MAPÚA UNIVERSITY
SCHOOL OF ELECTRICAL, ELECTRONICS, AND COMPUTER ENGINEERING

Experiment 3: Object Oriented Design and Implementation

CPE106L (Software Design Laboratory)

Danikka Villaron
Jessa Abigaile Tongol
Mark Vincent De Villa
Matthew Benedict Nepomuceno

Group No.: **2**
Section: **E03**

PreLab

Readings, Insights, and Reflection

Insights and Reflections

Systems Analysis and Design: An Object Oriented Approach with UML 5th Edition
ISBN: 9781119561217

De Villa (Chapter 4 : Use Case Analysis)

The fourth chapter of Systems Analysis and Design: An Object-Oriented Approach with UML, 5th Edition, explains and describes the purpose of use case analysis, its components, the process of developing a use case analysis, and its contribution to the development plans and functional requirements. The book first described the use cases as an illustration of the user-system interaction from the user's perspective, specifically, the sequence of activities the system performs when triggered by the user, which is essential in developing the functional requirements of the software. A use case contains Preconditions - the expected state of a system before performing the use case -, the Normal Course - the steps performed by the system when triggered, listing the utilization and production of inputs and outputs utilized by the system -, the Alternative Course - lists the steps that branch out from the normal course -, Postconditions - list and definition of outputs made from the use case courses -, exceptions - steps taken when errors take place, branching out from either the Normal and Alternative Courses -, Summary Inputs and Outputs - listing the significant inputs and outputs required or made by the use case -, and Additional Use Case Issues that may include the Frequency of Use, Business Rules, Special Requirements, Assumptions, and the Notes and Issues in regards to the use case. Furthermore, The development of use cases requires identifying triggering events, the actors, and the steps that would utilize inputs to produce expected outcomes from the event, where the steps mention how and where the utilization of inputs and outputs occur.

Nepomuceno (Chapter 5 : Process Modeling)

Chapter five is about Process Modeling. It explains how DFDs show processes, data flows, data stores, and external entities, providing a clear picture of system functions and data interactions. This chapter mainly focuses on DFDs or Data Flow Algorithms. They show how data moves through a system and how processes, data stores, and external entities connect. It breaks down the system into smaller, understandable parts, showing how each element works together. DFDs visualize processes where data is transformed or handled and map the movement of data between processes, data stores, and external entities like users or other systems. Additionally, they illustrate how data enters and leaves the system through external sources like suppliers or consumers, as well as where it is stored, such as in databases or files. Lastly, DFD gives analysts and designers a high-level perspective of a system's data flow, which aids in understanding and enhancing the system's performance.

Tongol (Chapter 4 & Chapter 5)

An Object-Oriented Approach with UML further discuss the nature of understanding system requirements and its modeling and analyzing cases. These focus on the relevance of clear, detailed communication with stakeholders and analysts to capture their functional and non-functional requirements well. The focus on use case diagrams, scenarios, and activity diagrams underscores the importance of visual tools in bridging conceptual ideas and technical implementation. What I learned about this chapter is the structured approach to identifying actors and use cases, which sets a strong basis for understanding how users interact with the system. The discussions on refining use cases and integrating them into the broader system design were very detailed, and the importance of iterative and collaborative development was reinforced to align with user needs and project objectives.

Fundamentals of Python: First Programs 2nd Edition
ISBN: 9781337560092

Villaron (Chapter 9 : Design with Classes)

In this chapter, I learned about oop in python. OOP is a way to organize our code so that we can group things together. One example in the textbook that creates a program about students, we could create the Student class that holds all the information or data about a student such as their name, age, and scores. There are also methods that can be created contained in a class that do stuff that is intended. Other terms were also encountered such as encapsulation and inheritance. Looking ahead, I think OOP is an efficient way to handle or organize bigger programs such as games or systems that manage data.

Answers to Questions

1. **A**. is owned by a particular instance of a class and no other
2. **C**. Self
3. **B.** Set the instance variables to initial values
4. **B.** Always must have at least one parameter, called self
5. **B.** The entire class in which it is introduces
6. **B.** When it can no longer be referenced anywhere in a program
7. **A.** All instances of a class have in common
8. **B.** __init__(self)
9. **B.** Pickle them using the pickle function dump
10. **A.** Has a single header but different bodies in different classes

InLab

- **Objectives**
  1. Familiarize oneself with Classes and methods declarations with Kenneth Lambert's sample codes
  2. Familiarize oneself with UML creation process and relation to Classes
  3. Familiarize oneself with UMLet and create UML Class diagrams of Kenneth Lambert's sample codes

- **Tools Used**
  - Anaconda
  - Git Terminal
  - Vistual Studio Code
  - UMLet

- **Procedure**

Before starting the InLab activity, understanding the format used on UML charts when presenting the attributes of a Class becomes crucial. Figure 1.1 presents a template utilized when representing a Class on UML charts. The first box in the UML diagram presents the class name, followed by a box containing the variable name and data type. Whereas the third box contains the methods declared within the class.

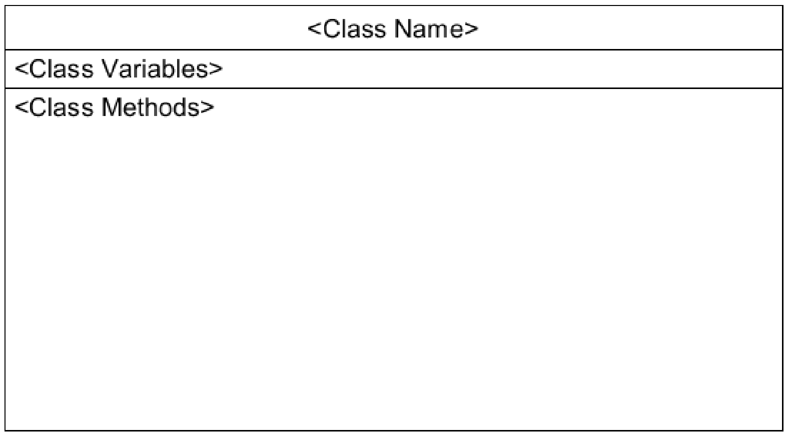
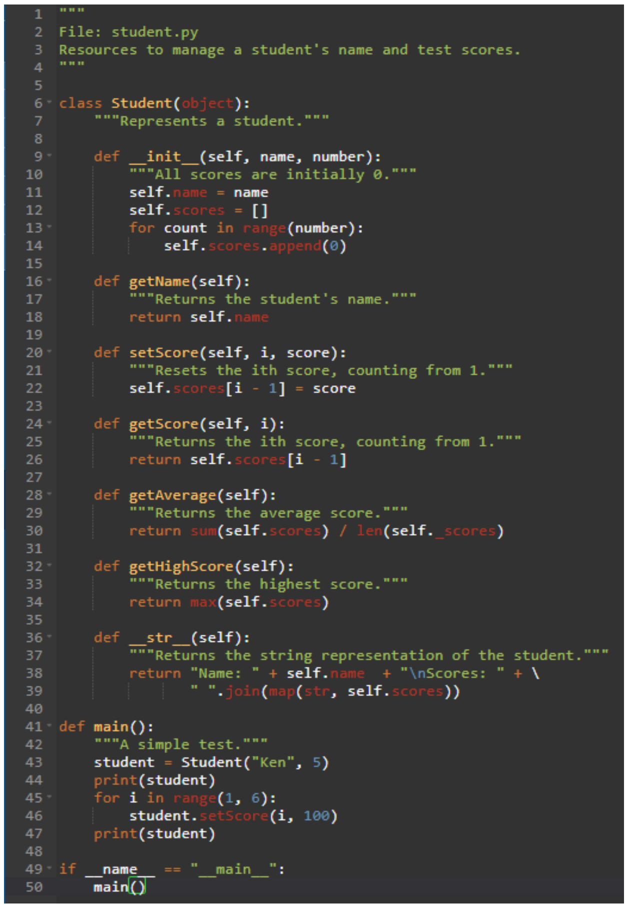


Figure 1.1 UML Class Diagram Format

The InLab activity begins with selecting a program from Kenneth Lambert's Fundamentals of Python: First Programs, 2nd Edition. The link below summarizes the codes found in the program. For example, we selected student.py with its code seen in Figure 1.2.



```
1  """
2  File: student.py
3  Resources to manage a student's name and test scores.
4  """
5
6  class Student(object):
7      """Represents a student."""
8
9      def __init__(self, name, number):
10         """All scores are initially 0."""
11         self.name = name
12         self.scores = []
13         for count in range(number):
14             self.scores.append(0)
15
16     def getName(self):
17         """Returns the student's name."""
18         return self.name
19
20     def setScore(self, i, score):
21         """Resets the ith score, counting from 1."""
22         self.scores[i - 1] = score
23
24     def getScore(self, i):
25         """Returns the ith score, counting from 1."""
26         return self.scores[i - 1]
27
28     def getAverage(self):
29         """Returns the average score."""
30         return sum(self.scores) / len(self._scores)
31
32     def getHighScore(self):
33         """Returns the highest score."""
34         return max(self.scores)
35
36     def __str__(self):
37         """Returns the string representation of the student."""
38         return "Name: " + self.name  + "\nScores: " + \
39                " ".join(map(str, self.scores))
40
41 def main():
42     """A simple test."""
43     student = Student("Ken", 5)
44     print(student)
45     for i in range(1, 6):
46         student.setScore(i, 100)
47     print(student)
48
49 if __name__ == "__main__":
50     main()
```

Figure 1.2. student.py Source Code

After selecting a program, dissect and identify the different components of the Class utilized in the program. For our example, the student.py program declares a Student Class on line 6 of the source code and declares an object on line 43. The first notable detail is on the declaration of Class, where the Class name is found and placed in the UML chart, as seen in Figure 1.2. After locating the class name on the declaration, take the class name and insert it in the first box of the UML class diagram to define the Class that the diagram would represent.

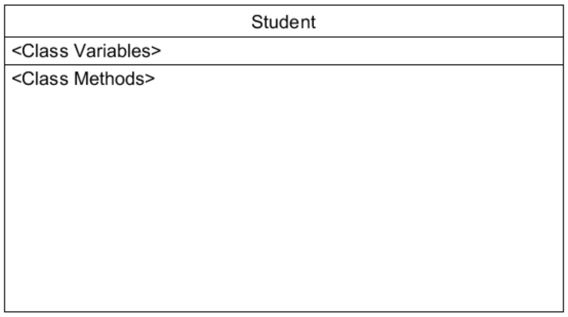
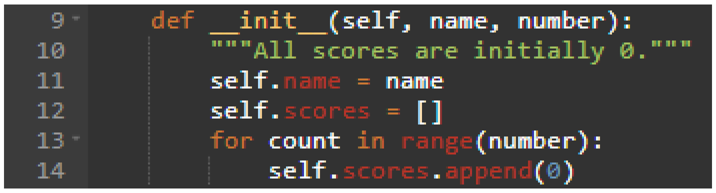


Figure 1.3. Naming the UML Chart based on the Student Class



```
 9    def __init__(self, name, number):
10        """All scores are initially 0."""
11        self.name = name
12        self.scores = []
13        for count in range(number):
14            self.scores.append(0)
```

Figure 1.4. Student Constructor Function

The next step is to identify the variables utilized in the Class. In the Student Class, the constructor found on lines 9 to 14, seen in Figure 1.4, declares two variables: one string represented by the "name" and one array list of float values represented by the "scores." The UML diagram presents these variables with a dash or minus symbol to inform the programmers that the variables are private, followed by the variable name, a colon, and the variable data type. For example, the representation of the name variable in the Class is "-name: str," also seen in Figure 1.5.

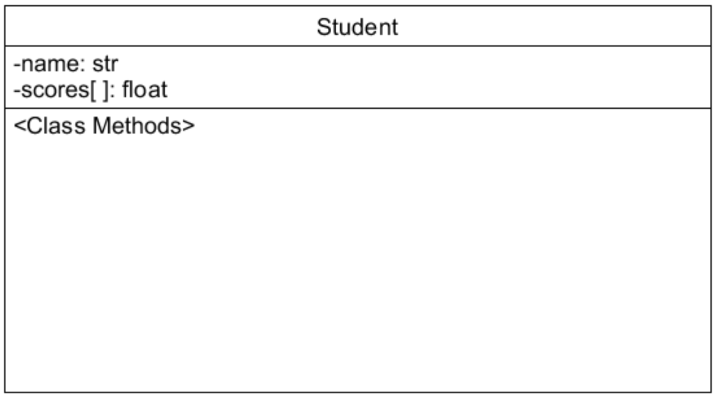
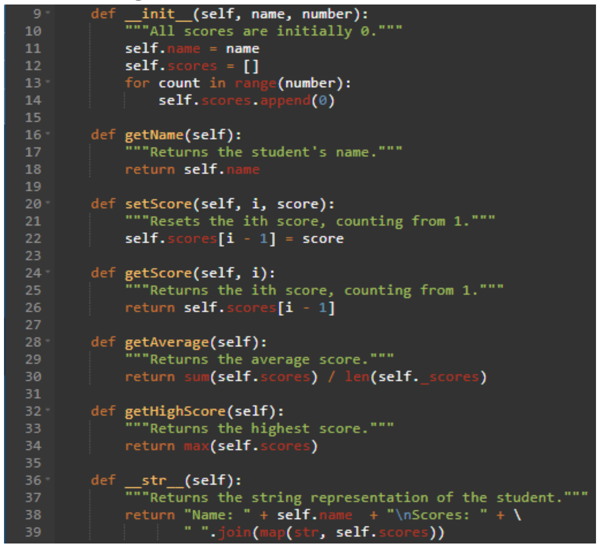


Figure 1.5 Adding of Variables in the Student UML Diagram



```
 9    def __init__(self, name, number):
10        """All scores are initially 0."""
11        self.name = name
12        self.scores = []
13        for count in range(number):
14            self.scores.append(0)
15
16    def getName(self):
17        """Returns the student's name."""
18        return self.name
19
20    def setScore(self, i, score):
21        """Resets the ith score, counting from 1."""
22        self.scores[i - 1] = score
23
24    def getScore(self, i):
25        """Returns the ith score, counting from 1."""
26        return self.scores[i - 1]
27
28    def getAverage(self):
29        """Returns the average score."""
30        return sum(self.scores) / len(self._scores)
31
32    def getHighScore(self):
33        """Returns the highest score."""
34        return max(self.scores)
35
36    def __str__(self):
37        """Returns the string representation of the student."""
38        return "Name: " + self.name  + "\nScores: " + \
39                " ".join(map(str, self.scores))
```

Figure 1.6 Methods Included in Student Class

The final procedure to complete the UML diagram representation of a Class begins by identifying the methods declared in the Class. Figure 1.6 presents all the seven methods part of the Student Class. A method's tell-tale feature includes having parameters and arguments when called.

The representation of methods declared in a Class into the UMLClass Diagrams begins with a plus symbol to signify that the function is public, followed by the method's name and parameters, which also contain the variables and data types called. When the method is a returning function, they have an arrow pointing to the data type of the value it returns, and if the function is void, it will not have the arrow pointing to a data type. Figure 1.7 presents the Student Class methods represented in the UML Class Diagram.

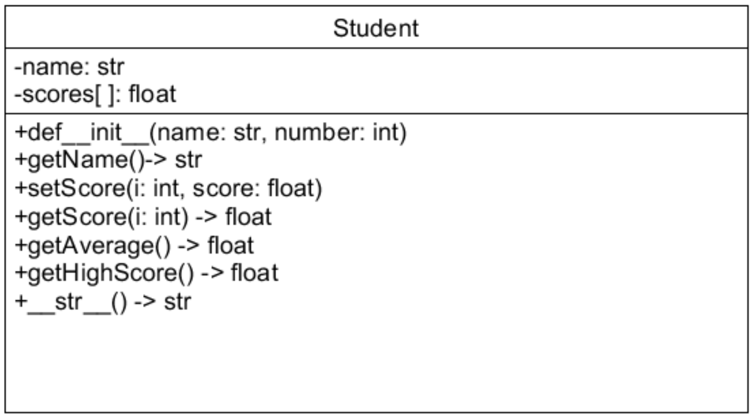
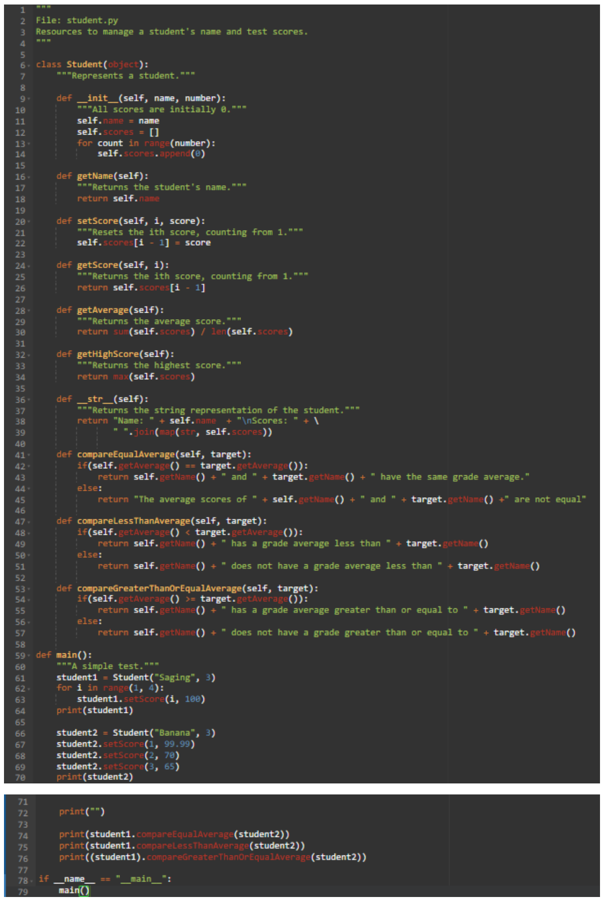


Figure 1.7 Completed UML Diagram of the Student Class

PostLab

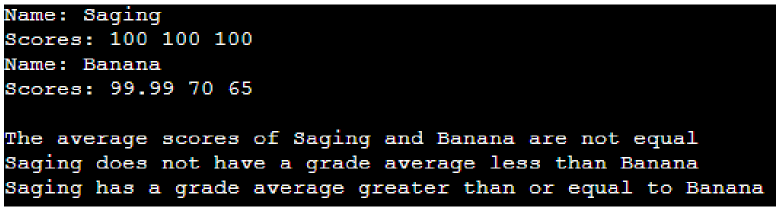
1. **Add three methods to the Student class that compare two Student objects. One method should test for equality. A second method should test for less than. The third method should test for greater than or equal to. In each case, the method returns the result of the comparison of the two students' names.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



```python
"""
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)

    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: " + \
                 " ".join(map(str, self.scores))

    def compareEqualAverage(self, target):
        if(self.getAverage() == target.getAverage()):
            return self.getName() + " and " + target.getName() + " have the same grade average."
        else:
            return "The average scores of " + self.getName() + " and " + target.getName() +" are not equal"

    def compareLessThanAverage(self, target):
        if(self.getAverage() < target.getAverage()):
            return self.getName() + " has a grade average less than " + target.getName()
        else:
            return self.getName() + " does not have a grade average less than " + target.getName()

    def compareGreaterThanOrEqualAverage(self, target):
        if(self.getAverage() >= target.getAverage()):
            return self.getName() + " has a grade average greater than or equal to " + target.getName()
        else:
            return self.getName() + " does not have a grade greater than or equal to " + target.getName()

def main():
    """A simple test."""
    student1 = Student("Saging", 3)
    for i in range(1, 4):
        student1.setScore(i, 100)
    print(student1)

    student2 = Student("Banana", 3)
    student2.setScore(1, 99.99)
    student2.setScore(2, 70)
    student2.setScore(3, 65)
    print(student2)

    print("")

    print(student1.compareEqualAverage(student2))
    print(student1.compareLessThanAverage(student2))
    print((student1).compareGreaterThanOrEqualAverage(student2))

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

Figure 2.1. Modified student.py code

Kenneth Lambert's book, Fundamentals of Python: First Programs 2nd Edition, on pages 296-297, contains a program called student.py, defining a class called Students with six functions and a constructor. Figure 1.1 presents the original code, and Figure 2.1 presents the modified code wherein three comparative functions - equality, less than, greater than, or equal - which also contains the compared student's names on its output along with the comparative statement.



```
Name: Saging
Scores: 100 100 100
Name: Banana
Scores: 99.99 70 65

The average scores of Saging and Banana are not equal
Saging does not have a grade average less than Banana
Saging has a grade average greater than or equal to Banana
```

Figure 2.2. Output of Modified student.py

Figure 2.2 presents the output of the modified code, which begins by printing the name and scores of a student, Saging as student one and Banana as student two, followed by their three predefined grades with the setGrades() function before printing the outputs of the three added functions. The first function tested for the equality of the two students, and since the averages were not equal, the output also mentions how the grades of the two compared students have unequal average scores. The second function compares whether the average of the first student is less than the second student. Since the average of the first student was not less than the second's, the output also mentions that the grade average of the first student is not less than the second's. The final function compares whether the grade average of the first student is greater than or equal to that of the second student. Since the statement was correct, the output statement mentions that student one's average grade is greater than or equal to the second student's average.

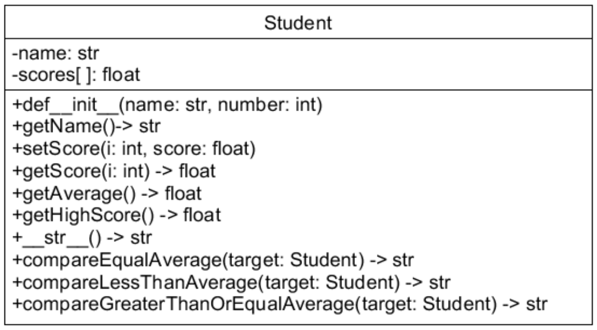
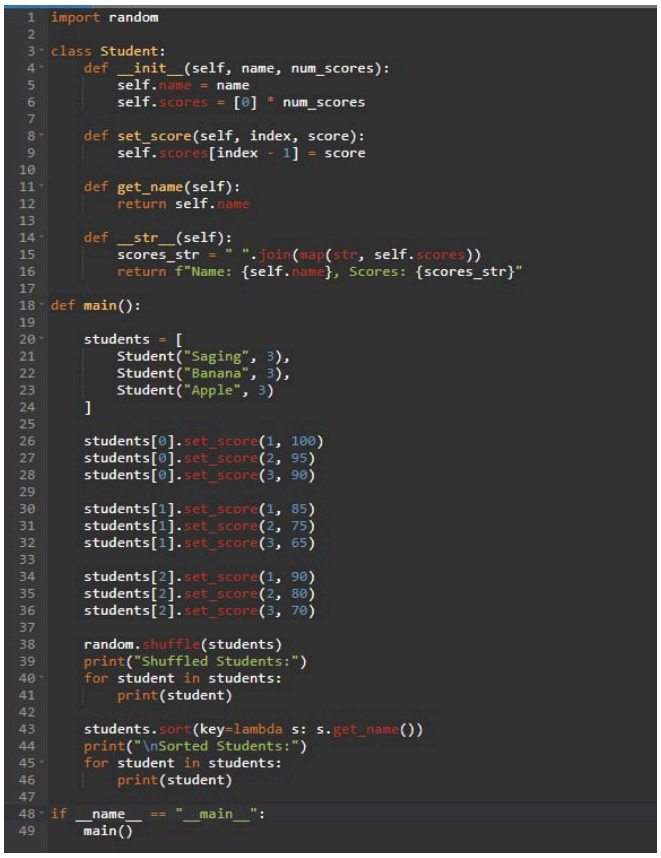

Student
-name: str
-scores[ ]: float
+def__init__(name: str, number: int)
+getName()-> str
+setScore(i: int, score: float)
+getScore(i: int) -> float
+getAverage() -> float
+getHighScore() -> float
+__str__() -> str
+compareEqualAverage(target: Student) -> str
+compareLessThanAverage(target: Student) -> str
+compareGreaterThanOrEqualAverage(target: Student) -> str

Figure 2.3. UML Class Diagram of Student Class

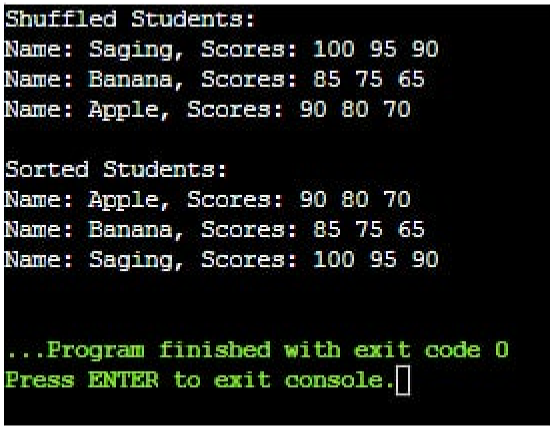
Figure 2.3 presents the UML class diagram of the Student class presented in Figure _.1, wherein only two private variables were present: the name variable, presented as a string, and the scores, presented in an array list of float values. Following the variables are the functions defined in the class. The __init__ function acts as the constructor with a string and integer parameter that declares the object's name and number of grades, respectively. Following the constructor is the getName() function, which returns the object's name as a string and contains no parameters. Right below is the setScore() function, one integer parameter acting as the target list index and the float as the value that will replace the list's value at the given index. The getScore() function only uses one integer parameter, which acts as the target index as the function returns a float value obtained from the list at the given index. The getAverage() function has no parameters, averages the values in the list, and returns a float value - the calculated average. The getHighScore() function utilizes the max() function to find the highest value in the list and returns that float value. The __str__() function executes when printing the object, outputting the student name and the list of float scores as a string. The compareEqualAverage() is the first addition to the original code, with a Student object as a parameter, comparing the average grades of the two objects, and returns a string statement to present its conclusion. The same goes for the LessThanAverage(), containing a Student object as a parameter and returning a string to present its conclusion after comparing the averages of two objects. The final additional function, the compareGreaterThanOrEqualAverage() with a Student object parameter, compares whether the grade averages of two students are greater than or equal to each other before presenting its conclusion as a string.

2. **This project assumes that you have completed Project 1. Place several Student objects into a list and shuffle it. Then run the sort method with this list and display all of the students' information.**



```python
import random

class Student:
    def __init__(self, name, num_scores):
        self.name = name
        self.scores = [0] * num_scores

    def set_score(self, index, score):
        self.scores[index - 1] = score

    def get_name(self):
        return self.name

    def __str__(self):
        scores_str = " ".join(map(str, self.scores))
        return f"Name: {self.name}, Scores: {scores_str}"

def main():

    students = [
        Student("Saging", 3),
        Student("Banana", 3),
        Student("Apple", 3)
    ]

    students[0].set_score(1, 100)
    students[0].set_score(2, 95)
    students[0].set_score(3, 90)

    students[1].set_score(1, 85)
    students[1].set_score(2, 75)
    students[1].set_score(3, 65)

    students[2].set_score(1, 90)
    students[2].set_score(2, 80)
    students[2].set_score(3, 70)

    random.shuffle(students)
    print("Shuffled Students:")
    for student in students:
        print(student)

    students.sort(key=lambda s: s.get_name())
    print("\nSorted Students:")
    for student in students:
        print(student)

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

Figure 3.1



```
Shuffled Students:
Name: Saging, Scores: 100 95 90
Name: Banana, Scores: 85 75 65
Name: Apple, Scores: 90 80 70

Sorted Students:
Name: Apple, Scores: 90 80 70
Name: Banana, Scores: 85 75 65
Name: Saging, Scores: 100 95 90


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 3.2

The code reorders the student list in a random order using random.shuffle() function. The order in the shuffled list (Figure 3.2) is different from the original because this shuffling is entirely random and not based on any specific data of the students. Students.sort(key=lambda s: s.get_name()) is used to sort the list after shuffling, arranging the students according to their names. During the sorting process, This results in the students being arranged in alphabetical order, as shown in the "Sorted Students" section.alphabetically.

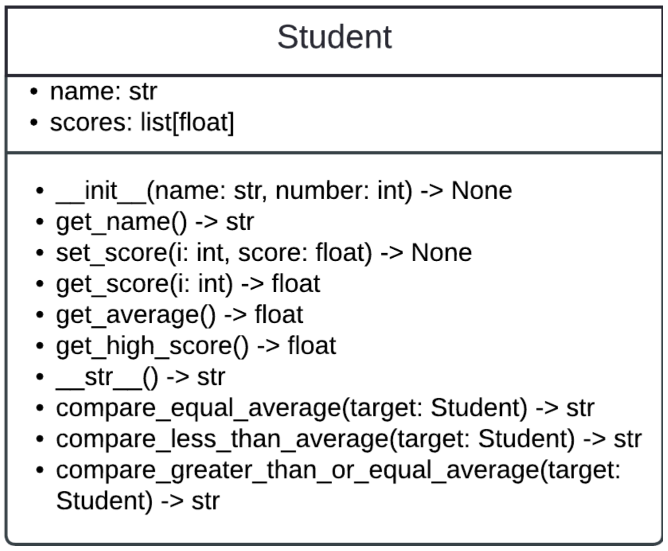
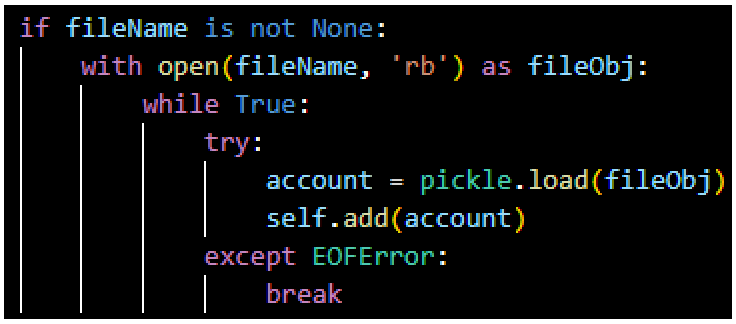

Student
• name: str
• scores: list[float]
• __init__(name: str, number: int) -> None
• get_name() -> str
• set_score(i: int, score: float) -> None
• get_score(i: int) -> float
• get_average() -> float
• get_high_score() -> float
• __str__() -> str
• compare_equal_average(target: Student) -> str
• compare_less_than_average(target: Student) -> str
• compare_greater_than_or_equal_average(target: Student) -> str

Figure 3.3

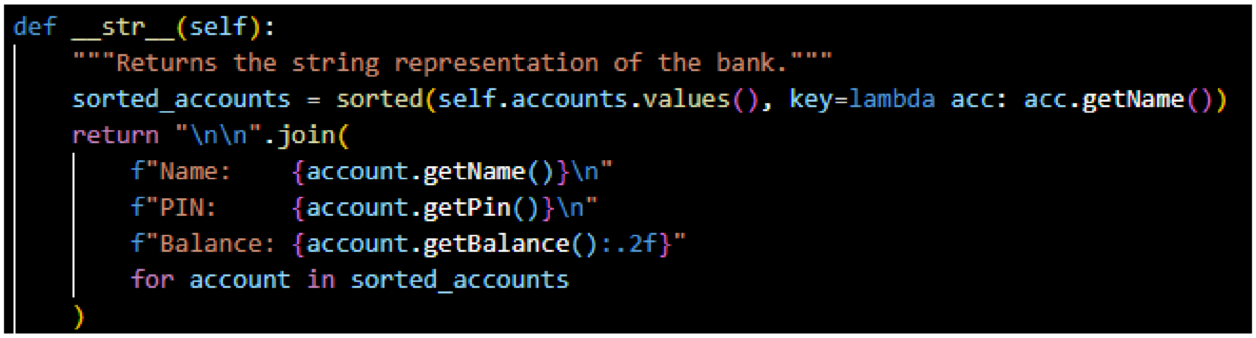
3. **The str method of the Bank class returns a string containing the accounts in random order. Design and implement a change that causes the accounts to be placed in the string by order of name. (Hint: You will also have to define some methods in the SavingsAccount class.) (VILLARON)**



```python
if fileName is not None:
    with open(fileName, 'rb') as fileObj:
        while True:
            try:
                account = pickle.load(fileObj)
                self.add(account)
            except EOFError:
                break
```

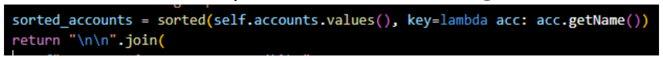
Figure 4.1: Made small adjustments in file handling

I just replaced some terms for better file handling and used a specific error rather than a generic one.



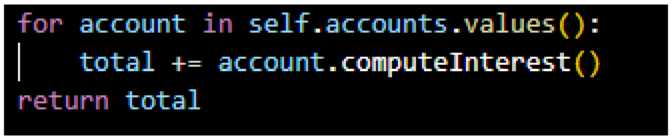
```python
def __str__(self):
    """Returns the string representation of the bank."""
    sorted_accounts = sorted(self.accounts.values(), key=lambda acc: acc.getName())
    return "\n\n".join(
        f"Name:    {account.getName()}\n"
        f"PIN:     {account.getPin()}\n"
        f"Balance: {account.getBalance():.2f}"
        for account in sorted_accounts
    )
```

Figure 4.2: Str Representation

I added a detailed format on how I want the output to be and sorting them in order as required in the problem. This 

```python
sorted_accounts = sorted(self.accounts.values(), key=lambda acc: acc.getName())
return "\n\n".join(
```

does sorting accounts by name using anonymous function key=lambda acc: Afterwhile, I placed a blank space in between accounts for better visual representation

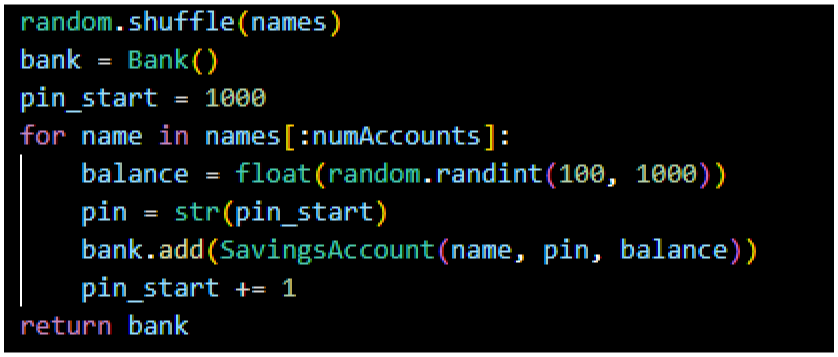


```python
for account in self.accounts.values():
    total += account.computeInterest()
return total
```

Figure 4.3: Fixed minimal punctuation error

.

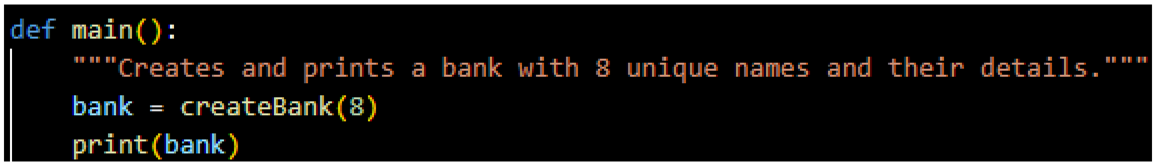
I replaced self._accounts to self.accounts since _accounts didn't exist, at least on my end.



```
random.shuffle(names)
bank = Bank()
pin_start = 1000
for name in names[:numAccounts]:
    balance = float(random.randint(100, 1000))
    pin = str(pin_start)
    bank.add(SavingsAccount(name, pin, balance))
    pin_start += 1
return bank
```

Figure 4.4: Corrected implementation of createBank

The output of the original code displayed repetition of some bank names and details, it just picked random names from the list and created accounts as specified. To make some changes, I used the shuffle function that randomizes the order of the names initially; it doesn't affect the final order in which the accounts are displayed due to the sorted() function in the _str_ method. I picked the first numAccounts names from the shuffled list to get the exact number of accounts I needed without repeating any names. For each name, I assigned a random balance between 100 and 1000 and gave it a unique PIN, starting from pin_start and increasing it for each account. Finally, I added each account, with its name, PIN, and balance, to the bank. This way, I made sure the process was both random and organized.



```
def main():
    """Creates and prints a bank with 8 unique names and their details."""
    bank = createBank(8)
    print(bank)
```

Figure 4.5: Function main

As shown in the original code, there was a testAccount() function to check accounts behaviour, which I think was not necessary. Thus, I just created the 8 accounts needed, displayed them, and removed the unnecessary tests.

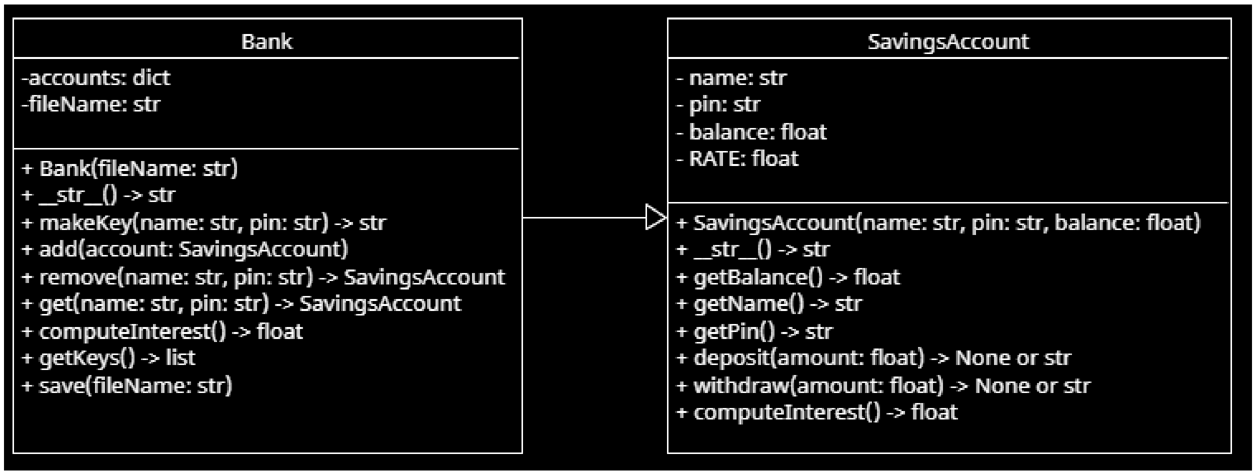
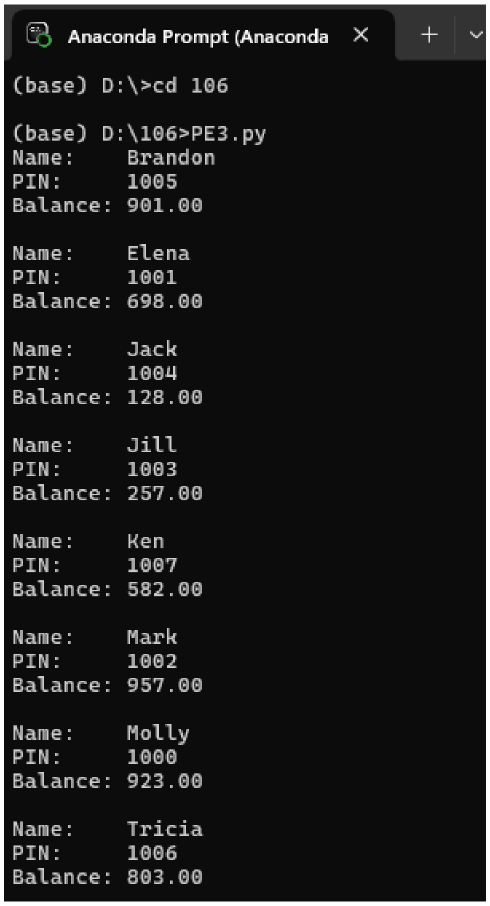

Bank
-accounts: dict
-fileName: str
+ Bank(fileName: str)
+ __str__() -> str
+ makeKey(name: str, pin: str) -> str
+ add(account: SavingsAccount)
+ remove(name: str, pin: str) -> SavingsAccount
+ get(name: str, pin: str) -> SavingsAccount
+ computeInterest() -> float
+ getKeys() -> list
+ save(fileName: str)
SavingsAccount
- name: str
- pin: str
- balance: float
- RATE: float
+ SavingsAccount(name: str, pin: str, balance: float)
+ __str__() -> str
+ getBalance() -> float
+ getName() -> str
+ getPin() -> str
+ deposit(amount: float) -> None or str
+ withdraw(amount: float) -> None or str
+ computeInterest() -> float

Figure 4.6: UML for bank.py and savingsaccount.py

The relationship between bank.py and savingsaccounts.py is called aggregation which means bank is like a container or “has” stuff for savingsaccount. Bank manages savingsaccount, in short.

OUTPUT:



```
(base) D:\>cd 106

(base) D:\106>PE3.py
Name:    Brandon
PIN:     1005
Balance: 901.00

Name:    Elena
PIN:     1001
Balance: 698.00

Name:    Jack
PIN:     1004
Balance: 128.00

Name:    Jill
PIN:     1003
Balance: 257.00

Name:    Ken
PIN:     1007
Balance: 582.00

Name:    Mark
PIN:     1002
Balance: 957.00

Name:    Molly
PIN:     1000
Balance: 923.00

Name:    Tricia
PIN:     1006
Balance: 803.00
```

Figure 4.7: Program execution in Anaconda prompt
Output of the modified program bank.py