ECE 364 Prelab Assignment 07

Object Oriented Programming in Python

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

Instructions

- You must meet all *base requirements* in the syllabus to receive any credit.
- Work in your Prelab07 directory.
- Remember to add and commit all **required** files to SVN. **We will grade the version of the file that is in SVN!**
- Do *not* add any file that is *not* required. **You will lose points if your repository contains more files than required!**
- 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. **You will lose some points, per our discretion, for any function that does not match the expected name.**
- 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.**
- Unless otherwise specified, you cannot use any external library, but you can use any module in the **Python Standard Library** to solve this lab, i.e. anything under:

  `https://docs.python.org/3.7/library/index.html`
- Make sure you are using Python 3.7 for your Prelab.

Object Oriented Programming in Python

Required File `oopTasks.py`

Description

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. You can write any number of "**classes**" and helper functions in this file, but **DO NOT CREATE ANY MODULE VARIABLES**. Refer to the **Python File Structure** section below for a reminder on the requirements. Note that the use of type annotation is *not* required, but it is recommended for better compile-time static analysis.

Students in some university are collaborating on several projects, where in each projects they are building multiple circuits using four component types: Resistors, Inductors, Capacitors and Transistors. Since managing the information of every student and every project can get out of hand, creating an Object-Oriented Design can help tackle this issue. Your task is to implement and test the classes described below.

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 any values with enum options.

ComponentType Enum

Create a `ComponentType` enum that represents a single component type with the following options:

```
Resistor
Capacitor
Inductor
Transistor
```

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 representing 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.

[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.

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 a `TypeError` with an appropriate message, e.g.:

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

- **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
  ```

Component Class

Create a `Component` class that represent a single component with the following members:

Member Variables:

- `ID` - A string representing the component's ID.
- `ctype` - An instance of the `ComponentType` enum to identify what component type it is.
- `price` - A float value representing the component price.

Member Functions:

- **Initializer** - initializes an instance through providing all of the member variables in the same order shown above, i.e. the ID, ctype, and price. Do not default any of the arguments, and validate that the ctype passed is an instance of the appropriate enum. If it is not, raise a `TypeError` with an appropriate message.

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

  ```
  <ctype>, <ID>, $<price>
  ```

  An example of the returned string would be:

  ```
  Resistor, REW-321, $1.40
  ```

  Note that the price should be preceded by a "`$`" and formatted to have two decimals.

- **Instance Hash**[2] - returns a hash value of this instance using the `hash()` function, as:

  ```
  hash(<Component ID>)
  ```

[2]The hash value is used by sets and other collections to uniquely identify the object. It is required to be able to add an object to a set.

Circuit Class

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

Member Variables:

- `ID` - A string that represents the circuit's ID.
- `components` - A set of `Component` instances containing the components used in this circuit.
- `cost` - A calculated float value representing the total cost of this circuit, rounded to two decimals.

Member Functions:

- **Initializer** - initializes an instance through providing the ID and the set of components, and calculates the cost. Do not default any of the arguments, and validate that each of the elements in the components' set passed is an instance of the `Component` class (or the set passed is empty). If not, raise a `TypeError` with an appropriate message.
- **String Representation** - returns a string representation of this instance in the following format:

  ```
  ID: (R = XX, C = XX, I = XX, T = XX), Cost = $<cost>
  ```

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

  ```
  11-0-88: (R = 12, C = 03, I = 75, T = 00), Cost = $123.90
  ```

  Note that each number must be written as two digits, and the cost as a float with two decimals. (You may assume that the count of components of any type will not exceed 99.)
- Write a function called `getByType` that takes in an instance of the `ComponentType` enum as an argument, and returns a set of all the elements in the circuit from that component type. Note that the result can be an empty set. Check if the component is an instance of the expected enum, and raise a `ValueError` with a descriptive error message if it is not.

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, return `True`, otherwise return `False`.

  Notes:

  - Check if the component in question is an instance of `Component` class. If it is not, raise a `TypeError` with a descriptive error message.
- `Circuit +/- Component` - implements the `'+'`/`'-'` operator that adds/subtracts a component to/from the circuit instance, updates the cost, and returns a reference to the current object instance, i.e. `return self`.

  Notes:

  - Addition operation is commutative, while subtraction is not.
  - Return the reference with no other actions, if the operation is an addition and the component is present, or if the operation is subtraction, and the component is not present.

  - Check if the component in question is an instance of `Component` class. If it is not, raise a `TypeError` with a descriptive error message.

- `Circuit1 (=,<,>) Circuit2` - implements comparison operators `=,<,>` and returns `True` or `False`, based on the circuits' costs.

  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 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.
- `cost` - A calculated float value representing the total cost of this project, rounded to two decimals.

Member Functions:

- **Initializer** - initializes an instance through providing the member variables: ID, participants and circuits, in order, and calculates the cost. Do not default any of the arguments, and validate that each of the lists passed contains valid elements of the respective type (either/or of the set passed can be empty). If not, raise a `ValueError` with a descriptive message indicating which list is incorrect.
- **String Representation** - returns a string representation of this instance in the following format:

  ```
  ID: (XX Circuits, XX Participants), Cost = $<cost>
  ```

  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), Cost = $213.14
  ```

  Note how the values are formatted in the string.

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`.

  Note: Check if the element in question is an instance of the `Component` class. If it is not, raise a `TypeError` 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/subtracts a circuit to/from the current project, updates the cost, and returns a reference to the current object instance, i.e. `return self`.

  Notes:

  - This operation is non-commutative.
  - Return the reference with no other actions, if the operation is an addition and the circuit is present, or if the operation is subtraction, and the circuit is not present.
  - 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[circuitID]` - implements the item getter, which takes in a circuit ID, and returns an instance of the `Circuit` class from the project, if the circuit is present. If it is not, raise a `KeyError` with an appropriate message.

Capstone Class

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

Member Override:

- **Initializer**: The initializer can instantiate the class, and populate the member variables, using one of two methods:

  - The keyword arguments `ID`, `participants` and `circuits` are provided, with the same types as those used in the `Project` class.
  - The keyword argument `project`, an instance of the base class, is provided.

  In both of these possibilities, validate that all participating students are seniors. If not, raise `ValueError` with a descriptive message.

Python File Structure

The following is the expected file structure that you need to conform to:

```
#######################################################
#   Author:     <Your Full Name>
#   email:      <Your Email>
#   ID:         <Your course ID, e.g. ee364j20>
#   Date:       <Start Date>
#######################################################

import os     # List of module import statements
import sys    # Each one on a line


#######################################################
# No Module-Level Variables or Statements!
# ONLY CLASSES AND FUNCTIONS BEYOND THIS POINT!
#######################################################

class MyClass:
    pass

def functionName1(a: float, b: float) -> float:
    return 0.


def functionName2(c: str, d: str) -> int:
    return 1


# This block is optional and can be used for testing.
# We will NOT look into its content.
#######################################################
if __name__ == "__main__":
    # Write anything here to test your code.
    z = functionName1(1, 2)
    print(z)
```