ASSIGNMENT

TASK 1 : Start a Python class named Student with attributes name, roll_number, and marks. Prompt GitHub Copilot to complete methods for displaying details and checking if marks are above average.

EXPECTED CODE WITH OUTPUT :

```python
def get_student_details():
    """Gets student details from user input."""
    students = []
    num_students = int(input("Enter the number of students: "))
    for i in range(num_students):
        print(f"\nEnter details for student {i+1}:")
        name = input("Name: ")
        roll_number = input("Roll Number: ")
        marks = float(input("Marks: "))
        status = input("Pass or Fail: ").lower()
        students.append({'name': name, 'roll_number': roll_number, 'marks': marks, 'status': status})
    return students

def calculate_average_marks(students):
    """Calculates the average marks of all students."""
    total_marks = sum(student['marks'] for student in students)
    return total_marks / len(students) if students else 0

def display_above_average_students(students, average_marks):
    """Displays students with marks above the average."""
    print("\nStudents with marks above average:")
    print("-" * 40)
    for student in students:
        if student['marks'] > average_marks:
            print(f"✓ Name: {student['name']}, Roll Number: {student['roll_number']}, Marks: {student['marks']}, Status: {student['status']}")
    print("-" * 40)

if __name__ == "__main__":
    student_list = get_student_details()
    avg_marks = calculate_average_marks(student_list)
    print(f"\nAverage marks: {avg_marks:.2f}")
    display_above_average_students(student_list, avg_marks)
```

```
Enter the number of students: 3

Enter details for student 1:
Name: vikas
Roll Number: 2116
Marks: 98
Pass or Fail: pass

Enter details for student 2:
Name: bhanu
Roll Number: 118
Marks: 99
Pass or Fail: pass

Enter details for student 3:
Name: sharath
Roll Number: 117
Marks: 89
Pass or Fail: pass

Average marks: 95.33

Students with marks above average:
----------------------------------------
✓ Name: vikas, Roll Number: 2116, Marks: 98.0, Status: pass
✓ Name: bhanu, Roll Number: 118, Marks: 99.0, Status: pass
----------------------------------------
```

EXPLANATION OF THE CODE :

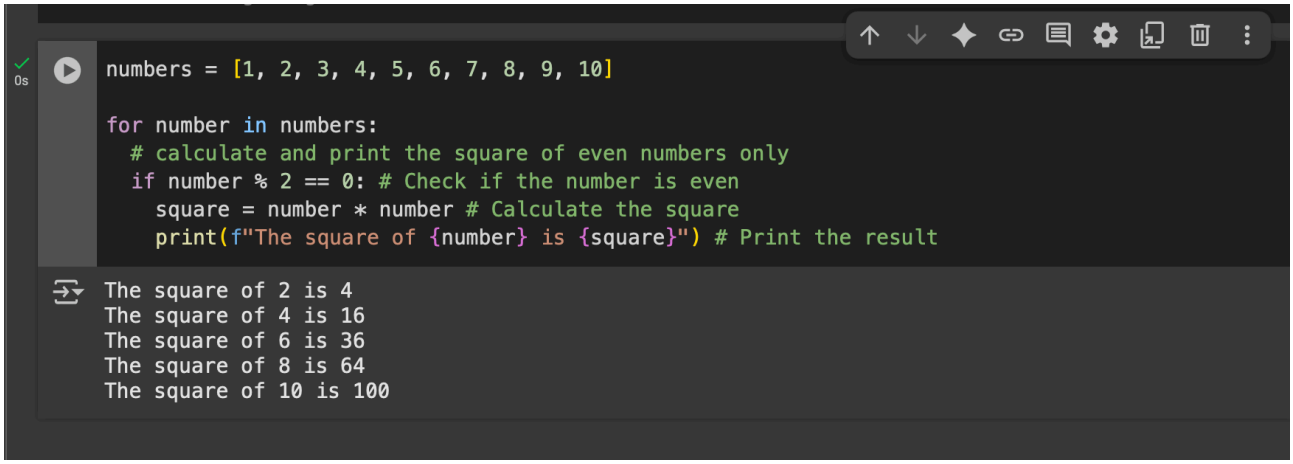
1. **`get_student_details()` Function:**
    - This function is responsible for getting the student information from the user.
    - It first asks for the total number of students.
    - Then, it loops that many times, each time prompting for the name, roll number, marks, and pass/fail status for a single student.
    - It stores each student's details as a dictionary within a list called `students`.
    - Finally, it returns the `students` list.
2. **`calculate_average_marks(students)` Function:**
    - This function takes the list of `students` as input.
    - It calculates the sum of the 'marks' for all students in the list.
    - It then divides the total marks by the number of students to get the average.
    - It includes a check (`if students else 0`) to prevent division by zero if the student list is empty.
    - It returns the calculated average marks.

3. **`display_above_average_students(students, average_marks)` Function:**
    - This function takes the list of `students` and the calculated `average_marks` as input.
    - It prints a header indicating that it will display students with above-average marks.
    - It then iterates through the `students` list.
    - For each student, it checks if their 'marks' are greater than the `average_marks`.
    - If a student's marks are above average, it prints their details (name, roll number, marks, and status) with a checkmark (✓) at the beginning of the line.
    - It prints a separator line before and after the list of students.
4. **`if __name__ == "__main__":` Block:**
    - This is the main part of the script that runs when you execute the code.
    - `student_list = get_student_details()`: It calls the `get_student_details()` function to get the student data and stores it in the `student_list` variable.
    - `avg_marks = calculate_average_marks(student_list)`: It calls the `calculate_average_marks()` function with the

- `student_list = get_student_details()`: It calls the `get_student_details()` function to get the student data and stores it in the `student_list` variable.
- `avg_marks = calculate_average_marks(student_list)`: It calls the `calculate_average_marks()` function with the `student_list` to calculate the average marks and stores it in the `avg_marks` variable.
- `print(f"\nAverage marks: {avg_marks:.2f}")`: It prints the calculated average marks, formatted to two decimal places.
- `display_above_average_students(student_list, avg_marks)`: It calls the `display_above_average_students()` function to print the details of students who scored above the average.

TASK 2 : Write the first two lines of a for loop to iterate through a list of numbers. Use a comment prompt to let Copilot suggest how to calculate and print the square of even numbers only.

Expected code with output:



Explanation of the code :

5. `square = number * number`: This line is inside the `if` block. It is only executed if the `number` is even. It calculates the square of the current `number` by multiplying it by itself and stores the result in a variable named `square`.
6. `print(f"The square of {number} is {square}")`: This line is also inside the `if` block and is executed only for even numbers.
    - `print()` is a function that displays output to the console.
    - `f"The square of {number} is {square}"` is an f-string (formatted string literal). It allows you to embed the values of variables directly within a string. In this case, it constructs a sentence showing the original even `number` and its calculated `square`.

1. `numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`: This line initializes a Python list named `numbers` containing integers from 1 to 10. This is the list that the code will iterate through.
2. `for number in numbers:`: This is a `for` loop. It's designed to process each item in the `numbers` list one by one. In each cycle of the loop, the current item from the `numbers` list is assigned to the variable `number`.
3. `# calculate and print the square of even numbers only`: This is a comment and is ignored by the Python interpreter. It serves as a note to explain the purpose of the code that follows within the loop.
4. `if number % 2 == 0:`: This is an `if` statement that checks a condition.
    - `number % 2` calculates the remainder when the current `number` is divided by 2.
    - `== 0` checks if the remainder is equal to 0.
    - If the remainder is 0, it means the `number` is even, and the code inside the `if` block will be executed. If the remainder is not 0 (meaning the number is odd), the code inside the `if` block is skipped.

Task 3 : Create a class called BankAccount with attributes account_holder and balance. Use Copilot to complete methods for deposit(), withdraw(), and check for insufficient balance.

Expected code with output:

```python
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}. New balance: ${self.balance}")
        else:
            print("Deposit amount must be positive.")

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

    def check_balance(self):
        print(f"Account holder: {self.account_holder}, Current balance: ${self.balance}")

# Example usage:
account1 = BankAccount("Alice Smith", 1000)
account1.check_balance()
account1.deposit(500)
account1.withdraw(200)
account1.withdraw(2000) # Attempt to withdraw more than balance
```

```
Account holder: Alice Smith, Current balance: $1000
Deposited: $500. New balance: $1500
Withdrew: $200. New balance: $1300
Insufficient balance.
```

Explanation of the code :

1. `class BankAccount:` : This line defines a new class named `BankAccount`. A class is a blueprint for creating objects (instances) that have certain properties (attributes) and behaviors (methods).
2. `def __init__(self, account_holder, initial_balance=0):` : This is the constructor method of the class.
    - `__init__` is a special method that is automatically called when you create a new object of the `BankAccount` class.
    - `self` refers to the instance of the class being created.
    - `account_holder` is a parameter that will store the name of the account holder.
    - `initial_balance=0` is a parameter for the starting balance. It has a default value of 0, meaning you don't have to provide an initial balance when creating an account if you want it to start at zero.
    - Inside the method, `self.account_holder = account_holder` and `self.balance = initial_balance` assign the values passed to the constructor to the object's attributes. These attributes (`account_holder` and `balance`) will store the state of each individual bank account object.

3. `def deposit(self, amount):` This method is used to deposit money into the account.
    - `self` refers to the instance of the class.
    - `amount` is the amount to be deposited.
    - `if amount > 0:` checks if the deposit amount is positive. You can't deposit a negative amount.
    - `self.balance += amount`: If the amount is positive, it adds the deposit amount to the current `balance` of the account.
    - `print(f"Deposited: ${amount}. New balance: ${self.balance}")`: Prints a confirmation message showing the deposited amount and the new balance.
    - `else: print("Deposit amount must be positive.")`: If the amount is not positive, it prints an error message.
4. `def withdraw(self, amount):` This method is used to withdraw money from the account.
    - `self` refers to the instance of the class.
    - `amount` is the amount to be withdrawn.
    - `if amount > 0:` checks if the withdrawal amount is positive.

    - `if self.balance >= amount:`: This is a nested `if` statement that checks if the current `balance` is greater than or equal to the `amount` being withdrawn. This prevents withdrawing more money than is available (insufficient balance).
    - `self.balance -= amount`: If the balance is sufficient, it subtracts the withdrawal amount from the current `balance`.
    - `print(f"Withdrew: ${amount}. New balance: ${self.balance}")`: Prints a confirmation message showing the withdrawn amount and the new balance.
    - `else: print("Insufficient balance.")`: If the balance is insufficient, it prints an error message.
    - The outer `else` block handles cases where the withdrawal amount is not positive.
5. `def check_balance(self):` This method is used to display the current account balance.
    - `self` refers to the instance of the class.
    - `print(f"Account holder: {self.account_holder}, Current balance: ${self.balance}")`: Prints the account holder's name and the current balance.

6. **Example Usage:** The lines outside the class definition demonstrate how to use the `BankAccount` class:
    - `account1 = BankAccount("Alice Smith", 1000)`: This creates a new object (an instance) of the `BankAccount` class. It calls the `__init__` constructor, setting the `account_holder` to "Alice Smith" and the `initial_balance` to 1000. The object is assigned to the variable `account1`.
    - `account1.check_balance()`: Calls the `check_balance` method on the `account1` object to display its initial balance.
    - `account1.deposit(500)`: Calls the `deposit` method on `account1` to add $500. * `account1.withdraw(200)`: Calls the `withdraw` method on `account1` to remove $$500. * `account1.withdraw(200)`: Calls the `withdraw` method on `account1` to remove $200.
    - `account1.withdraw(2000)`: Calls the `withdraw` method on `account1` to attempt to remove $2000. Since the balance is only $$2000. Since the balance is only $1300 at this point, the "Insufficient balance" message will be printed.

Task 4 : Define a list of student dictionaries with keys name and score. Ask Copilot to write a while loop to print the names of students who scored more than 75

Expected code with output :

```
[14] students = [
        {"name": "Alice", "score": 85},
        {"name": "Bob", "score": 70},
        {"name": "Charlie", "score": 92},
        {"name": "David", "score": 78},
        {"name": "Eve", "score": 65}
    ]

    i = 0
    while i < len(students):
      if students[i]["score"] > 75:
        print(students[i]["name"])
      i += 1
```

```
Alice
Charlie
David
```

Explanation of the code :

1. `students = [...]`: This line creates a list named `students`. Each element in this list is a dictionary. Each dictionary represents a student and has two key-value pairs: `"name"` (the student's name) and `"score"` (the student's score).
2. `i = 0`: This line initializes a variable `i` to 0. This variable will be used as an index to access elements in the `students` list and as a counter for the `while` loop.
3. `while i < len(students):`: This is the start of a `while` loop.
    - `len(students)` calculates the number of elements (dictionaries) in the `students` list.
    - The condition `i < len(students)` means the loop will continue to execute as long as the value of `i` is less than the total number of students in the list. This ensures that the loop iterates through each student from the first one (index 0) up to the last one.

4. `if students[i]["score"] > 75:`: This is an `if` statement inside the `while` loop.
    - `students[i]` accesses the dictionary at the current index `i` in the `students` list.
    - `students[i]["score"]` accesses the value associated with the key `"score"` within that dictionary (the current student's score).
    - `> 75` checks if the student's score is greater than 75.
    - If the condition is true (the student's score is greater than 75), the code block inside this `if` statement is executed.
5. `print(students[i]["name"])`: This line is inside the `if` block. It is executed only if the student's score is greater than 75.
    - `students[i]["name"]` accesses the value associated with the key `"name"` within the current student's dictionary (the student's name).
    - `print()` displays the student's name to the console.

6. `i += 1`: This line is inside the `while` loop but outside the `if` block. It is executed in every iteration of the `while` loop.
    - `i += 1` is a shorthand for `i = i + 1`. It increments the value of `i` by 1. This is crucial for the `while` loop to eventually terminate, as it moves the index to the next student in the list in each iteration.

Task 5 : Begin writing a class ShoppingCart with an empty items list. Prompt Copilot to generate methods to add_item, remove_item, and use a loop to calculate the total bill using conditional discounts.

Expected code with output :

```
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, name, price, quantity):
        item = {'name': name, 'price': price, 'quantity': quantity}
        self.items.append(item)

    def remove_item(self, name):
        initial_item_count = len(self.items)
        self.items = [item for item in self.items if item['name'] != name]
        if len(self.items) < initial_item_count:
            print(f"Removed item: {name}")
        else:
            print(f"Item not found: {name}")

    def calculate_total(self):
        total_bill = 0
        for item in self.items:
            total_bill += item['price'] * item['quantity']

        # Apply conditional discounts
        if total_bill > 100:
            total_bill *= 0.9  # 10% discount for total over $100
            print("Applied 10% discount.")
        elif any(item['name'] == 'DiscountItem' for item in self.items):
            total_bill *= 0.95 # 5% discount if 'DiscountItem' is in the cart
            print("Applied 5% discount for having 'DiscountItem'.")

        return total_bill
```

(variable) item: Any

```
Applied 10% discount.
Final total bill: $1192.50
Final total bill: $80.00
Applied 5% discount for having 'DiscountItem'.
Final total bill: $19.00
```

```python
# Example usage:
cart = ShoppingCart()
cart.add_item("Laptop", 1200, 1)
cart.add_item("Mouse", 25, 2)
cart.add_item("Keyboard", 75, 1)
total = cart.calculate_total()
print(f"Final total bill: ${total:.2f}")

cart2 = ShoppingCart()
cart2.add_item("Book", 20, 3)
cart2.add_item("Pen", 2, 10)
total2 = cart2.calculate_total()
print(f"Final total bill: ${total2:.2f}")

cart3 = ShoppingCart()
cart3.add_item("DiscountItem", 10, 1)
cart3.add_item("Notebook", 5, 2)
total3 = cart3.calculate_total()
print(f"Final total bill: ${total3:.2f}")
```

```python
# Create an instance of the ShoppingCart class
my_cart = ShoppingCart()

# Add several items to the shopping cart
my_cart.add_item("Apple", 0.5, 10)
my_cart.add_item("Banana", 0.3, 20)
my_cart.add_item("Orange", 0.7, 5)
my_cart.add_item("Grapes", 2.5, 1)

# Optionally, remove an item
my_cart.remove_item("Banana")

# Calculate the final total bill
final_bill = my_cart.calculate_total()

# Print the final total bill
print(f"Your final bill is: ${final_bill:.2f}")
```

```
Removed item: Banana
Your final bill is: $11.00
```

Explanation of the code :

- **Define the `shoppingcart` class**: Start by defining the class structure and the `__init__` method to initialize an empty list for items.
- **Implement `add item` method**: Create a method to add items to the `items` list. Each item could be a dictionary with details like name, price, and quantity.
- **Implement `remove item` method**: Create a method to remove items from the `items` list based on certain criteria (e.g., item name).
- **Implement `calculate total` method**: Create a method to iterate through the `items` list, calculate the total bill, and apply conditional discounts based on criteria you define (e.g., total amount, specific items, quantity).
- **Example usage**: Provide example code to demonstrate how to create a `ShoppingCart` object, add/remove items, and calculate the total bill.
- **Finish task**: Review the code and plan to ensure all requirements are met.