ASSIGNMENT – 6.3

2303A51525

Batch-10

Task-1

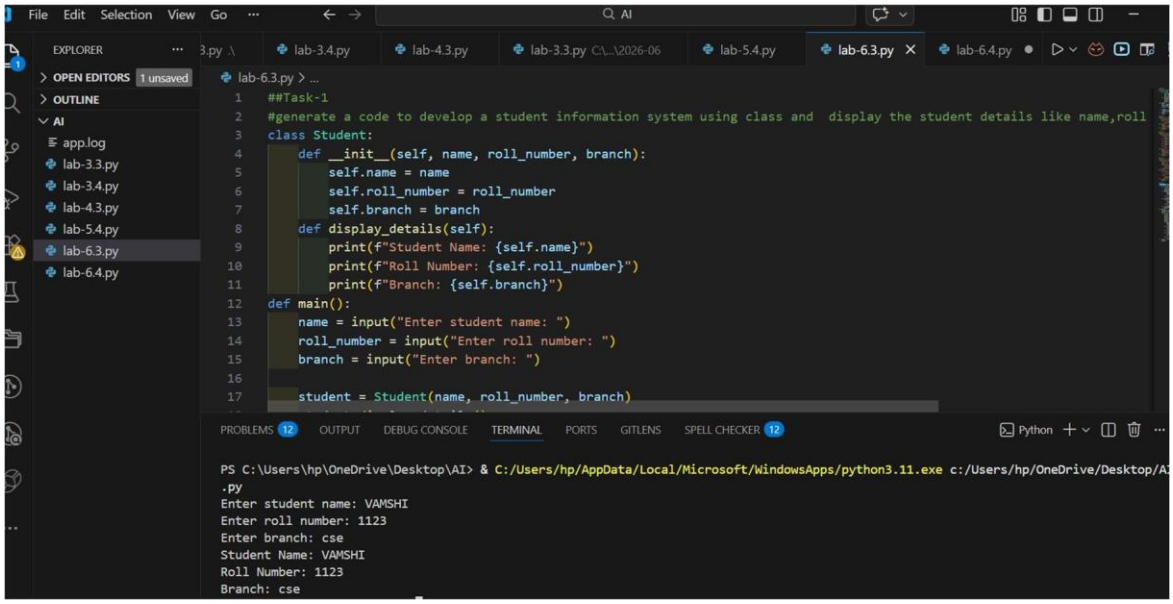
Prompt: generate a code to develop a student information system using class and display the student details like name,roll number and branch with user input.

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"Student Name: {self.name}")
print(f"Roll Number: {self.roll_number}")
print(f"Branch: {self.branch}") def main():
    name = input("Enter student name: ")
roll_number = input("Enter roll number: ")     branch
= input("Enter branch: ")


    student = Student(name, roll_number, branch)
student.display_details() if __name__ ==
"__main__":     main()
```

Output :



Code Analysis :

- Uses class and constructor (__init__) to store student details, showing OOP concept.
- Accepts user input dynamically, making the program interactive.
- display_details() method separates logic and printing, improving readability.
- Simple structure suitable for beginners to understand classes.
- Can be improved by storing multiple students using lists.

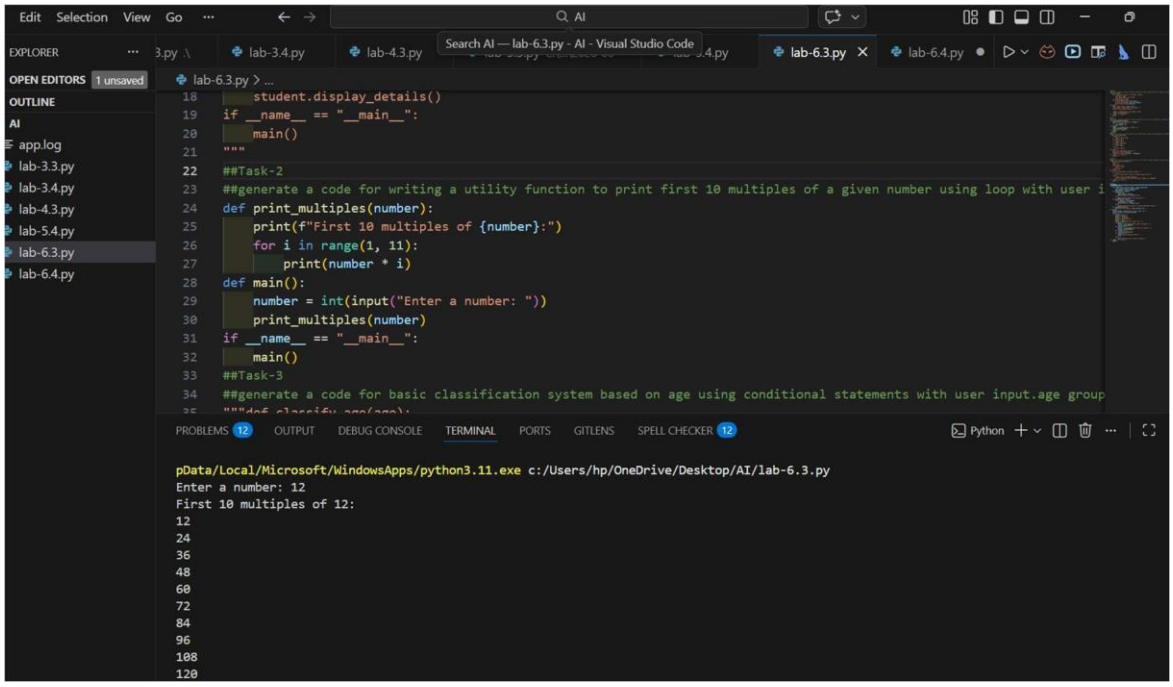
Task-2

Prompt: generate a code for writing a utility function to print first 10 multiples of a given number using loop with user input.without using try and except block.

Code :

```
def print_multiples(number):
    print(f"First 10 multiples of {number}:")
for i in range(1, 11):          print(number * i)
def main():
    number = int(input("Enter a number: "))
print_multiples(number) if __name__ ==
"__main__":     main()
```

Output :



Code Analysis :

- Uses a for loop with range(1,11) to generate first 10 multiples efficiently.
- Function-based design makes the code reusable.
- Takes integer input directly without exception handling as required.
- Clear output formatting helps users understand results easily.
- Could add input validation to avoid wrong data types.

Task-3

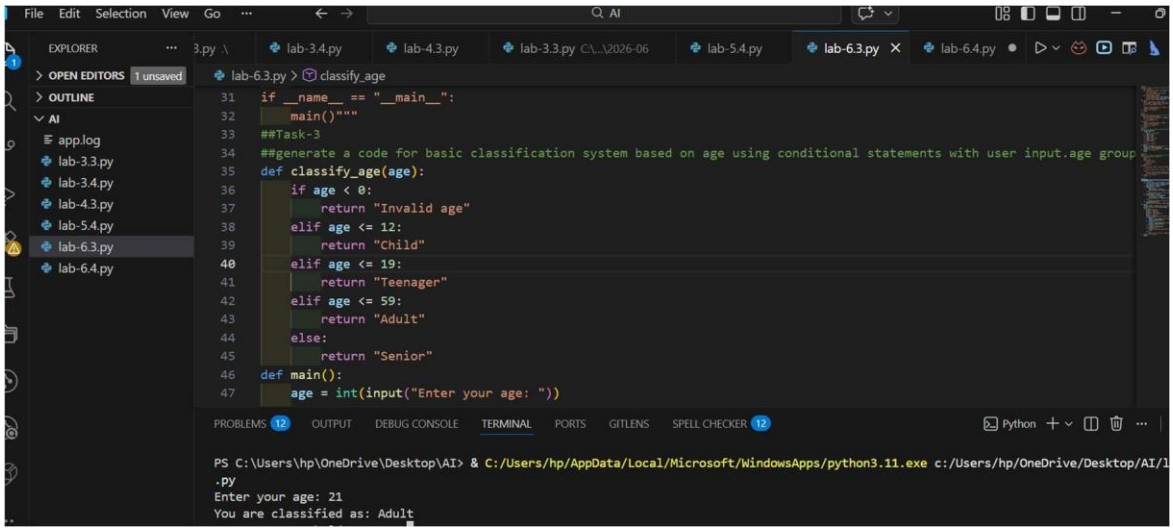
Prompt: generate a code for basic classification system based on age using conditional statements with user input.age groups like(child,teenager.adult and senior)without using try and except block.

Code :

```
def classify_age(age):
if age < 0:
    return "Invalid
age"    elif age <= 12:
return "Child"    elif
age <= 19:
```

```
        return "Teenager"
elif age <= 59:
return "Adult"     else:
        return "Senior"
def main():
    age = int(input("Enter your age: "))
category = classify_age(age)     print(f"You
are classified as: {category}") if __name__
== "__main__":
    main()
```

Output :



Code Analysis :

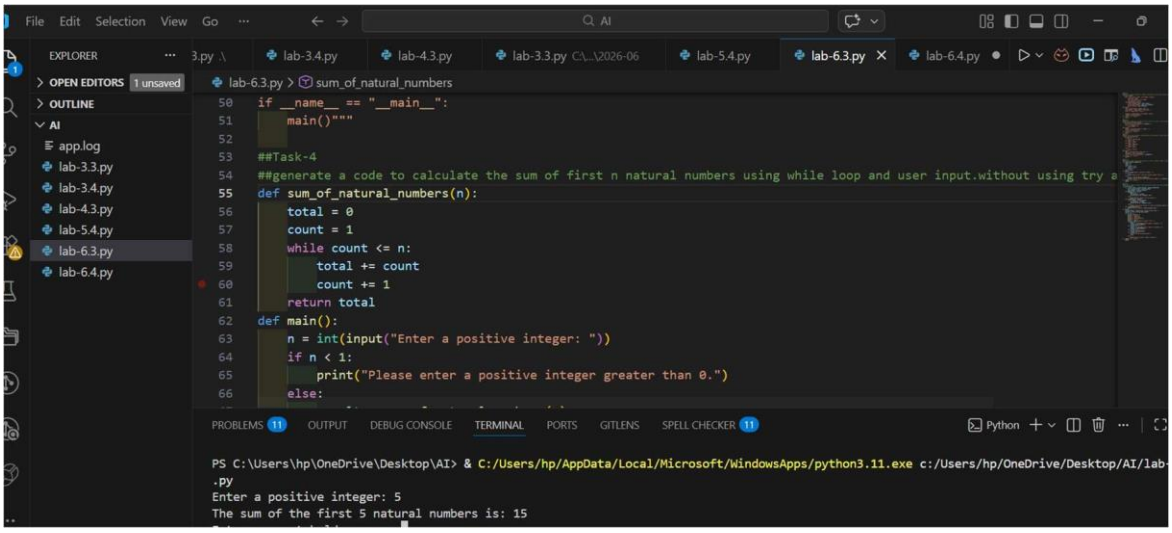
- Implements conditional statements (if-elif-else) for decision making.
- Categories like Child, Teenager, Adult, and Senior are clearly defined.
- Function returns values instead of printing directly, improving modularity.
- Handles negative age with “Invalid age” condition.
- Logic is simple but effective for basic classification problems.

Task-4

Prompt: generate a code to calculate the sum of first n natural numbers using while loop and user input without using try and except block.

Code :

```
def sum_of_natural_numbers(n):
    total = 0     count
= 1     while count
<= n:         total +=
count         count +=
1     return total def
main():
    n = int(input("Enter a positive integer: "))
if n < 1:
        print("Please enter a positive integer greater than 0.")
else:
        result = sum_of_natural_numbers(n)         print(f"The sum of
the first {n} natural numbers is: {result}") if __name__ ==
"__main__":
    main()   Output
:
```



Code Analysis :

- Uses while loop to iteratively calculate the sum step by step.
- Maintains separate variables (total, count) for clarity.
- Checks for positive input before calculation, improving correctness.
- Function returns result, making it reusable in other programs.
- Could be optimized using formula n*(n+1)/2 for efficiency.

Task 5

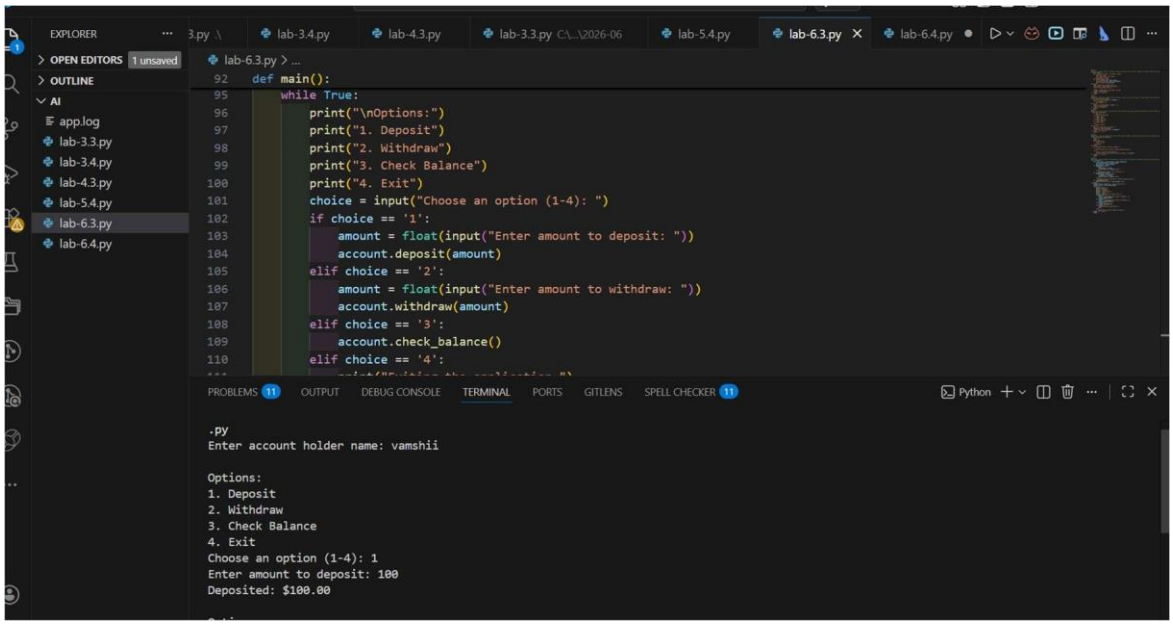
Prompt: #generate a code to design a bank application using class with methods like deposit,withdraw and check balance take user input. class BankAccount:

```
    def __init__(self,
account_holder,
initial_balance=0):
self.account_holder =
account_holder
        self.balance =
initial_balance
    def deposit(self,
amount):
        if amount > 0:
self.balance +=
amount
            print(f"Deposited:
${amount:.2f}")
        else:
print("Deposit amount
must be positive.")     def
withdraw(self, amount):
if 0 < amount <=
self.balance:
self.balance -=
amount
            print(f"Withdrew:
${amount:.2f}")
        else:
print("Insufficient balance
```

```
or invalid withdrawal
amount.")     def
check_balance(self):
print(f"Current balance:
${self.balance:.2f}") def
main():
    account_holder =
input("Enter account
holder name: ")     account
=
BankAccount(account_hol
der)     while True:
print("\nOptions:")
print("1. Deposit")
print("2. Withdraw")
print("3. Check
Balance")
        print("4. Exit")
choice = input("Choose an
option (1-4): ")         if
choice == '1':
amount =
float(input("Enter amount
to deposit: "))
account.deposit(am ount)
elif choice == '2':
amount =
float(input("Enter amount
to withdraw: "))
```

```
account.withdraw(a
mount)
        elif choice == '3':
account.check_bala nce()
elif choice == '4':
print("Exiting the
application.")
            break
else:
            print("Invalid
choice. Please try again.")
if __name__ ==
"__main__":
    main()
```

Output :



Code Analysis :

- Demonstrates OOP with methods like deposit, withdraw, and check balance.
- Menu-driven loop allows continuous user interaction.

- Includes validation for negative deposits and insufficient balance.
- Keeps balance as an instance variable, showing encapsulation concept.
- Can be enhanced by adding PIN security or transaction history.