ECE 364 Lab 7 Handout

Object Oriented Programming in Python

October 4, 2017

Passing this lab will satisfy course objectives CO2, CO3, CO6

Instructions

- Work in your Lab07 directory.
- Remember to add and commit all files to SVN. **We will grade the version of the file that is in SVN!**
- Make sure you file compiles. **You will not receive any credit if your file does not compile.**
- Name and spell the file and the functions exactly as instructed. Your scripts will be graded by an automated process. Also, make sure your output from all functions match the given examples. **You will not receive points for a question whose output mismatches the expected result.**
- Make sure you are using Python 3.4 for your project. In PyCharm, go to:

  File Menu → Settings → Project Interpreter

  And make sure that Python 3.4 (`/usr/local/bin/python3.4`) is selected.

Object Oriented Programming in Python

Description

Students in some university are collaborating on several projects, where in each projects they are building multiple circuits using four component types. The format of each component is: `<Symbol><floatValue>`, e.g. `L411.320`, where the symbols used are: `R`, `L`, `C` and `T` for Resistors, Inductors, Capacitors and Transistors, respectively. Since managing the information of every student and every project can get out of hand, you will use OOP to help tackle this issue.

Create a Python file named `oopTasks.py`, and do all of your work in that file. This is the only file you need to submit. This file should only consists of one or more "**classes**", and, optionally, a conditional main block, (i.e. `if __name__ == "__main__":`). Do not include loose statements (i.e. statements that do not belong to any function, class, nor the conditional main block.) You can, however, write within any class, any number of additional member functions and member variables that you might need, but they have to be part of your class implementation.

Level Enum

Create a `Level` enum[1] that represents a student's level in school, with the following options:

```
freshman
sophomore
junior
senior
```

Note that you can optionally associate values with these enum options.

Student Class

Implement the `Student` class, which holds the information about a student participating in school projects. Class members are:

Member Variables:

- `ID` - A string of the form `XXXXX-XXXXX` that represents the student's ID.
- `firstName` - The first name of the student.
- `lastName` - The last name of the student.
- `level` - An instance of the `Level` enum representing the student's level in school.

Member Functions:

- **Initializer** - initializes an instance through providing all of the member variables in the same order shown above, i.e. the ID, first name, last name, and level. Do not default any of the arguments, and validate that the level passed is an instance of the `Level` enum. If it is not, raise an error[2] as follows, or you may use your own message:

  ```
  raise TypeError("The argument must be an instance of the 'Level' Enum.")
  ```

[1]The Enum is a special "class" that has been added to Python since version 3.4. Please refer to Python's Documentations for more information.

[2]We will be covering errors and exceptions later.

- **String Representation** - returns a string representation of this instance in the following format:

  ```
  ID, first last, level
  ```

  An example of the returned string would be:

  ```
  15487-79431, John Smith, Freshman
  ```

  Note that the student's level is capitalized, unlike the enum member.

Circuit Class

Implement the `Circuit` class that contains the information related to a circuit in this problem. Class members are:

Member Variables:

- `ID` - A five-digit string that represents the circuit's ID.
- `resistors` - A list of strings containing the resistors used in this circuit.
- `capacitors` - A list of strings containing the capacitors used in this circuit.
- `inductors` - A list of strings containing the inductors used in this circuit.
- `transistors` - A list of strings containing the transistors used in this circuit.

Member Functions:

- **Initializer** - initializes an instance through providing all of the member variables in the same order shown above, i.e. the ID, resistors, capacitors, inductors and transistors. Do not default any of the arguments, and validate that each of the component lists passed are either empty, or contain valid components of the respective type. If any list contains a component that does not begin with the expected letter for that component, raise an error indicating what list is incorrect. For example:

  ```
  raise ValueError("The resistors' list contain invalid components.")
  ```

  You also might want to include the exact components that were invalid to be more informative to the user.

- **String Representation** - returns a string representation of this instance in the following format:

  ```
  ID: (R = XX, C = XX, L = XX, T = XX)
  ```

  where the `XX` is the number of elements of the respective component type. An example of the returned string would be:

  ```
  99887: (R = 12, C = 03, L = 75, T = 00)
  ```

  Note that each number must be written as two digits. (You may assume that the count of components of any type will not exceed 99.)

- Write a function called `getDetails` that takes in no arguments, and returns a string, different from the string representation, listing all of the components as follows:

  ```
  ID: R List, C List, L List, T List
  ```

An example of the this string would be:

```
99887: R206.298, R436.943, C194.315, C261.054, C668.027, L49.234, T663.350
```

Note that each list must be printed in a sorted order, and all components are separated by a comma and a space. This string must not end with a comma, a space nor a newline character.

Operator Overloads:

- `component in Circuit` - implements a membership check using the `"in"` operator to identify whether a specific component is present in the circuit or not. If it is present in one of the lists, return `True`, otherwise return `False`.

  Notes:

  - Check if the component in question is a string. If it is not, raise a `TypeError` with a descriptive error message.
  - Check if the component is one of the known types, i.e. it starts with one of the four expected letters. If not, raise a `ValueError` with a descriptive error message.

- `Circuit + component` - implements the '`+`' operator that adds a component to the respective list and returns a reference to the current object instance, i.e. `return self`.

  Notes:

  - This operation should be commutative.
  - If the component is already present, just return the reference with no other actions.
  - Check if the component in question is a string. If it is not, raise a `TypeError` with a descriptive error message.
  - Check if the component is one of the known types, i.e. it starts with one of the four expected letters. If not, raise a `ValueError` with a descriptive error message.

- `Circuit - component` - implements the '`-`' operator that removes a component from the respective list and returns a reference to the current object instance, i.e. `return self`.

  Notes:

  - This operation is non-commutative.
  - If the component is *not* present, just return the reference with no other actions.
  - Check if the component in question is a string. If it is not, raise a `TypeError` with a descriptive error message.
  - Check if the component is one of the known types, i.e. it starts with one of the four expected letters. If not, raise a `ValueError` with a descriptive error message.

- `Circuit1 + Circuit2` - implements the '`+`' operator that combines the information from two circuits as follows:

  - Create a random 5-digit string as a new circuit ID. (You may need the `sample` function from the `random` module.)
  - Combine the components from both circuits and create a new list for each type containing the union of elements from both circuits, with no duplicates.
  - Create and return a new `Circuit` instance using the new ID and the new lists.

  Note: Check if `Circuit2` is an instance of the `Circuit` class. If it is not, raise a `TypeError` with a descriptive error message.

Project Class

Implement the `Project` class that contains the information related to a project in this problem. Class members are:

Member Variables:

- `ID` - A UUID string that represents the project's ID.
- `participants` - A list of `Student` instances representing the students who worked on the project.
- `circuits` - A list of `Circuit` instances of the circuits used in this project.

Member Functions:

- **Initializer** - initializes an instance through providing all of the member variables in the same order shown above, i.e. the ID, participants and circuits. Do not default any of the arguments, and validate that each of the lists passed is *not* empty, and contains valid elements of the respective type. If not, raise `ValueError` with a descriptive message indicating what list is incorrect.
- **String Representation** - returns a string representation of this instance in the following format:

  ```
  ID: XX Circuits, XX Participants
  ```

  where the `XX` is the number of elements of the respective variable. An example of the returned string would be:

  ```
  38753067-e3a8-4c9e-bbde-cd13165fa21e: 11 Circuits, 04 Participants
  ```

  Note that each number must be written as two digits.
- Write a function called `getDetails` that takes in no arguments, and returns a multiline string with the following format:

  ```
  <ProjectID>

  Participants:
  <Student1>
  <Student2>
  ...
  <StudentN>

  Circuits:

  <Circuit1 Details>
  <Circuit2 Details>
  ```

  Notes:

  - Both of the lists, the participants and circuits, must be printed in a sorted order, by the IDs.
  - The `<Student>` line is the string representation of the student.
  - The `<Circuit Details>` line is the detail information of the circuit, showing all of the components.

  An example would be:

```
38753067-e3a8-4c9e-bbde-cd13165fa21e

Participants:
15487-79431, John Smith, Freshman

Circuits:
99887: R206.298, R436.943, C194.315, C261.054, C668.027, L49.234, T663.350
```

Operator Overloads:

- `component in Project` - implements a membership check using the `"in"` operator to identify whether a specific component is present in the project or not. If it is present in any circuit, return `True`, otherwise return `False`.

  Notes:

  - Check if the component in question is a string. If it is not, raise a `TypeError` with a descriptive error message.
  - Check if the component is one of the known types, i.e. it starts with one of the four expected letters. If not, raise a `ValueError` with a descriptive error message.

- `Circuit in Project` - implements a membership check using the `"in"` operator to identify whether the given `Circuit` instance is present in the project or not. If it is present, return `True`, otherwise return `False`. (You may assume that a circuit is uniquely identified by its ID.)

  Note: Check if the element in question is an instance of the `Circuit` class. If it is not, raise a `TypeError` with a descriptive error message.

- `Student in Project` - implements a membership check using the `"in"` operator to identify whether the given `Student` instance is present in the project or not. If it is present, return `True`, otherwise return `False`. (You may assume that a student is uniquely identified by his/her ID.)

  Note: Check if the element in question is an instance of the `Student` class. If it is not, raise a `TypeError` with a descriptive error message.

- `Project + Circuit` - implements the `'+'` operator that adds a circuit to the current project and returns a reference to the current object instance, i.e. `return self`.

  Notes:

  - This operation is non-commutative.
  - If the circuit is already present, just return the reference with no other actions.
  - Check if the circuit in question is an instance of the `Circuit` class. If it is not, raise a `TypeError` with a descriptive error message.

- `Project - Circuit` - implements the `'-'` operator that removes a circuit from the current project and returns a reference to the current object instance, i.e. `return self`.

  Notes:

  - This operation is non-commutative.
  - If the circuit is *not* present, just return the reference with no other actions.
  - Check if the circuit in question is an instance of the `Circuit` class. If it is not, raise a `TypeError` with a descriptive error message.

- `Project + Student` - implements the `'+'` operator that adds a participant to the current project and returns a reference to the current object instance, i.e. `return self`.

Notes:

- This operation is non-commutative.
- If the student is already present, just return the reference with no other actions.
- Check if the participant is an instance of the `Student` class. If it is not, raise a `TypeError` with a descriptive error message.

- `Project - Student` - implements the '`-`' operator that removes a participant from the current project and returns a reference to the current object instance, i.e. `return self`.

  Notes:

  - This operation is non-commutative.
  - If the participant is *not* present, just return the reference with no other actions.
  - Check if the participant is an instance of the `Student` class. If it is not, raise a `TypeError` with a descriptive error message.

Capstone Class

The `Capstone` class is an extension of the `Project` that has the following enhancements:

Member Override:

- **Initializer**: In addition to the base requirements, validate that all participating students are seniors. If not, raise `ValueError` with a descriptive message.
- `Project + Student`: In addition to the base requirements, validate that the given student is a senior. If not, raise `ValueError` with a descriptive message.