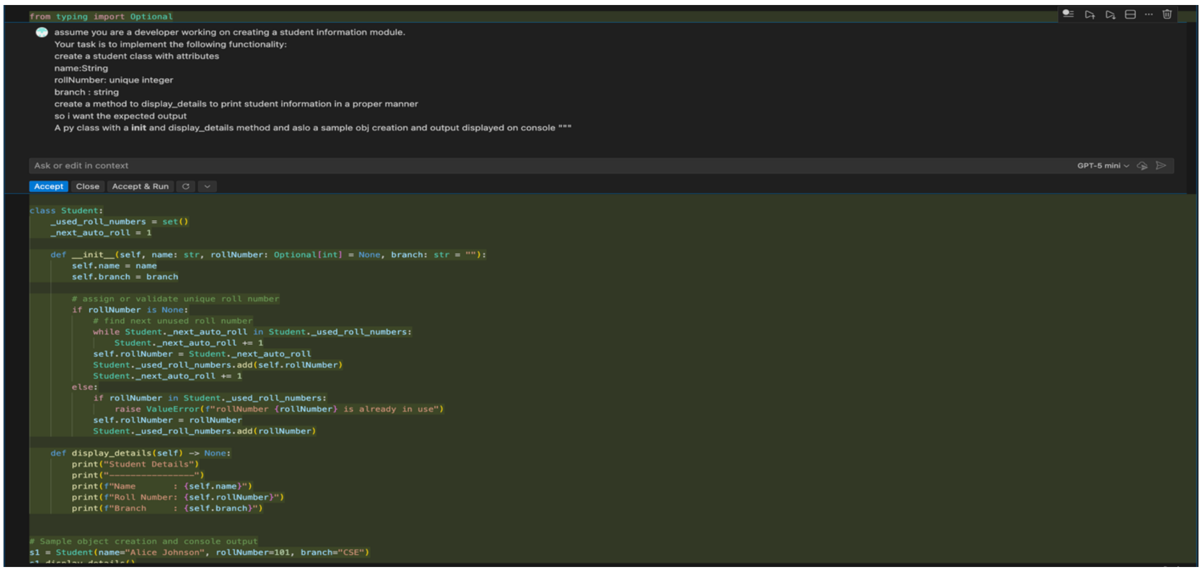
• A Python class with a constructor (__init__) and a display_details() method.

• Sample object creation and output displayed on the console.

• Brief analysis of AI-generated code.

Prompt :

assume you are a developer working on creating a student information module.create a student class with attributes name:String ,rollNumber: unique integer ,branch : string create a method to display_details to print student information in a proper manner so i want the expected outputA py class with a __init__ and display_details method and aslo a sample obj creation and output displayed on console

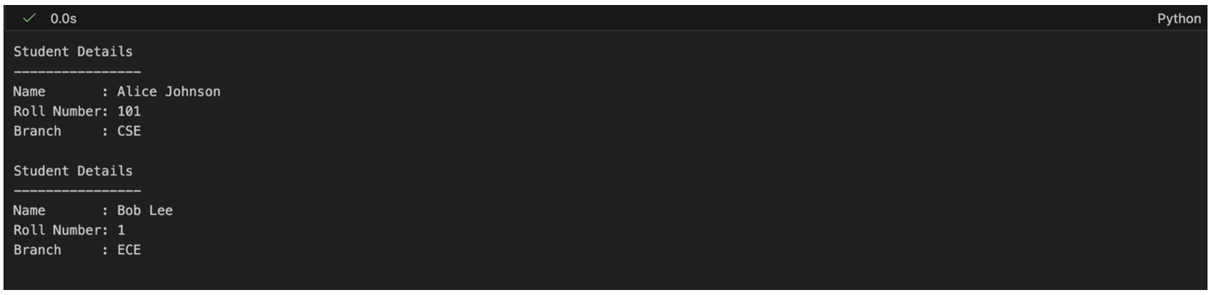


```
from typing import Optional
```

assume you are a developer working on creating a student information module.
Your task is to implement the following functionality:
create a student class with attributes
name:String
rollNumber: unique integer
branch : string
create a method to display_details to print student information in a proper manner
so i want the expected output
A py class with a **init** and display_details method and aslo a sample obj creation and output displayed on console """

```
class Student:
    _used_roll_numbers = set()
    _next_auto_roll = 1

    def __init__(self, name: str, rollNumber: Optional[int] = None, branch: str = ""):
        self.name = name
        self.branch = branch

        # assign or validate unique roll number
        if rollNumber is None:
            # find next unused roll number
            while Student._next_auto_roll in Student._used_roll_numbers:
                Student._next_auto_roll += 1
            self.rollNumber = Student._next_auto_roll
            Student._used_roll_numbers.add(self.rollNumber)
            Student._next_auto_roll += 1
        else:
            if rollNumber in Student._used_roll_numbers:
                raise ValueError(f"rollNumber {rollNumber} is already in use")
            self.rollNumber = rollNumber
            Student._used_roll_numbers.add(rollNumber)

    def display_details(self) -> None:
        print("Student Details")
        print("---------------")
        print(f"Name       : {self.name}")
        print(f"Roll Number: {self.rollNumber}")
        print(f"Branch     : {self.branch}")


# Sample object creation and console output
s1 = Student(name="Alice Johnson", rollNumber=101, branch="CSE")
```

Output :



```
✓ 0.0s                                                    Python
Student Details
---------------
Name       : Alice Johnson
Roll Number: 101
Branch     : CSE

Student Details
---------------
Name       : Bob Lee
Roll Number: 1
Branch     : ECE
```

Justification:

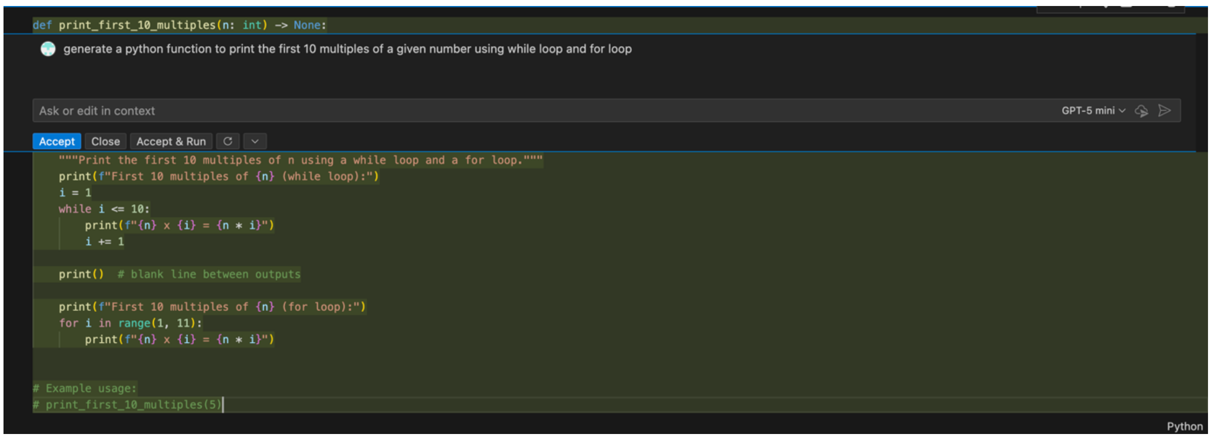
The AI-generated code correctly implements a Student class with automatic roll number assignment and validation.It uses a class variable `_used_roll_numbers` to keep track of all assigned roll numbers and ensures uniqueness.The `_next_auto_roll` variable is used to generate the next available roll number. The `display_details` method prints all student information in a formatted way. The example usage demonstrates both manual assignment and automatic assignment of roll numbers.

Task 2: Loops (Multiples of a Number)

Scenario
You are writing a utility function to display multiples of a given number.
Task
• Prompt the AI tool to generate a function that prints the first 10 multiples of a given number
using a loop.
• Analyze the generated loop logic.
• Ask the AI to generate the same functionality using another controlled looping structure (e.g.,
while instead of for).
Expected Output #2
• Correct loop-based Python implementation.
• Output showing the first 10 multiples of a number.
• Comparison and analysis of different looping approaches

Prompt :

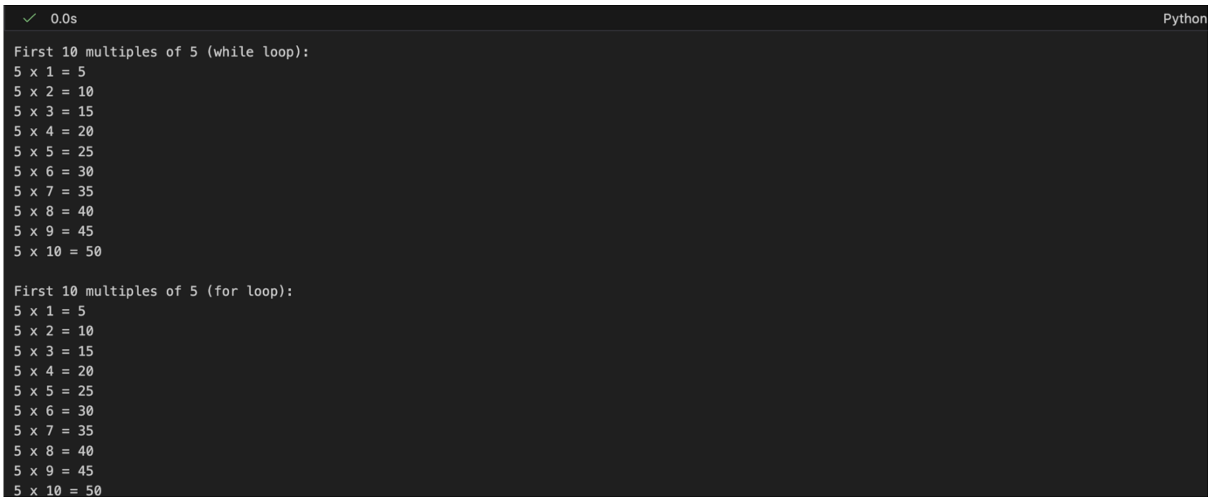
generate a python function to print the first 10 multiples of a given number using while loop and for loop



generate a python function to print the first 10 multiples of a given number using while loop and for loop

```python
def print_first_10_multiples(n: int) -> None:
    """Print the first 10 multiples of n using a while loop and a for loop."""
    print(f"First 10 multiples of {n} (while loop):")
    i = 1
    while i <= 10:
        print(f"{n} x {i} = {n * i}")
        i += 1

    print()  # blank line between outputs

    print(f"First 10 multiples of {n} (for loop):")
    for i in range(1, 11):
        print(f"{n} x {i} = {n * i}")

# Example usage:
# print_first_10_multiples(5)
```

Output :



```
First 10 multiples of 5 (while loop):
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50

First 10 multiples of 5 (for loop):
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Justification:

The while loop is used to demonstrate a simple counting mechanism, incrementing a counter until it reaches 10. The for loop is used to iterate over a fixed range of numbers (1 to 10) directly, which is more concise and readable for this specific task.

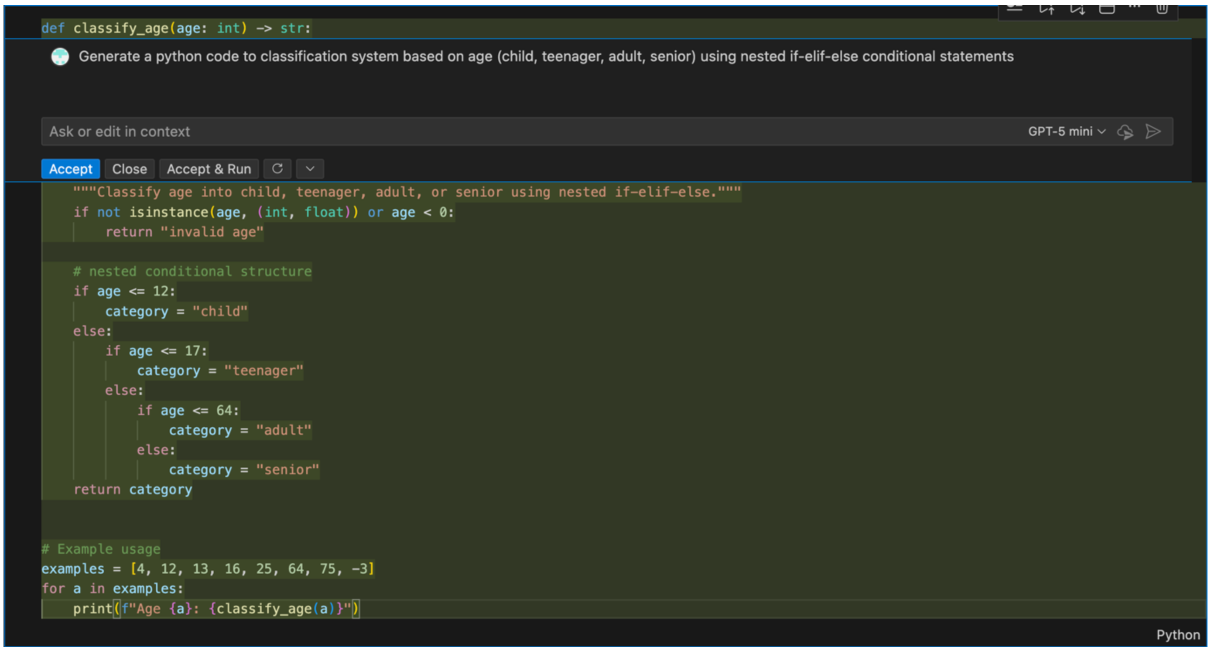
Task 3: Conditional Statements (Age Classification)

Scenario
You are building a basic classification system based on age.
Task
• Ask the AI tool to generate nested if-elif-else conditional statements to classify age groups(e.g., child, teenager, adult, senior).
• Analyze the generated conditions and logic.
• Ask the AI to generate the same classification using alternative conditional structures (e.g.,

simplified conditions or dictionary-based logic).
Expected Output #3
• A Python function that classifies age into appropriate groups.
• Clear and correct conditional logic.
• Explanation of how the conditions work

Prompt :

Generate a python code to classification system based on age (child, teenager, adult, senior) using nested if-elif-else conditional statements

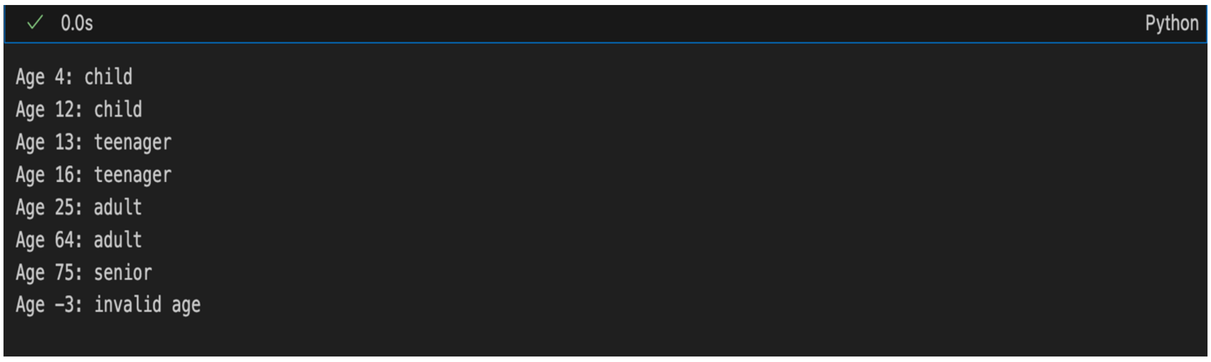


```python
def classify_age(age: int) -> str:
```

Generate a python code to classification system based on age (child, teenager, adult, senior) using nested if-elif-else conditional statements

```python
    """Classify age into child, teenager, adult, or senior using nested if-elif-else."""
    if not isinstance(age, (int, float)) or age < 0:
        return "invalid age"

    # nested conditional structure
    if age <= 12:
        category = "child"
    else:
        if age <= 17:
            category = "teenager"
        else:
            if age <= 64:
                category = "adult"
            else:
                category = "senior"
    return category


# Example usage
examples = [4, 12, 13, 16, 25, 64, 75, -3]
for a in examples:
    print(f"Age {a}: {classify_age(a)}")
```

Output :



```
Age 4: child
Age 12: child
Age 13: teenager
Age 16: teenager
Age 25: adult
Age 64: adult
Age 75: senior
Age -3: invalid age
```

Justification:

The nested if-elif-else structure evaluates age in a hierarchical manner:
1. If age is less than or equal to 12, it is classified as "child".
2. If age is greater than 12 but less than or equal to 17, it is classified as "teenager".

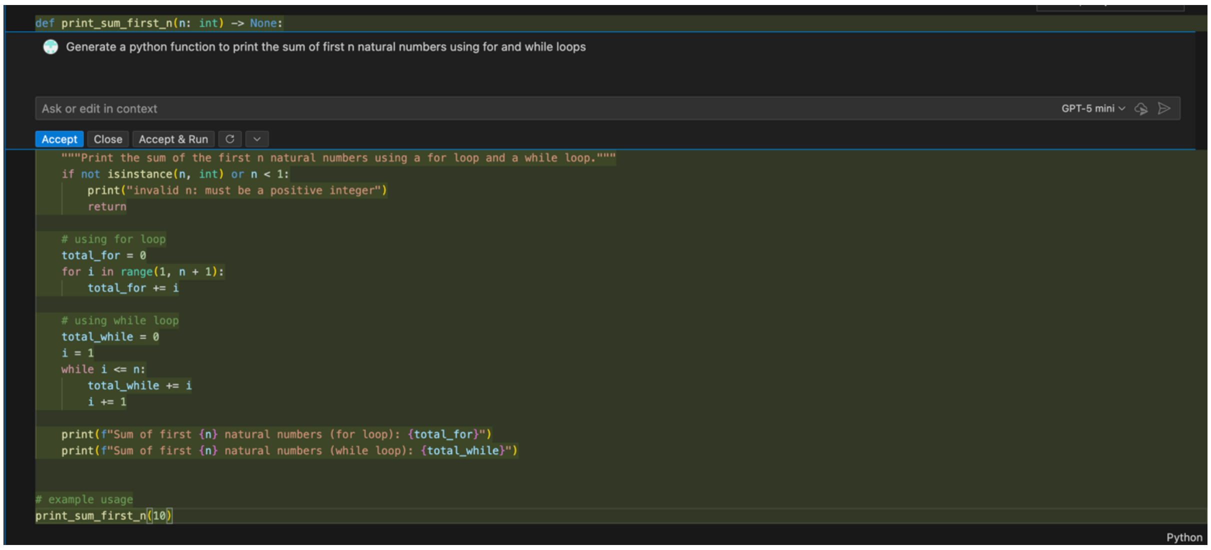
3. If age is greater than 17 but less than or equal to 64, it is classified as "adult".
4. If age is greater than 64, it is classified as "senior".
This structure ensures that only one category is assigned based on the age value.

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

Scenario
You need to calculate the sum of the first n natural numbers.
Task
• Use AI assistance to generate a sum_to_n() function using a for loop.
• Analyze the generated code.
• Ask the AI to suggest an alternative implementation using a while loop or a mathematical formula.
Expected Output #4
• Python function to compute the sum of first n numbers.
• Correct output for sample inputs.
• Explanation and comparison of different approaches.

Prompt :

Generate a python function to print the sum of first n natural numbers using for and while loops

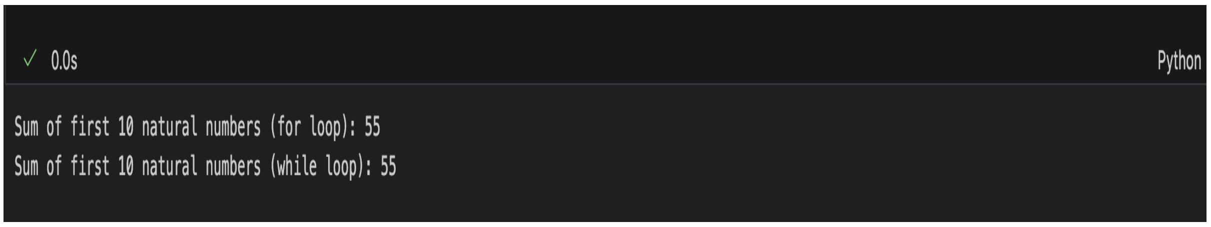


```
def print_sum_first_n(n: int) -> None:
```

Generate a python function to print the sum of first n natural numbers using for and while loops

```
    """Print the sum of the first n natural numbers using a for loop and a while loop."""
    if not isinstance(n, int) or n < 1:
        print("invalid n: must be a positive integer")
        return

    # using for loop
    total_for = 0
    for i in range(1, n + 1):
        total_for += i

    # using while loop
    total_while = 0
    i = 1
    while i <= n:
        total_while += i
        i += 1

    print(f"Sum of first {n} natural numbers (for loop): {total_for}")
    print(f"Sum of first {n} natural numbers (while loop): {total_while}")


# example usage
print_sum_first_n(10)
```

Output :



```
Sum of first 10 natural numbers (for loop): 55
Sum of first 10 natural numbers (while loop): 55
```

Justification:

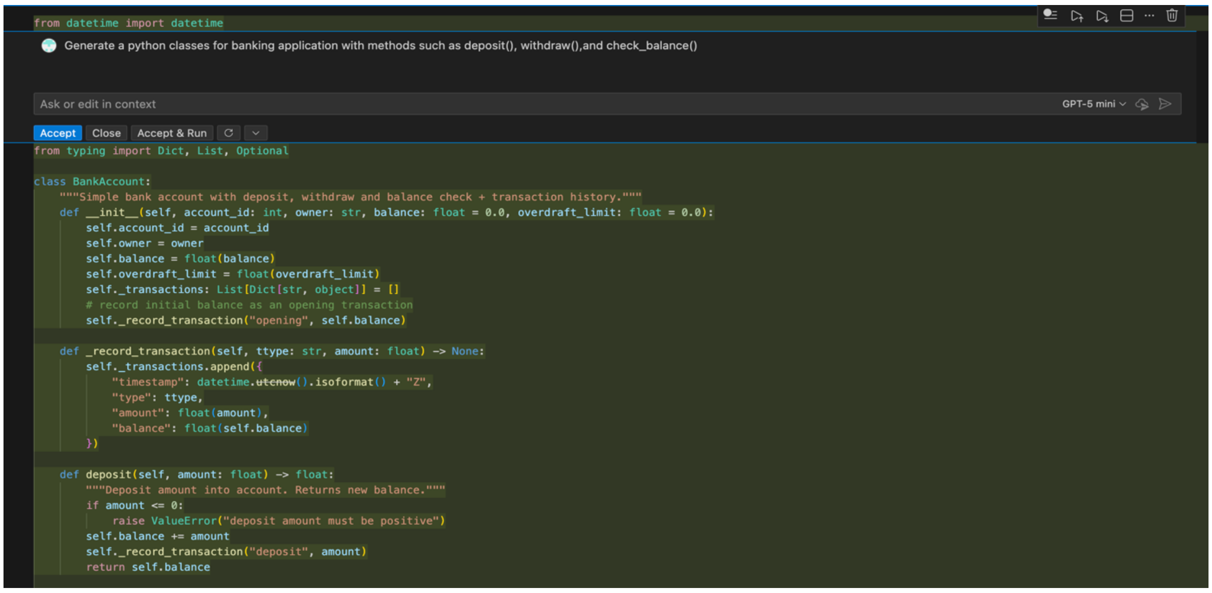
The for loop approach is more concise and easier to read when the number of iterations is known in advance . The while loop approach provides more flexibility for scenarios where the number of iterations may not be predetermined. In this specific case, both approaches yield the same result, but the for loop is generally preferred for its clarity.

Task 5: Classes (Bank Account Class)

Scenario
You are designing a basic banking application.
Task
• Use AI tools to generate a Bank Account class with methods such as deposit(), withdraw(), and check_balance().
• Analyze the AI-generated class structure and logic.
• Add meaningful comments and explain the working of the code.
Expected Output #5
• Complete Python Bank Account class.
• Demonstration of deposit and withdrawal operations with updated balance.
• Well-commented code with a clear explanation.

Prompt :

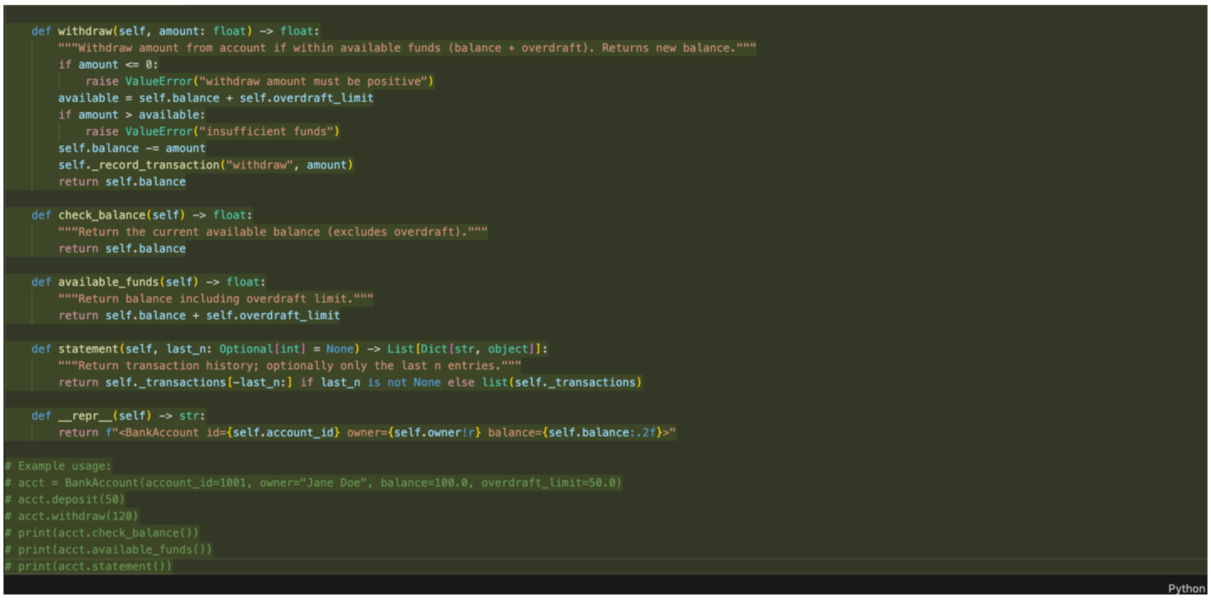
Generate a python classes for banking application with methods such as deposit(), withdraw(),and check_balance()



```
from datetime import datetime
```

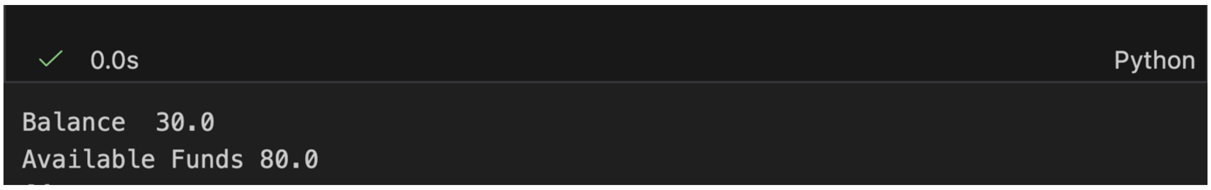
Generate a python classes for banking application with methods such as deposit(), withdraw(),and check_balance()

```
from typing import Dict, List, Optional

class BankAccount:
    """Simple bank account with deposit, withdraw and balance check + transaction history."""
    def __init__(self, account_id: int, owner: str, balance: float = 0.0, overdraft_limit: float = 0.0):
        self.account_id = account_id
        self.owner = owner
        self.balance = float(balance)
        self.overdraft_limit = float(overdraft_limit)
        self._transactions: List[Dict[str, object]] = []
        # record initial balance as an opening transaction
        self._record_transaction("opening", self.balance)

    def _record_transaction(self, ttype: str, amount: float) -> None:
        self._transactions.append({
            "timestamp": datetime.~~utcnow~~().isoformat() + "Z",
            "type": ttype,
            "amount": float(amount),
            "balance": float(self.balance)
        })

    def deposit(self, amount: float) -> float:
        """Deposit amount into account. Returns new balance."""
        if amount <= 0:
            raise ValueError("deposit amount must be positive")
        self.balance += amount
        self._record_transaction("deposit", amount)
        return self.balance
```



```python
    def withdraw(self, amount: float) -> float:
        """Withdraw amount from account if within available funds (balance + overdraft). Returns new balance."""
        if amount <= 0:
            raise ValueError("withdraw amount must be positive")
        available = self.balance + self.overdraft_limit
        if amount > available:
            raise ValueError("insufficient funds")
        self.balance -= amount
        self._record_transaction("withdraw", amount)
        return self.balance

    def check_balance(self) -> float:
        """Return the current available balance (excludes overdraft)."""
        return self.balance

    def available_funds(self) -> float:
        """Return balance including overdraft limit."""
        return self.balance + self.overdraft_limit

    def statement(self, last_n: Optional[int] = None) -> List[Dict[str, object]]:
        """Return transaction history; optionally only the last n entries."""
        return self._transactions[-last_n:] if last_n is not None else list(self._transactions)

    def __repr__(self) -> str:
        return f"<BankAccount id={self.account_id} owner={self.owner!r} balance={self.balance:.2f}>"

# Example usage:
# acct = BankAccount(account_id=1001, owner="Jane Doe", balance=100.0, overdraft_limit=50.0)
# acct.deposit(50)
# acct.withdraw(120)
# print(acct.check_balance())
# print(acct.available_funds())
# print(acct.statement())
```

Output:



```
Balance  30.0
Available Funds 80.0
```

Justification:

The Bank Account class models a simple bank account with methods to deposit and withdraw funds, check the balance, and view transaction history. It includes an overdraft limit feature, allowing withdrawals beyond the current balance up to a specified limit. Each transaction is recorded with a timestamp, type, amount, and resulting balance. The class also provides a string representation for easy debugging and logging.