\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSC121 PYTHON Programming**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lesson 12 Object-Oriented Programming [Part 1]

# **Objectives**

In this lesson, students will learn:

- How to design objects

- How to define classes

- How to create and use objects

- How to control accessibility to class members

# **12.1 What is Object Oriented Programming?**

Object-oriented programming (OOP) is a programming language model organized around objects rather than steps. Historically, a program has been viewed as a set of steps that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the steps, not how to define and organize the data. Object-oriented programming takes the view that what we really care about are the data we want to manipulate rather than the steps required to manipulate them.

The primary focus of a program using object-oriented design is the objects. Objects contain data, in the form of fields, often known as attributes; and code, in the form of functions, often known as methods. A feature of objects is that an object's method can access and often modify the data fields of the object with which they are associated. In an earlier lesson we learned what objects are and how they are used. In this lesson, we will learn how to design and create objects.

# **12.2 Designing Objects**

When we design a program with OOP, the first step is to identify all the objects we want to manipulate and how they relate to each other, an exercise often known as data modeling.

Once the objects are identified, the next step is to design the details of each type of objects. The design of a type of objects is called a **class**. A class specifies the data attributes (i.e. variables) and methods (i.e. functions) that are bundled together for defining a type of objects. A class, therefore, is a “cookie cutter” that can be used to make as many instances of that type of objects as needed.

In the software industry there is a standard way to specify object oriented design. The **Unified Modeling Language (UML)** is a language-independent, graphical standardized design specification (modeling) language for specifying an object-oriented design. Although it has the word “language” in its name, UML is nothing about writing program code. Instead, it contains numerous types of graphical diagrams for expressing various aspects of an object-oriented design. One of the most widely used graphical diagrams is called **class diagrams**.

In UML, class diagrams are used to express the static aspects of a design, such as the data attributes and methods of individual classes. A class diagram has the following format:

| **Class Name** |
| --- |
| data attributes |
| methods |

The following is an example.

| **Student** |
| --- |
| +name: String  +project: Float  +midterm: Float  +final: Float |
| +Student(student\_name: String)  +inputScores()  +calcAvg(): Float |

The class diagram above specifies a class Student. This class has four **instance variables**: a string name and three floating point numbers project, midterm and final. Instance variables are one kind of data attributes. We will explain more about instance variables later. The “+” signs before the variable names indicate accessibility of these fields, which will also be explained later.

There are three methods in the Student class. The Student method is a **constructor**, which is executed when an object of this class is created. Constructor is typically used to create and initialize instance variables, and do other preparation for the object. Different programming languages have different rules to name constructors. In Python, constructors must be named \_\_init\_\_. Since UML is language-independent, the generic name create is used in class diagrams for constructors. The constructor of Student class has one string-type parameter student\_name. That means whenever we create a new Student object, we must provide the name of the student to the constructor.

The Student class also has an inputScores method and a calcAvg method. The inputScores method has no parameter and no return value. The calcAvg method also has no parameter but it returns a floating point number.

# **12.3 Defining Classes in Python**

Once we have the specification of a class, the next step is to write code to define the class. The following is the code for class Student.

**class** Student:  
  
 **def** \_\_init\_\_(self, student\_name):  
  
 *""" constructor of class Student """* self.name = student\_name  
 self.project = 0.0  
 self.midterm = 0.0  
 self.final = 0.0

**def** inputScores(self):  
  
 *""" input scores from user """* self.project = float(input(**"Enter project score: "**))  
 self.midterm = float(input(**"Enter midterm score: "**))  
 self.final = float(input(**"Enter final score: "**))

**def** calcAvg(self):  
  
 *""" return average of project, midterm and final """* avg = (self.project + self.midterm + self.final)/3  
 **return** avg

The keyword class in the first line indicates that we are defining a class. Student is the name of this class. The convention is to capitalize the first letter of the class name.

Next, we define the methods of this class. The name of the first one is \_\_init\_\_ (note: there are two underscore characters both before and after “init”). It is a constructor, i.e., a special method the computer executes automatically when a new instance (i.e. a new object) of a class is created. Constructors are mainly used to create and initialize instance variables. This constructor has two parameters. The first parameter, self, is required in all methods. It is used to reference the object itself. When a method is called, there is no need to pass an argument to this parameter. The memory address of the object itself is passed to self automatically. In addition to self, you can choose to have as many parameters as you want, or no parameter at all, depending on how much data you want the new object receives when the object is created. In this constructor, we have a string-type parameter student\_name.

In addition to the docstring, there are four lines of code in the constructor. These four lines of code create and initialize four variables: name, project, midterm and final. These variables are called **instance variables**. They belong to the whole object rather than just a single method. Any method defined in class Student can access these variables directly. You will actually see that later. The following statement creates an instance variable name and copies the string from student\_name to it:

self.name = student\_name **# name is an instance variable**

We add “self.” before name to indicate that name is an instance variable. This is how we tell the Python interpreter that name is an instance variable. If we did not put “self.” there, name would have been just a local variable in this method:

name = student\_name **# name is a local variable in this method**

What is the difference between an instance variable and a local variable? An instance variable is accessible to all methods in this class, which is exactly what we want for the variable name. A local variable is accessible only in the method where it is created. In this case, it will make the variable name pretty useless.

The next three statements create and initialize three more instance variables: project, midterm and final. The convention is to initialize numerical variables to 0 and string variable to null string (i.e. a string with no characters). You can assign a different initial value to a variable if there is a reason to do so.

Next, we define the inputScores method. The purpose of this method is to input scores from the user. In each statement, we get user input, convert it to float type and assign it to an instance variable. There is no return statement in this method because we do not return anything to the client code (i.e. the statement that calls this method). Once the input values are stored in the instance variables, our goal has achieved. There is nothing else we need to do in this method.

The next method we define is the calAvg method. We have one statement to calculate the average of project, midterm and final scores. We also have a return statement to return the average to the client code. Notice that there is no “self.” before the name avg. That means avg is just a local variable instead of an instance variable. It is created and used only in this calAvg method. Since we do not access this variable from other methods in this class, there is no need to make it an instance variable.

# **12.4 Creating and Using Objects**

Once the class Student is defined, we can use it like a cookie-cutter to create Student objects. Each object is called an **instance** of the class Student. The following is the code.

**class** Student:  
  
 **def** \_\_init\_\_(self, student\_name):  
  
 *""" constructor of class Student """* self.name = student\_name  
 self.project = 0.0  
 self.midterm = 0.0  
 self.final = 0.0

**def** inputScores(self):  
  
 *""" input scores from user """* self.project = float(input(**"Enter project score: "**))  
 self.midterm = float(input(**"Enter midterm score: "**))  
 self.final = float(input(**"Enter final score: "**))

**def** calcAvg(self):  
  
 *""" return average of project, midterm and final """* avg = (self.project + self.midterm + self.final)/3  
 **return** avg  
  
  
**def** main():  
  
 name = input(**"Enter student name: "**)  
 student1 = Student(name)  
 student1.inputScores()  
 average = student1.calcAvg()  
 print(**"Average score of this student:"**, average)  
  
 name = input(**"Enter student name: "**)  
 student2 = Student(name)  
 student2.inputScores()  
 average = student2.calcAvg()  
 print(**"Average score of this student:"**, average)  
  
main()

The first part of this program is the definition of class Student. Then we write the main function of this program, which uses the class definition to create two instances of Student. First, we ask the user to enter the name of the first student. Then we use the following statement to create a Student object and assign it to the variable student1:

student1 = Student(name)

The right hand side of this statement calls the constructor, i.e., the \_\_init\_\_ method, of class Student to create an instance of Student. Suppose the user enters the name ‘Allen Jones’ in the previous statement, the string ‘Allen Jones’ is stored in the variable name and passed to the constructor of class Student as an argument. When the constructor \_\_init\_\_ is executed, four instance variables are created and initialized. Therefore, the object student1 has four instance variables name, project, midterm and final, which store ‘Allen Jones’, 0, 0, and 0, respectively.

The next statement in the main function calls the inputScores method of student1. Statements in this method get scores from the user and store the scores in the instance variables project, midterm and final of student1.

The next statement in the main function calls the calAvg method of student1, which calculates the average of the scores stored in the instance variables project, midterm and final of student1. The calculated average score is returned and stored in the variable average, which is then displayed by the next statement.

The next set of statements in the main function creates and uses another Student object named student2. The average score of this second student is calculated and displayed.

The following is a sample test run of the program:

*Enter student name: Allen Jones*

*Enter project score: 91*

*Enter midterm score: 84*

*Enter final score: 88*

*Average score of this student: 87.66666666666667*

*Enter student name: Ben Decker*

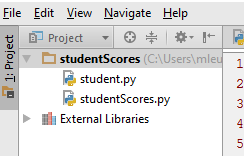
*Enter project score: 86*

*Enter midterm score: 90*

*Enter final score: 94*

*Average score of this student: 90.0*

Before we continue, let’s reorganize the code of this program a little. When we use object oriented programming, the convention is to store the definition of each class in a separate file. Let’s create a project studentScores, which includes two files: student.py and studentScores.py.



The Python file student.py contains the definition of the Student class:

**class** Student:  
  
 **def** \_\_init\_\_(self, student\_name):  
  
 *""" constructor of class Student """* self.name = student\_name  
 self.project = 0.0  
 self.midterm = 0.0  
 self.final = 0.0  
  
  
 **def** inputScores(self):  
  
 *""" input scores from user """* self.project = float(input(**"Enter project score: "**))  
 self.midterm = float(input(**"Enter midterm score: "**))  
 self.final = float(input(**"Enter final score: "**))  
  
  
 **def** calcAvg(self):  
  
 *""" return average of project, midterm and final """* avg = (self.project + self.midterm + self.final)/3  
 **return** avg

We have actually created a module named student (because the file name is student.py) and put class Student in this module.

The Python file studentScores.py contains the main module that creates and uses Student objects:

**from** student **import** Student  
  
**def** main():  
  
 name = input(**"Enter student name: "**)  
 student1 = Student(name)  
 student1.inputScores()  
 average = student1.calcAvg()  
 print(**"Average score of this student:"**, average)  
  
 name = input(**"Enter student name: "**)  
 student2 = Student(name)  
 student2.inputScores()  
 average = student2.calcAvg()  
 print(**"Average score of this student:"**, average)  
  
main()

Since class Student is defined in the module student, we need to import it into this file. We use the syntax **from** student **import** Student so we do not need to include the namespace student when we use Student class. If we use the syntax **import** student to import the student module, we have to use the fully qualified student.Student. That means the following statement:

student1 = Student(name)

must be written as:

student1 = student.Student(name)

# **12.5 The Bank Account Example**

Let’s look at another example. We are writing a program to manage bank accounts. First of all, we will create an Account class, which has three instance variables: name, acct\_num and balance. Arguments will be passed to the constructor and stored in these instance variables. Write a deposit method to deposit money and a withdrawal method to withdraw money.

In the main module, ask the user to enter name, account number and initial balance. Create a bank account with the information entered. Write a loop to allow the user deposit and withdraw money repeatedly. Every time the account balance changes, display the new balance.

Let’s start with designing the class Account. The following is the class diagram.

| **Account** |
| --- |
| +name: String  +acct\_num: String  +balance: Float |
| +Account(name: String, acct\_num: String, balance: Float)  +deposit()  +withdraw() |

The following is the Python code of the class Account:

**class** Account:  
  
 **def** \_\_init\_\_(self, name, acct\_num, balance):  
  
 *""" constructor """* self.name = name  
 self.acct\_num = acct\_num  
 self.balance = balance

**def** deposit(self, amt):  
  
 *""" deposit money in the account """* self.balance = self.balance + amt

**def** withdraw(self, amt):  
  
 *""" withdraw money from the account """* **if** amt > self.balance:  
 print(**"Withdrawal denied due to insufficient fund."**)  
 **else**:  
 self.balance = self.balance - amt

This class has three methods. The constructor creates and initializes three instance variables: name, acct\_num and balance. The deposit method increases the balance. The withdraw method decreases the balance. If the withdrawal amount is higher than the balance, the withdrawal is refused.

The following is the Python code of the main module.

**from** account **import** Account  
  
**def** main():  
 acct\_name = input(**"Enter name: "**)  
 acct\_num = input(**"Enter account number: "**)

balance = float(input(**"Enter initial balance: "**))  
 acct1 = Account(acct\_name, acct\_num, balance)  
 print(**"Balance:"**, acct1.balance)  
 oper = 0  
 **while** oper != 3:  
 oper = int(input(**"Enter 1 for deposit, 2 for withdrawal, 3 for exit: "**))  
 **if** oper == 1:  
 amount = int(input(**"Enter deposit amount: "**))  
 acct1.deposit(amount)  
 print(**"Balance:"**, acct1.balance)  
 **elif** oper == 2:  
 amount = int(input(**"Enter withdrawal amount: "**))  
 acct1.withdraw(amount)  
 print(**"Balance:"**, acct1.balance)  
  
main()

The user is asked to enter a name and an account number, which are used to create a new account. A loop is then used for the user to deposit and withdraw money. The following is a sample test run:

Enter name: Allen Johnson

Enter account number: 123-4-5678

Enter initial balance: 200

Balance: 200.0

Enter 1 for deposit, 2 for withdrawal, 3 for exit: 1

Enter deposit amount: 150

Balance: 350.0

Enter 1 for deposit, 2 for withdrawal, 3 for exit: 2

Enter withdrawal amount: 100

Balance: 250.0

Enter 1 for deposit, 2 for withdrawal, 3 for exit: 2

Enter withdrawal amount: 400

Withdrawal denied due to insufficient fund.

Balance: 250.0

Enter 1 for deposit, 2 for withdrawal, 3 for exit: 3

In object oriented programming, it is quite common to make the constructor more flexible so objects can be created with partial data. This can be done in Python by using default parameters.

We are going to modify the \_\_init\_\_ method of the Account class so that a bank account can be created without initial balance. In that case, the balance is set to 0.

**class** Account:  
  
 **def** \_\_init\_\_(self, name, acct\_num, balance = 0):  
  
 *""" constructor """* self.name = name  
 self.acct\_num = acct\_num  
 self.balance = balance

**def** deposit(self, amt):  
  
 *""" deposit money in the account """* self.balance = self.balance + amt

**def** withdraw(self, amt):  
  
 *""" withdraw money from the account """* **if** amt > self.balance:  
 print(**"Withdrawal denied due to insufficient fund."**)  
 **else**:  
 self.balance = self.balance - amt

To test the modified Account class, let’s make some changes to the main module. We are going to create two accounts, one with an initial balance and one without. The account with no initial balance passed to constructor will have balance set to 0 when the Account object is created.

**from** account **import** Account  
  
**def** main():  
 acct\_name = input(**"Enter name: "**)  
 acct\_num = input(**"Enter account number: "**)  
 balance = float(input(**"Enter initial balance: "**))  
 acct1 = Account(acct\_name, acct\_num, balance)  
 print(**"Balance:"**, acct1.balance)  
 oper = 0  
 **while** oper != 3:  
 oper = int(input(**"Enter 1 for deposit, 2 for withdrawal, 3 for exit: "**))  
 **if** oper == 1:  
 amount = int(input(**"Enter deposit amount: "**))  
 acct1.deposit(amount)  
 print(**"Balance:"**, acct1.balance)  
 **elif** oper == 2:  
 amount = int(input(**"Enter withdrawal amount: "**))  
 acct1.withdraw(amount)  
 print(**"Balance:"**, acct1.balance)  
  
 acct\_name = input(**"Enter name: "**)  
 acct\_num = input(**"Enter account number: "**)  
 acct2 = Account(acct\_name, acct\_num)  
 print(**"Balance:"**, acct2.balance)  
 oper = 0  
 **while** oper != 3:  
 oper = int(input(**"Enter 1 for deposit, 2 for withdrawal, 3 for exit: "**))  
 **if** oper == 1:  
 amount = int(input(**"Enter deposit amount: "**))  
 acct2.deposit(amount)  
 print(**"Balance:"**, acct2.balance)  
 **elif** oper == 2:  
 amount = int(input(**"Enter withdrawal amount: "**))  
 acct2.withdraw(amount)  
 print(**"Balance:"**, acct2.balance)  
  
main()

The following is a sample test run. The balance of the second account is set to 0 by default.

Enter name: Allen Johnson

Enter account number: 123-4-5678

Enter initial balance: 200

Balance: 200.0

Enter 1 for deposit, 2 for withdrawal, 3 for exit: 3

Enter name: Ben Davis

Enter account number: 987-6-5432

Balance: 0

Enter 1 for deposit, 2 for withdrawal, 3 for exit: 3

# **12.6 Accessibility of Class Members**

By default, all instance variables and methods of an object are publicly accessible. That means client code (e.g. code in the main module) can access any instance variable and call any method of the object. Sometimes it may be a bad idea to make class members, especially instance variables, that wide open. Python allows us to block the access to any instance variable or method if we want to. In fact, it is a convention to make all instance variables private to protect their integrity.

Let’s look at the bank account example for a second time. Logically the only way the balance of an account should change after the account is created is either through deposit or withdrawal. However, since the instance variable balance is public by default, the client code can change balance directly:

**def** main():  
 acct\_name = input(**"Enter name: "**)  
 acct\_num = input(**"Enter account number: "**)  
 balance = float(input(**"Enter initial balance: "**))  
 acct1 = Account(acct\_name, acct\_num, balance)  
 acct1.balance = 500 # set balance to 500  
 print(**"Balance:"**, acct1.balance)

This is undesirable because it makes debugging harder and it goes against the logic of changing account balance only through deposit and withdrawal. To block access to the instance variable balance, we need to make it private. Different programming languages have different syntax to make a class member private. In Python, the syntax is adding two underscore characters at the beginning of the variable’s name. In other words, by change the variable name balance to \_\_balance, the accessibility of this instance variable is changed from public to private. The following is the modified code of the class Account:

**class** Account:  
  
 **def** \_\_init\_\_(self, name, acct\_num, balance):  
  
 *""" constructor """* self.\_\_name = name  
 self.\_\_acct\_num = acct\_num  
 self.\_\_balance = balance

**def** deposit(self, amt):  
  
 *""" deposit money in the account """* self.\_\_balance = self.\_\_balance + amt

**def** withdraw(self, amt):  
  
 *""" withdraw money from the account """* **if** amt > self.\_\_balance:  
 print(**"Withdrawal denied due to insufficient fund."**)  
 **else**:  
 self.\_\_balance = self.\_\_balance – amt

All instance variables are changed from public to private. Their names are changed to \_\_name, \_\_acct\_num and \_\_balance. The two underscore characters at the beginning of the names make these variables private. Client code cannot directly access these private instance variables.

If there is a need for client code to access a private instance variable, we can add methods to the class to do that. There are two types of methods that are specially designed to access private instance variables. The first type is called **getter**. They are used to retrieve the value of a private instance variable. For instance, in the class Account, we can add a getter method to retrieve the value stored in \_\_balance:

**class** Account:  
  
 **def** \_\_init\_\_(self, name, acct\_num, balance):  
  
 *""" constructor """* self.\_\_name = name  
 self.\_\_acct\_num = acct\_num  
 self.\_\_balance = balance

**def** deposit(self, amt):  
  
 *""" deposit money in the account """* self.\_\_balance = self.\_\_balance + amt

**def** withdraw(self, amt):  
  
 *""" withdraw money from the account """* **if** amt > self.\_\_balance:  
 print(**"Withdrawal denied due to insufficient fund."**)  
 **else**:  
 self.\_\_balance = self.\_\_balance - amt

**def** getBalance(self):  
 **return** self.\_\_balance

The method getBalance is a getter for the private instance variable \_\_balance. The convention is to start the name of the method with “get”, followed by the name of the instance variable it is designed to access.

The following is the modified class diagram of class Account. The “-“ sign before the instance variables indicate that they are private.

| **Account** |
| --- |
| -name: String  -acct\_num: String  -balance: Float |
| +Account(name: String, acct\_num: String, balance: Float)  +deposit()  +withdraw()  +getBalance(): Float |

The following is the modified code of the main module. Whenever we need to display account balance, we use the getBalance method to retrieve the information.

**from** account **import** Account  
  
**def** main():  
 acct\_name = input(**"Enter name: "**)  
 acct\_num = input(**"Enter account number: "**)  
 balance = float(input(**"Enter initial balance: "**))  
 acct1 = Account(acct\_name, acct\_num, balance)  
  
 oper = 0  
 **while** oper != 3:  
 oper = int(input(**"Enter 1 for deposit, 2 for withdrawal, 3 for exit: "**))  
 **if** oper == 1:  
 amount = int(input(**"Enter deposit amount: "**))  
 acct1.deposit(amount)  
 print(**"Balance:"**, acct1.getBalance())  
 **elif** oper == 2:  
 amount = int(input(**"Enter withdrawal amount: "**))  
 acct1.withdraw(amount)  
 print(**"Balance:"**, acct1.getBalance())  
  
main()

We can also write methods to change the value of a private instance variable. These methods are commonly known as **setters**. The following is an example.

We are modifying the student scores example we saw earlier. If a student’s final exam score is more than 10 points higher than his midterm exam score, the student will receive 5 extra points in the final. We add getters for \_\_midterm and \_\_final and setter for \_\_final in the Student class.

**class** Student:  
  
 **def** \_\_init\_\_(self, student\_name):  
  
 *""" constructor of class Student """* self.\_\_name = student\_name  
 self.\_\_project = 0.0  
 self.\_\_midterm = 0.0  
 self.\_\_final = 0.0  
  
  
 **def** inputScores(self):  
  
 *""" input scores from user """* self.\_\_project = float(input(**"Enter project score: "**))  
 self.\_\_midterm = float(input(**"Enter midterm score: "**))  
 self.\_\_final = float(input(**"Enter final score: "**))  
  
  
 **def** calcAvg(self):  
  
 *""" return average of project, midterm and final """* avg = (self.\_\_project + self.\_\_midterm + self.\_\_final)/3  
 **return** avg  
  
  
 **def** getMidterm(self):  
 **return** self.\_\_midterm  
  
  
 **def** getFinal(self):  
 **return** self.\_\_final  
  
  
 **def** setFinal(self, final\_score):  
 self.\_\_final = final\_score

The setter setFinal receives an argument and stores its value in the private instance variable \_\_final.

The following is modified class diagram of class Student.

| **Student** |
| --- |
| -name: String  -project: Float  -midterm: Float  -final: Float |
| +Student(student\_name: String)  +inputScores()  +calcAvg(): Float  +getMidterm(): Float  +getFinal(): Float  +setFinal(final\_score: Float) |

The following is the modified main module:

**from** student **import** Student  
  
**def** main():  
  
 name = input(**"Enter student name: "**)  
 student1 = Student(name)  
 student1.inputScores()  
 midterm = student1.getMidterm()  
 final = student1.getFinal()  
 **if** final - midterm > 10:  
 print(**"Improvement target met. Five points added to final exam score."**)  
 student1.setFinal(final + 5)  
 print(**"New final exam score:"**, student1.getFinal())  
 average = student1.calcAvg()  
 print(**"Average score of this student:"**, average)  
  
 print()  
 name = input(**"Enter student name: "**)  
 student2 = Student(name)  
 student2.inputScores()  
 midterm = student2.getMidterm()  
 final = student2.getFinal()  
 **if** final - midterm > 10:  
 print(**"Improvement target met. Five points added to final exam score."**)  
 student2.setFinal(final + 5)  
 print(**"New final exam score:"**, student2.getFinal())  
 average = student2.calcAvg()  
 print(**"Average score of this student:"**, average)  
  
main()

After the scores are entered, the improvement from midterm to final is checked. If it is more than 10 points, five extra points are added to the final score. The following is a sample test run:

Enter student name: Allen Johnson

Enter project score: 95

Enter midterm score: 82

Enter final score: 93

Improvement target met. Five points added to final exam score.

New final exam score: 98.0

Average score of this student: 91.66666666666667

Enter student name: Ben Morris

Enter project score: 91

Enter midterm score: 78

Enter final score: 96

Improvement target met. Five points added to final exam score.

New final exam score: 101.0

Average score of this student: 90.0

We can do more than just copying value to a private instance variable in a setter. Sometimes programmers add code in setters to limit what values can be stored in the private instance variables. Suppose we want to limit the final exam score to the range of 0 to 100, we can add statements in the setFinal method to do that:

**def** setFinal(self, final\_score):  
 **if** final\_score < 0:  
 final\_score = 0  
 **elif** final\_score > 100:  
 final\_score = 100  
 self.\_\_final = final\_score

The following is the modified definition of class Student:

**class** Student:  
  
 **def** \_\_init\_\_(self, student\_name):  
  
 *""" constructor of class Student """* self.\_\_name = student\_name  
 self.\_\_project = 0.0  
 self.\_\_midterm = 0.0  
 self.\_\_final = 0.0  
  
  
 **def** inputScores(self):  
  
 *""" input scores from user """* self.\_\_project = float(input(**"Enter project score: "**))  
 self.\_\_midterm = float(input(**"Enter midterm score: "**))  
 self.\_\_final = float(input(**"Enter final score: "**))  
  
  
 **def** calcAvg(self):  
  
 *""" return average of project, midterm and final """* avg = (self.\_\_project + self.\_\_midterm + self.\_\_final)/3  
 **return** avg  
  
  
 **def** getMidterm(self):  
 **return** self.\_\_midterm  
  
  
 **def** getFinal(self):  
 **return** self.\_\_final  
  
  
 **def** setFinal(self, final\_score):  
 **if** final\_score < 0:  
 final\_score = 0  
 **elif** final\_score > 100:  
 final\_score = 100  
 self.\_\_final = final\_score

The following is a sample test run:

Enter student name: Allen Johnson

Enter project score: 95

Enter midterm score: 82

Enter final score: 93

Improvement target met. Five points added to final exam score.

New final exam score: 98.0

Average score of this student: 91.66666666666667

Enter student name: Ben Morris

Enter project score: 91

Enter midterm score: 78

Enter final score: 96

Improvement target met. Five points added to final exam score.

New final exam score: 100

Average score of this student: 89.66666666666667

The second student scores 96 in final exam. This score is increased to 100 instead of 101 for meeting the improvement goal.

# **12.7 The \_\_str\_\_ Method**

Each class in Python has a set of **special methods**. These methods are called automatically when they are needed. For example, the \_\_init\_\_ method is called when a new instance of a class is created. All special methods are named in the same format. They must start and end with two underscore characters. We are going to see another special method in this section.

Although objects are not strings, we can actually treat them like strings and use a print statement to display information of an object. For example, the statement

print(acct1)

displays the type of an object and its memory address by default:

<account.Account object at 0x0000019C4ED03320>

We can actually define a \_\_str\_\_ method to customize what is displayed. The \_\_str\_\_ method returns a string when the object is treated like a string. In the following example, we are adding a \_\_str\_\_ method to the Account class.

**class** Account:  
  
 **def** \_\_init\_\_(self, name, acct\_num):  
  
 *""" constructor """* self.\_\_name = name  
 self.\_\_acct\_num = acct\_num  
 self.\_\_balance = 0  
  
  
 **def** deposit(self, amt):  
  
 *""" deposit money in the account """* self.\_\_balance = self.\_\_balance + amt  
  
  
 **def** withdraw(self, amt):  
  
 *""" withdraw money from the account """* **if** amt > self.\_\_balance:  
 print(**"Withdrawal denied due to insufficient fund."**)  
 **else**:  
 self.\_\_balance = self.\_\_balance - amt  
  
  
 **def** getBalance(self):  
 **return** self.\_\_balance  
  
  
 **def** \_\_str\_\_(self):  
 **return 'Account name: '** + self.\_\_name + **'\nAccount number '** + self.\_\_acct\_num + **'\nBalance: '** + str(self.\_\_balance)

A multiple-line string that displays name, account number and balance is returned by the \_\_str\_\_ method.

A print statement is added at the end of the main function to display the object:

**from** account **import** Account  
  
**def** main():  
 acct\_name = input(**"Enter name: "**)  
 acct\_num = input(**"Enter account number: "**)  
 acct1 = Account(acct\_name, acct\_num)  
  
 oper = 0  
 **while** oper != 3:  
 oper = int(input(**"Enter 1 for deposit, 2 for withdrawal, 3 for exit: "**))  
 **if** oper == 1:  
 amount = int(input(**"Enter deposit amount: "**))  
 acct1.deposit(amount)  
 print(**"Balance:"**, acct1.getBalance())  
 **elif** oper == 2:  
 amount = int(input(**"Enter withdrawal amount: "**))  
 acct1.withdraw(amount)  
 print(**"Balance:"**, acct1.getBalance())  
  
 print(acct1)  
  
main()

The following is a test run of the program.

Enter name: Allen Johnson

Enter account number: 123-4-5678

Enter 1 for deposit, 2 for withdrawal, 3 for exit: 1

Enter deposit amount: 250

Balance: 250

Enter 1 for deposit, 2 for withdrawal, 3 for exit: 3

Account name: Allen Johnson

Account number 123-4-5678

Balance: 250

# **12.8 Conclusion**

Object oriented programming is introduced in this lesson. In this programming model, related data and functions are bundled together as objects, and outside access to certain members of an object can be limited and controlled through getter and setter methods. This idea of creating and using self-contained parts as building blocks of a program is known as **encapsulation**. There are three fundamental features of object oriented programming. Encapsulation is the first one. In the next lesson, we will learn the other two.