



MAPÚA UNIVERSITY

SCHOOL OF ELECTRICAL, ELECTRONICS, AND COMPUTER ENGINEERING

Experiment 3: Object Oriented Design and Implementation

CPE106L (Software Design Laboratory)

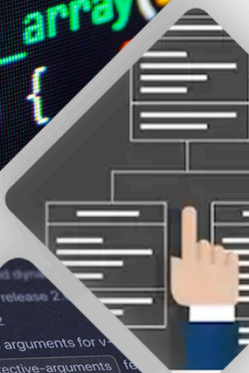
Member 1 (Cromuel Pangilinan)

Member 2 (Karl Ignatius G. Gavino)

Member 3 (Nicole Allyson B. Torres)

Member 3 (Xavier Alonzo)

**Group No.: 9
Section: B3**



Readings, Insights, and Reflection

METIS #1:

Systems Analysis and Design: An Object Oriented Approach with UML

Alan Dennis, Barbara Haley Wixom, David Tegarden

Edition 5

ISBN: 9781119561217

Wiley Global Education US

2015

- Chapter 4, pp 121 – 130 (Use case Diagram)

- Chapter 5, pp 176 – 185 (Class Diagram)

METIS #2:

Fundamentals of Python: First Programs

Kenneth A. Lambert

Edition 2

ISBN: 9781337671019

Cengage Learning US

2019

- Chapter 9: Design with Classes

- Torres, Nicole Allyson B.
 - Fundamentals of Python: First Programs (Chapter 9 – Design with classes)
 - Chapter 9 of "Fundamentals of Python: First Programs" by Kenneth A. Lambert is a helpful guide for designing programs with classes in Python. The chapter explains crucial object-oriented design principles like encapsulation, inheritance, and polymorphism and is an excellent resource for beginners. The chapter describes creating custom classes in Python and defining attributes and methods to represent object characteristics and behaviors. It also explores how different classes can work together effectively within a program. The chapter introduces common design patterns like Singleton, Factory, and Observer, which encourage best code reusability and maintainability practices. Overall, Chapter 9 equips learners with the skills to effectively design robust and scalable Python programs using object-oriented principles and design patterns.
- Gavino, Karl Ignatius G.

- o In the second edition of "Fundamentals of Python: First Programs" by Kenneth A. Lambert, readers are provided with a comprehensive grasp of Python programming, highlighting Chapter 9: "Design with Classes." The author carefully explained the properties and different uses of object-oriented programming (OOP) principles, equipping us students with a solid groundwork for crafting and deploying classes to encapsulate functionality. Emphasizing the significance of inheritance, polymorphism, and abstraction, this chapter enhanced us with essential tools for constructing dependable Python applications. The book enables novice programmers to cultivate skills in writing structured and coherent code, while also fostering a deeper comprehension of software architecture and design patterns—essential assets for tackling complex programming endeavors.

Answers to Questions:

1. A
2. C
3. B
4. B
5. B
6. B
7. A
8. B
9. B
- 10.A

InLab

Objectives

1. **Debug** the sample programs provided (student.py).
2. **Use** Anaconda terminal in running python statements.
3. **Use** Microsoft Visual Studio Code in debugging.
4. Using VM Virtual Box. .

● Tools Used

- Microsoft Visual Studio Code 2022
- Anaconda (or Python Virtual Environment)
- VM Virtual Box

● Procedure.

1. We created a UML Diagram for the program (**student.py**), as shown in Figure 1. This diagram helped us understand how the program works.

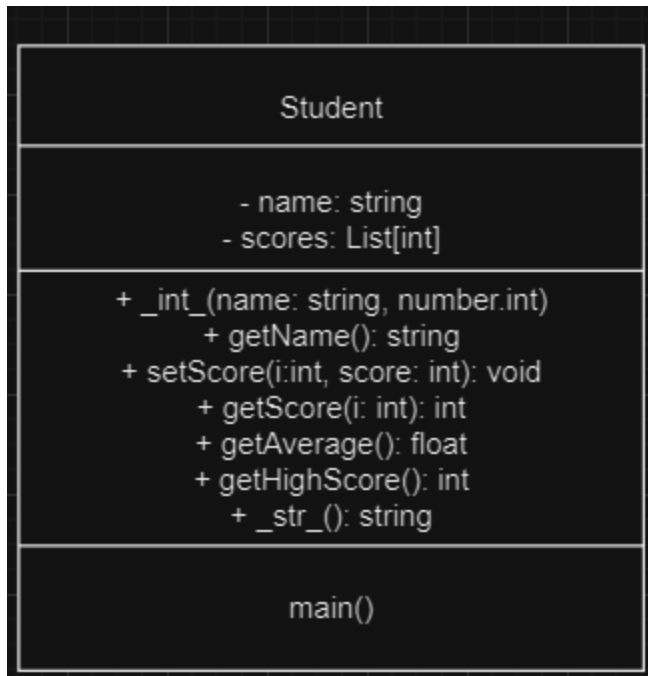
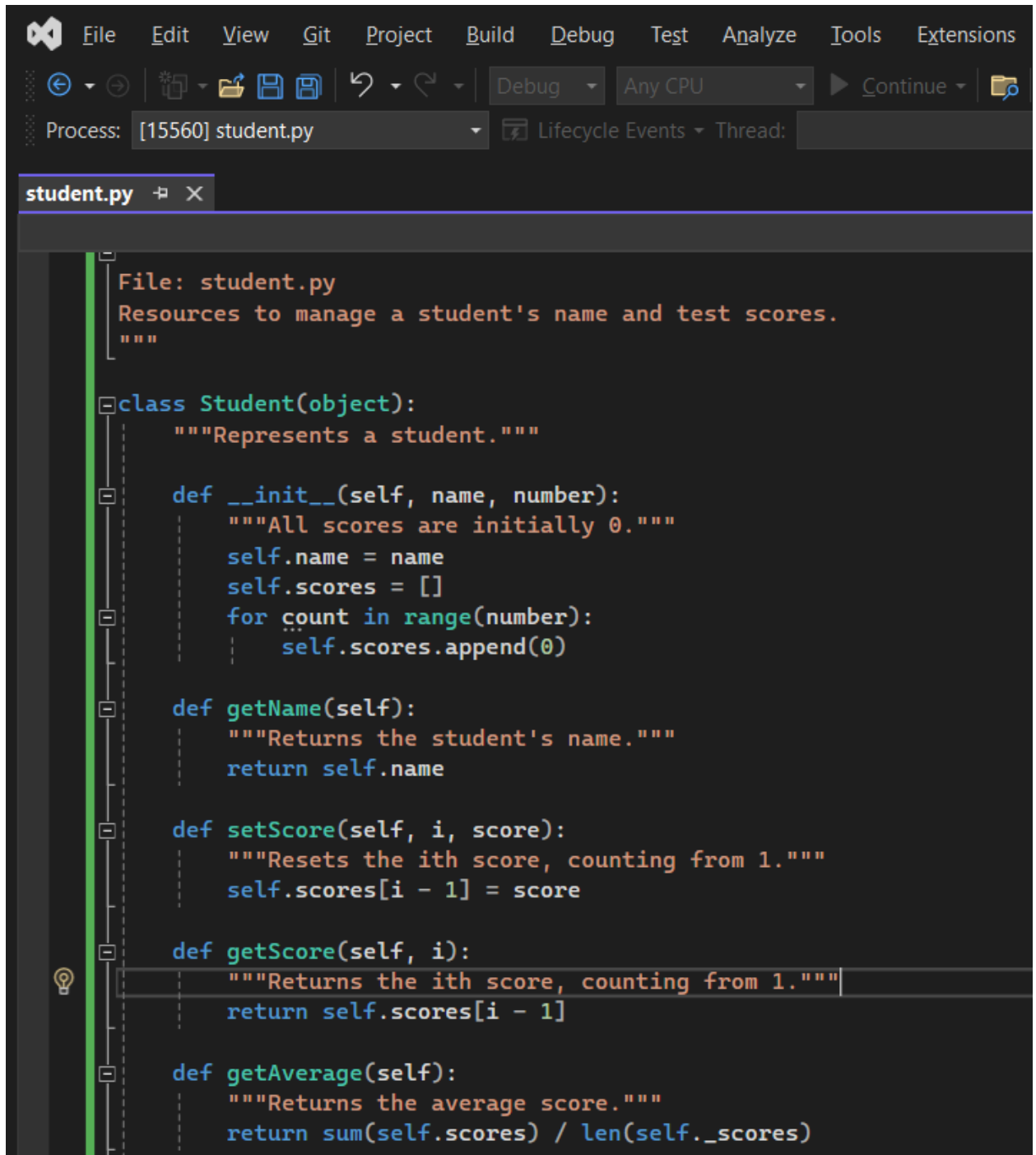


Figure 1: UML Diagram for Sample Program student.py

2. In Figures 2 and 2.1, we successfully ran the student.py sample program on Microsoft Visual Studio Code 2022.

The image shows the Visual Studio Code interface with a Python file named 'student.py' open. The top menu bar includes File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, and Extensions. Below the menu is a toolbar with icons for running and debugging. The 'Process' dropdown shows '[15560] student.py' and the 'Thread' dropdown is empty. The code editor displays the following Python code:

```
File: student.py
Resources to manage a student's name and test scores.
"""

class Student(object):
    """Represents a student."""

    def __init__(self, name, number):
        """All scores are initially 0."""
        self.name = name
        self.scores = []
        for count in range(number):
            self.scores.append(0)

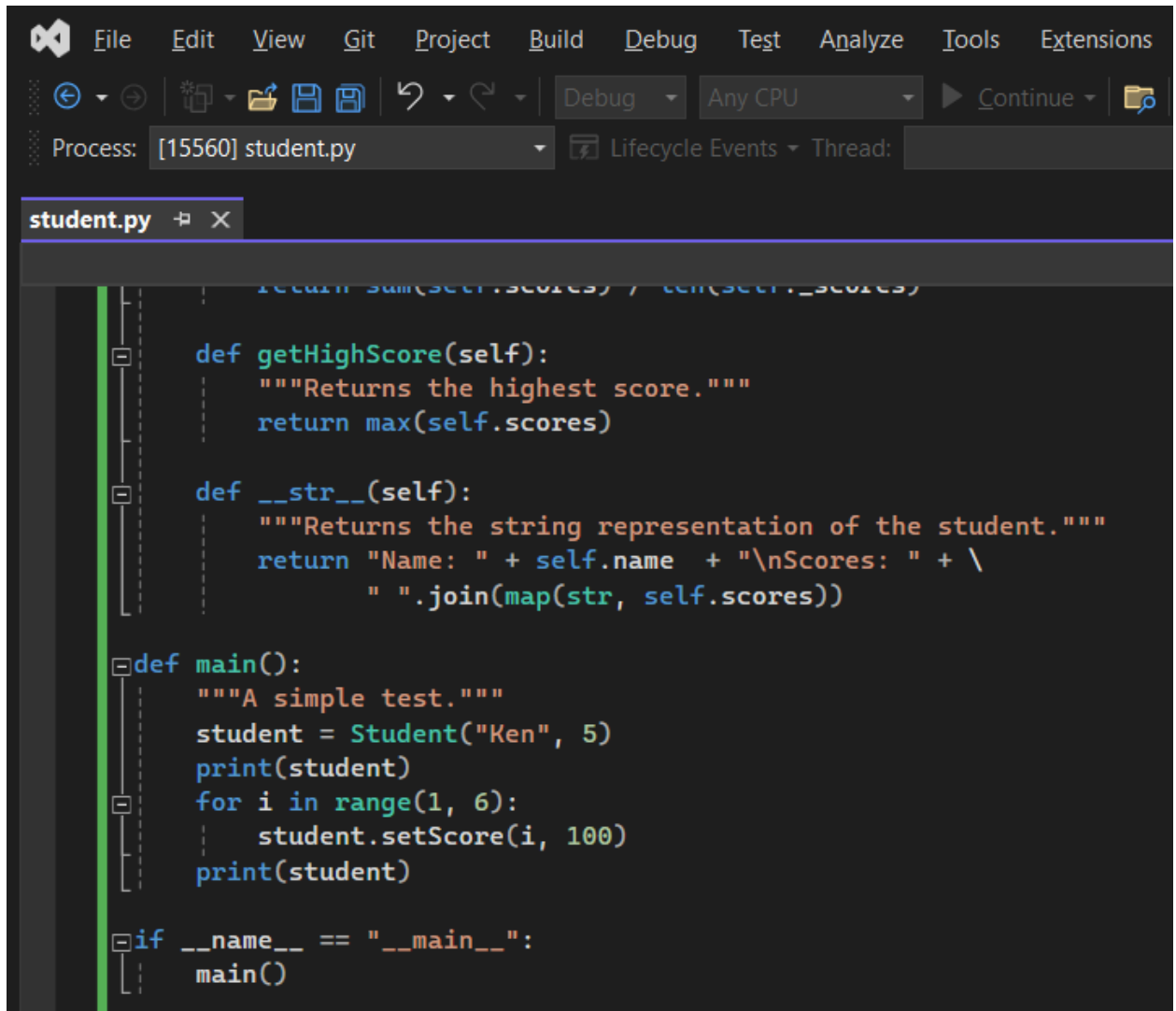
    def getName(self):
        """Returns the student's name."""
        return self.name

    def setScore(self, i, score):
        """Resets the ith score, counting from 1."""
        self.scores[i - 1] = score

    def getScore(self, i):
        """Returns the ith score, counting from 1."""
        return self.scores[i - 1]

    def getAverage(self):
        """Returns the average score."""
        return sum(self.scores) / len(self._scores)
```

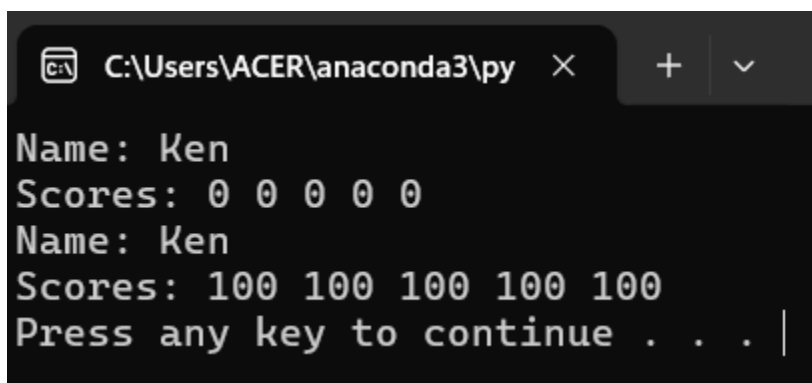
Figure 2: Running the sample program student.py in VS code

A screenshot of the Visual Studio Code interface. The top menu bar includes File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, and Extensions. Below the menu is a toolbar with icons for running and debugging. The status bar at the bottom shows 'Process: [15560] student.py' and 'Lifecycle Events'. The main editor window displays the code for 'student.py'. The code includes a class 'Student' with methods 'getHighScore' and '__str__', a 'main' function, and a standard Python entry point.

```
def getHighScore(self):  
    """Returns the highest score."""  
    return max(self.scores)  
  
def __str__(self):  
    """Returns the string representation of the student."""  
    return "Name: " + self.name + "\nScores: " + \n        " ".join(map(str, self.scores))  
  
def main():  
    """A simple test."""  
    student = Student("Ken", 5)  
    print(student)  
    for i in range(1, 6):  
        student.setScore(i, 100)  
    print(student)  
  
if __name__ == "__main__":  
    main()
```

Figure 2.1: Running the sample program student.py in VS code

3. In Figure 3, the output of the sample program "student.py" is displayed while running.

A screenshot of a terminal window titled 'C:\Users\ACER\anaconda3\py'. It shows the output of the 'student.py' program. The output consists of two lines of text: 'Name: Ken' followed by 'Scores: 0 0 0 0 0', and then 'Name: Ken' followed by 'Scores: 100 100 100 100 100'. The prompt 'Press any key to continue . . . |' is visible at the bottom.

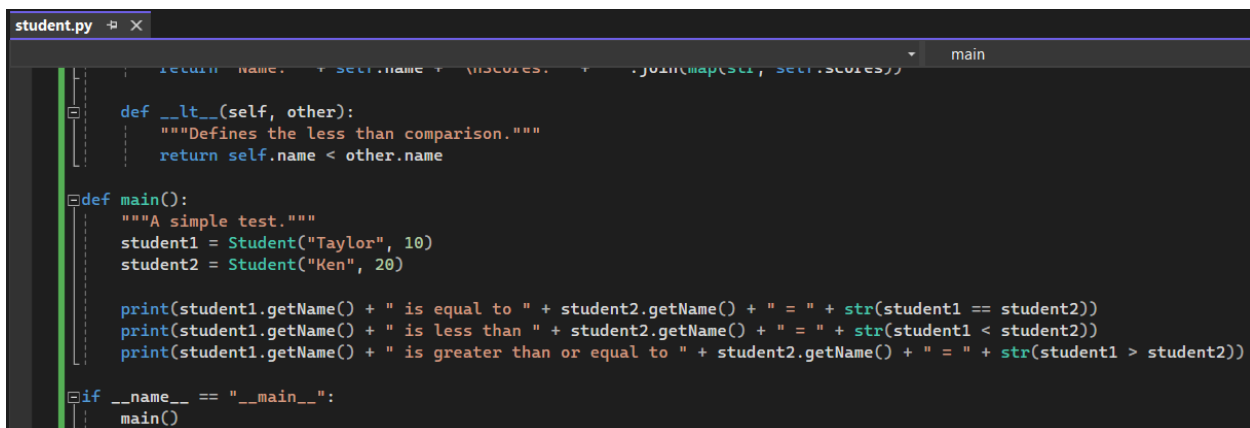
```
Name: Ken  
Scores: 0 0 0 0 0  
Name: Ken  
Scores: 100 100 100 100 100  
Press any key to continue . . . |
```

Figure 3: Output of the Sample Program student.py

PostLab

Programming Problems

1. Add three methods to the Student class (in the file student.py) that compare two Student objects. One method should test for equality. A second method should test for less than. The third method should test for greater than or equal to. In each case, the method returns the result of the comparison of the two students' names. Include a main function that tests all of the comparison operators. (LO: 10.2).



```
student.py ✕
return name + " " + self.name + " (" + str(scores) + ")".join(map(str, self.scores))

def __lt__(self, other):
    """Defines the less than comparison."""
    return self.name < other.name

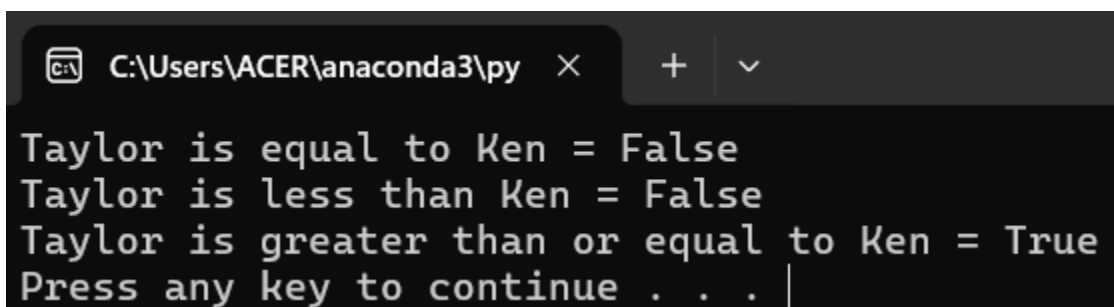
def main():
    """A simple test."""
    student1 = Student("Taylor", 10)
    student2 = Student("Ken", 20)

    print(student1.getName() + " is equal to " + student2.getName() + " = " + str(student1 == student2))
    print(student1.getName() + " is less than " + student2.getName() + " = " + str(student1 < student2))
    print(student1.getName() + " is greater than or equal to " + student2.getName() + " = " + str(student1 > student2))

if __name__ == "__main__":
    main()
```

Figure 4: Adding Three Methods to Student Class

This code defines a class called *Student* with a method called `__lt__` that compares two *Student* objects' names alphabetically using the `<` operator. The primary function creates two *Student* objects with different names and compares their names using string formatting and comparison operators (`==`, `<`, `>`). The output shows whether the two students have the same name (`==`), the order of their names (`</>`), and how to define custom comparison methods for a class based on specific criteria such as alphabetical order by name.

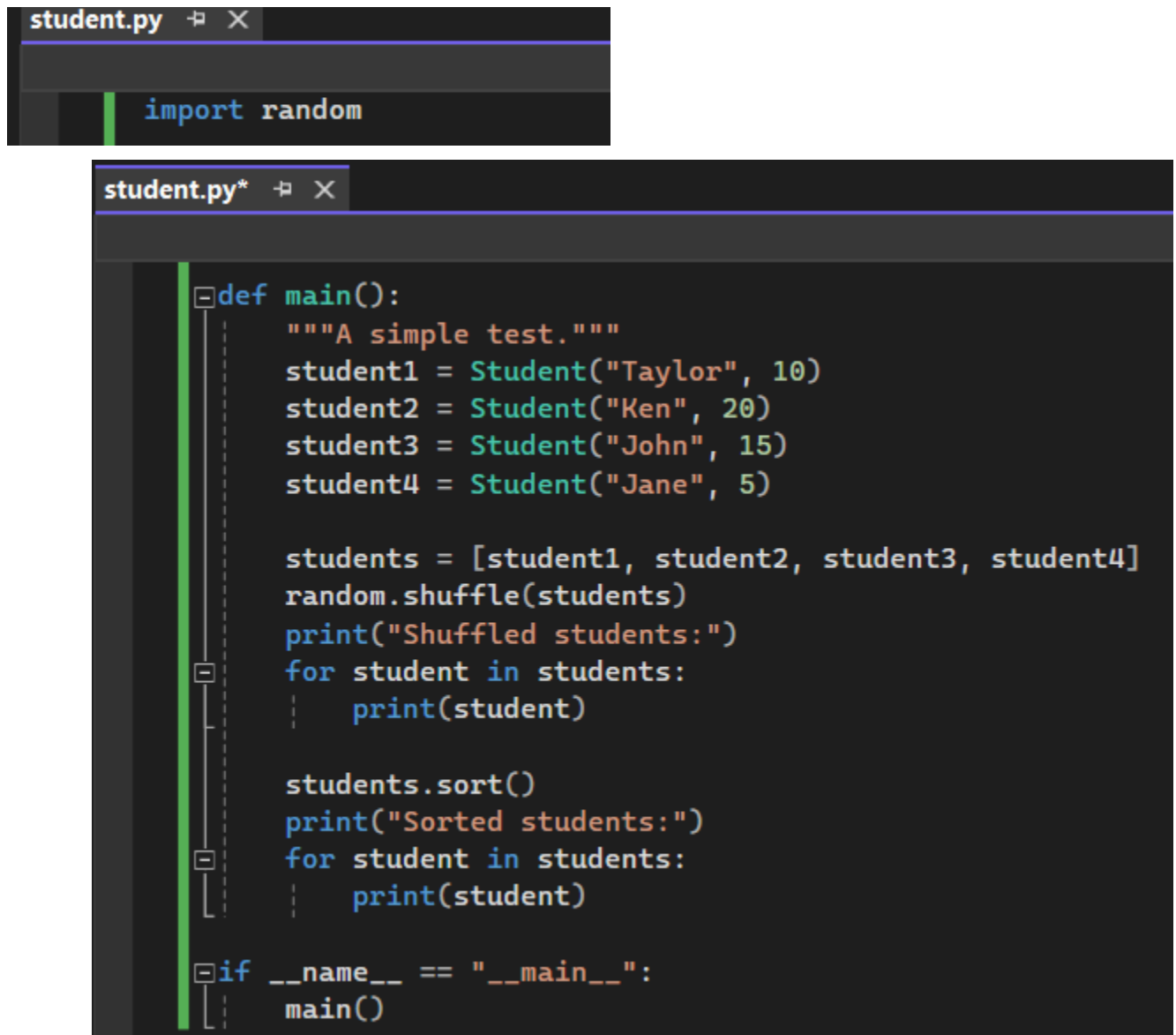


```
C:\Users\ACER\anaconda3\py ✕ + v
Taylor is equal to Ken = False
Taylor is less than Ken = False
Taylor is greater than or equal to Ken = True
Press any key to continue . . .
```

Figure 5: Output of the program

Figure 5 shows that the two names are compared as equal, not equal, and greater than or equal.

2. This exercise assumes that you have completed Programming Exercise 1. Place several Student objects containing different names into a list and shuffle it. Then run the sort method with this list and display all of the students' information. (LO: 10.1, 10.2).

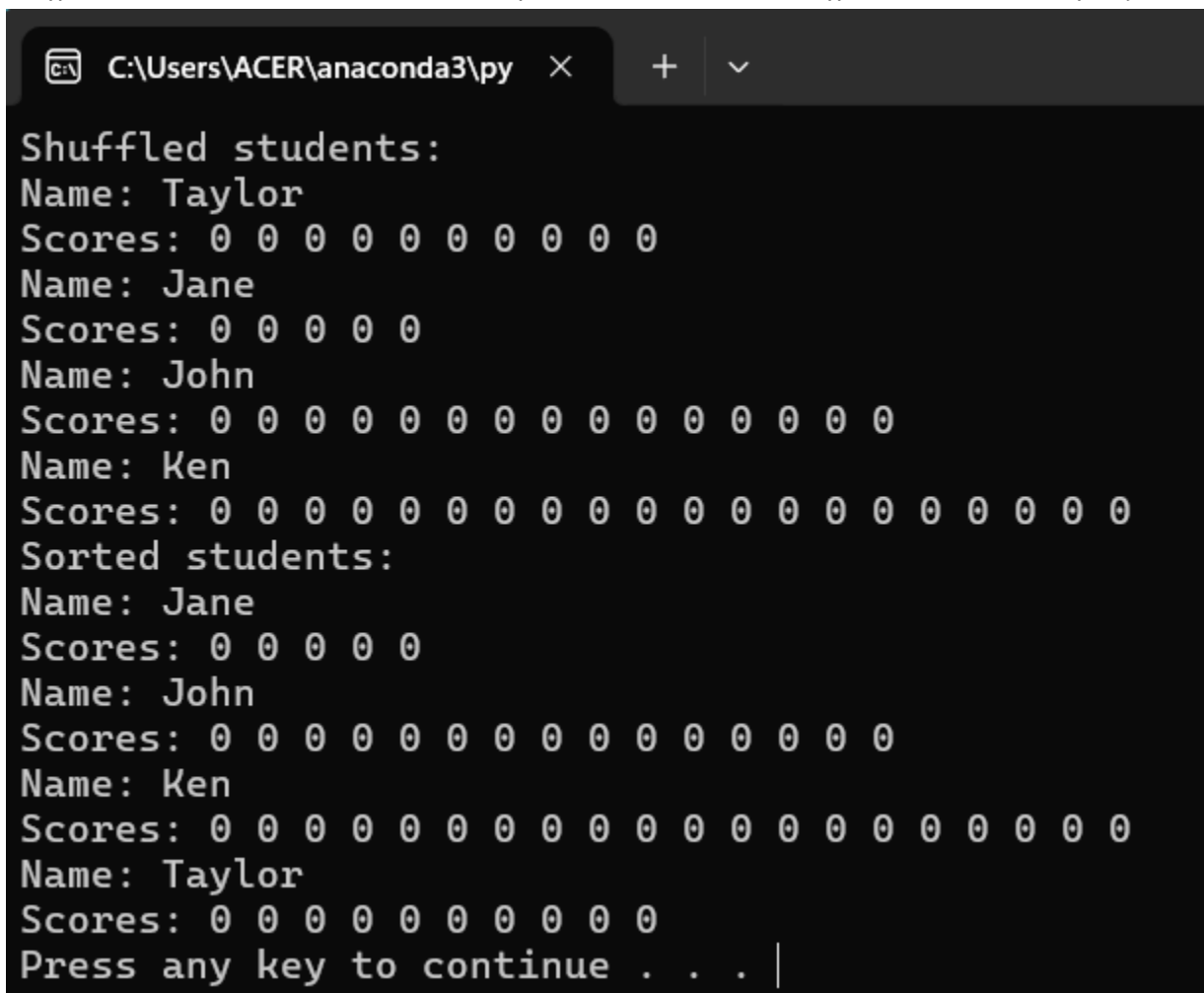


```
student.py  ↗ ✕  
  
import random  
  
student.py*  ↗ ✕  
  
def main():  
    """A simple test."""  
    student1 = Student("Taylor", 10)  
    student2 = Student("Ken", 20)  
    student3 = Student("John", 15)  
    student4 = Student("Jane", 5)  
  
    students = [student1, student2, student3, student4]  
    random.shuffle(students)  
    print("Shuffled students:")  
    for student in students:  
        print(student)  
  
    students.sort()  
    print("Sorted students:")  
    for student in students:  
        print(student)  
  
if __name__ == "__main__":  
    main()
```

Figure 6: The main function that shuffled and sorted a list of students

The code defines the "main" function to manage a list of four students. I then added the required imports for the random module, enabling shuffling. Then, I created a list called students, which contains Student objects. Then it uses random.shuffle() to shuffle the list

and the default sorting behavior to sort it. It creates instances of the Student class with names and numbers, shuffles the list, sorts it based on student names, and prints both the shuffled and sorted lists. This code helps us learn how to shuffle and sort lists of objects.



```
C:\Users\ACER\anaconda3\py × + v

Shuffled students:
Name: Taylor
Scores: 0 0 0 0 0 0 0 0 0 0 0
Name: Jane
Scores: 0 0 0 0 0
Name: John
Scores: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Name: Ken
Scores: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Sorted students:
Name: Jane
Scores: 0 0 0 0 0
Name: John
Scores: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Name: Ken
Scores: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Name: Taylor
Scores: 0 0 0 0 0 0 0 0 0 0
Press any key to continue . . . |
```

Figure 6.2: Output of the program

For Figure 6.2, the output shows and proves that the four names are sorted and shuffled.

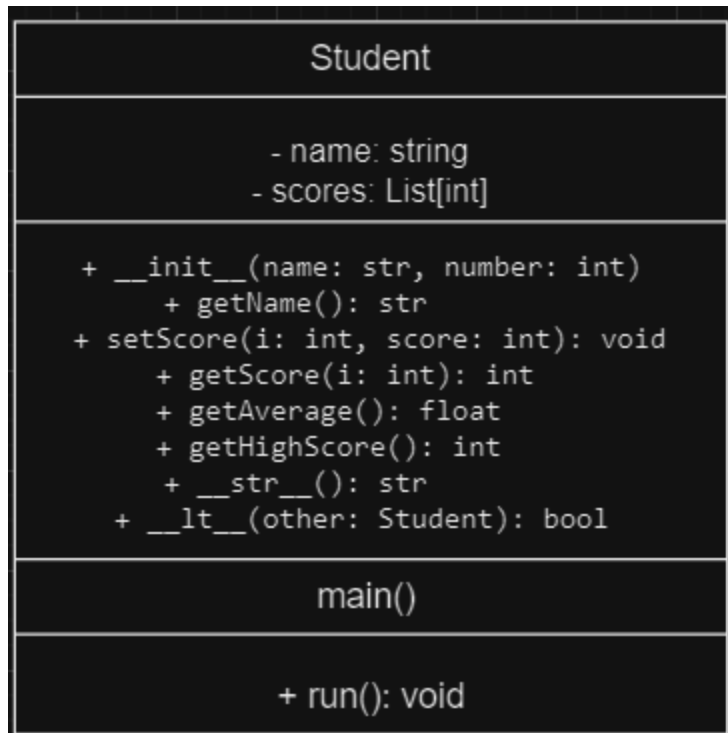


Figure 7: Update UML Diagram with 3 Additional Methods

Figure 7 shows that the student class has been updated, and three methods have been added. The `run()` method interacts with instances of the Student class (`student1`, `student2`, etc.) by creating, shuffling, and sorting them based on their names.

3. The `str` method of the Bank class (in the file `bank.py`) returns a string containing the accounts in random order. Design and implement a change that causes the accounts to be placed in the string by order of name. (Hint: You will also have to define some methods in the SavingsAccount class, in the file `savingsaccount.py`.) (LO: 10.1, 10.2).