Assignment 1

Due date: Wednesday, February 19, 2020 before noon sharp (not 12:10). This is one day later than originally planned, due to the Family Day holiday.

You may complete this assignment individually or with a partner, who can be from any section of the course.

Introduction

In every field there are unsolved problems: P vs. NP (you'll learn about this as you progress in CS), the Oberwolfach problem (in math), whether the universal grammar exists (in linguistics). But in the field of teaching university courses, the true unsolvable problem is:

> What is the best way to put students into groups?

Luckily, with your extensive programming knowledge that you have acquired from taking CSC148, you will be able to solve this problem once and for all.

Your task is to complete a program that analyzes data extracted from survey questions and uses this data to make optimal groups of students. This task has been broken down into several steps that are outlined in detail below.

Learning Goals

By the end of this assignment you should be able to:

- read code you didn't write and understand its design and implementation, including:
  - reading the class and method docstrings carefully (including attributes, representation invariants, preconditions, etc.)
  - determining relationships between classes, by applying your knowledge of composition and inheritance
- complete a partial implementation of a class, including:
  - reading the representation invariants to enforce important facts about implementation decisions
  - reading the preconditions to factor in assumptions that they permit
  - writing the required methods according to their docstrings
  - using inheritance: defining a subclass of another class
- perform unit testing on a program with many interacting classes

General Guidelines

- You may complete this assignment individually or with a partner.
- Please read this handout carefully and ask questions if there are any steps you do not understand.

- The tasks are not designed to be equally difficult, or even in increasing order of difficulty. They are just laid out in logical order.
- Although implementing a complex application can be challenging at first, we will guide you through a progression of tasks, in order to gradually build pieces of your implementation. However, it is your responsibility to read through this handout and the starter code we provide, carefully, and to understand how the classes work together in the context of the application.

Coding Guidelines

These guidelines are designed to help you write well-designed code that will adhere to the interfaces we have defined (and thus will be able to pass our test cases).

You must:

- write each method in such a way that the docstrings you have been given in the starter code accurately describe the body of the method.
- write at least one unit test for every method and function in the code you submit (this includes any helper methods that you may write).
  - You do NOT need to write doctests for the functions and methods in this assignment. The setup required to do the testing would create disruptively long docstrings.
- incorporate inheritance into your code, as described in the Your Task section.
- avoid writing duplicate code.

You must NOT:

- change the parameters, parameter type annotations, or return types in any of the methods you have been given in the starter code.
- add or remove any parameters in any of the methods you have been given in the starter code.
- change the type annotations of any public or private attributes you have been given in the starter code.
- create any new public attributes.
- create any new public methods that do not override or extend a method of the same name in a parent class.
- create any new top-level functions, that is, functions defined outside of any class.
- write code that mutates objects unnecessarily.
- add any more import statements to your code.

You may need to:

- create new private methods for the classes you have been given.
  - if you create new private methods you must provide type annotations for every parameter and return value. You must also write a full docstring for this method as described in the **function design recipe (https://q.utoronto.ca/courses/130571/files/5593988/download) ***
- create new private attributes for the classes you have been given.

    - if you create new private attributes you must give them a type annotation and include a description of them in the class's docstring as described in the **class design recipe (https://q.utoronto.ca/courses/130571/files/5594041/download) ***
- change the inheritance structure of classes you have been given in the starter code.
- import more objects from the `typing` module

While writing your code you can assume that all arguments passed to the methods you have been given in the starter code will respect the preconditions outlined in the methods' docstrings.

Your Task

Complete the classes and methods given to you in the starter code to create a piece of software that keeps track of survey data and uses the results of that data to make optimal groups of students according to provided criteria.

You are encouraged to complete this assignment in the order outlined in the steps below but you may choose a different order if you wish.

Make sure you complete each *"Test your code!"* section before moving on to the next step to make sure that your code works as expected.

In each of the steps below you are encouraged to read questions in the *"Something to think about"* sections. You are not required to answer these questions for this assignment but thinking about them might help you write better code!

Step 1: Get the starter code and read the documentation

1. Download the zip file that contains the starter code here **a1.zip (https://q.utoronto.ca/courses/130571/files/5960200/download?wrap=1)**
2. Unzip the file and place the contents in pycharm in your `a1` folder (remember to set your `a1` folder as a sources root)
3. You should see the following files:
    - `course.py`
    - `criterion.py`
    - `grouper.py`
    - `survey.py`
    - `tests.py`
    - `example_tests.py`
    - `example_usage.py`
    - `example_course.json`
    - `example_survey.json`

For this assignment, you will be required to edit and submit the following files only:

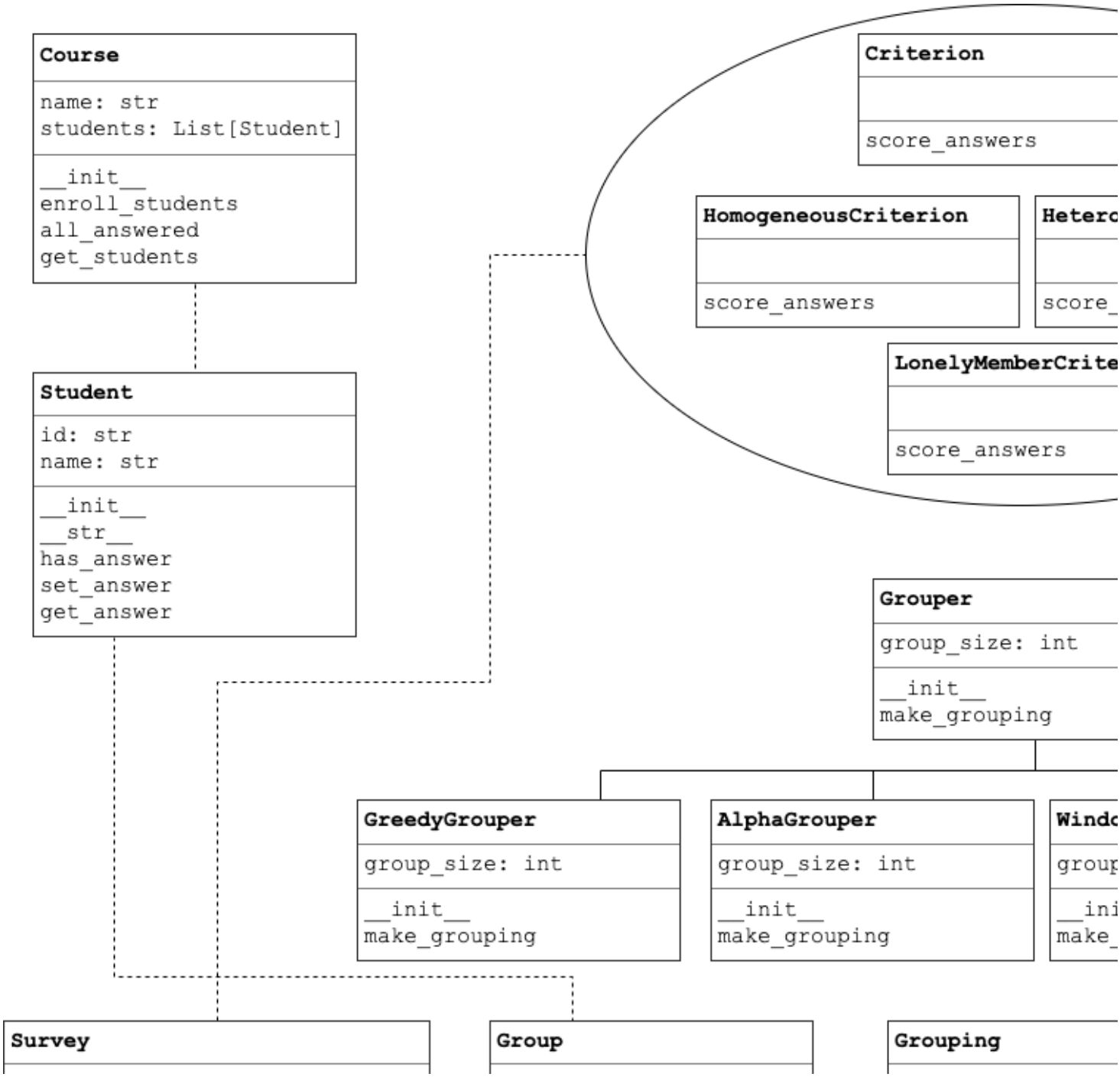
- `course.py`
- `criterion.py`

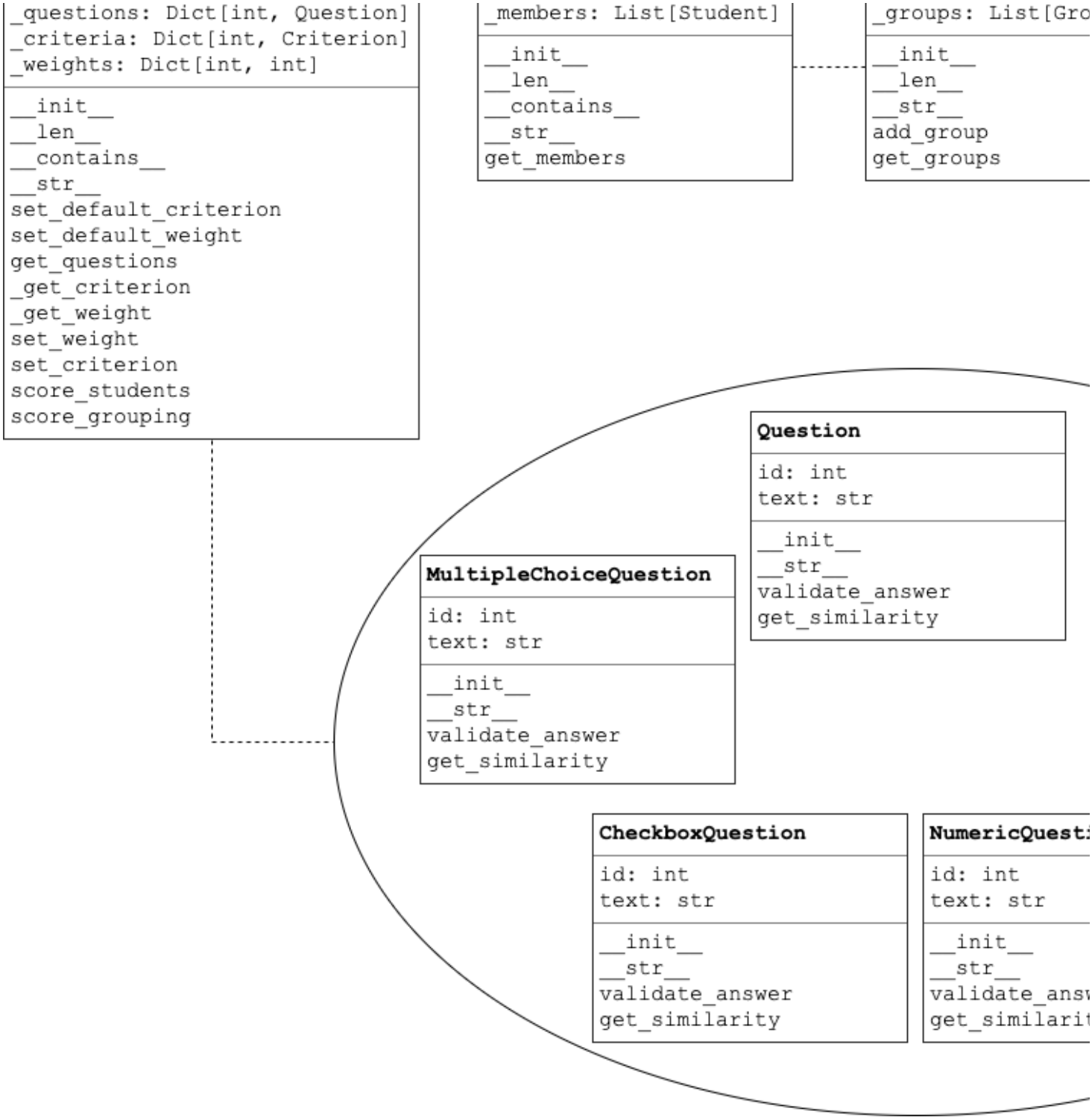
- `grouper.py`
- `survey.py`
- `tests.py`

If you look at these files you will notice that you have been given the signature and docstrings for all classes and methods. Read through these docstrings carefully; they describe how you are expected to implement these classes and methods.

A picture!

It might be difficult to imagine how all the classes defined in these files will interact before you start writing the code itself. To help you out, here is a diagram of all the classes you will be asked to contribute to for this assignment:





Note that the attributes and methods shown in this diagram are only the ones that we have given you in the starter code. You may need to define additional private attributes or private helper methods.

Legend:

- dashed lines indicate a composition relationship between classes
- solid lines indicate an inheritance relationship between classes
- a solid circle around a group of classes indicates that there exists an inheritance relationship between these classes but it is not defined (you get to decide!)

Test your code!

- Try running the `example_tests.py` file: all of the tests should fail because you haven't written any code yet!
- Try running `example_usage.py` file: you should get an error since you haven't written any code yet!
- Open up the `tests.py` file: it is empty! This is where you will be writing all of your tests for this assignment

Something to think about!

Unlike A0, you will be submitting code split across multiple files. Open up each of the files and look at which functions and classes are defined in each file. Why do you think the files were organized in this way? Is there a different way we could have organized these files?

Step 2: Complete the Student class

The `Student` class represents a student who can be enrolled in a university course.

The starter code for the `Student` class can be found in `course.py`. Open up this file and read through the docstrings for each of the the `Student` class's methods. Then, implement each of the methods in the `Student` class.

Remember: you may need to define additional private attributes or private helper methods!

Test your code!

- Write at least one unit test for each method in `Student`. You are not required to write tests for initializers.
- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.
- Run the tests in `example_tests.py`, the tests in the `TestStudent` class should now pass.

Something to think about!

The `Student.has_answer` method asks you to check if a student has a *valid* answer to a given question. Do we have a way to determine if an answer is valid or not yet? Answer: no and we won't until we complete step 4. You may need to come back and finish this method after completing step 5.

Step 3: Complete the Course class

The `Course` class represents a university course.

The starter code for the `Course` class can be found in `course.py`. Open up this file and read through the docstrings for each of the the `Course` class's methods. Then, implement each of the methods in the `Course` class. You may find the function `sort_students` helpful.

Remember: you may need to define additional private attributes or private helper methods!

Test your code!

- Write at least one unit test for each method in `Course`. You are not required to write tests for initializers.
- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.
- Run the tests in `example_tests.py`, the tests in the `TestCourse` class should now pass.

Something to think about!

The `Course.all_answered` method asks you to check if all students have a *valid* answer for every question in a Survey. Which steps do you need to complete before you can finish this method? You may have to come back later to finish the `Course.all_answered` method.

Step 4: Complete the Question classes

The file `survey.py` contains an abstract `Question` class, and the following classes for representing different types of questions that you might find on a survey:

- `Question`
- `MultipleChoiceQuestion`
- `NumericQuestion`
- `YesNoQuestion`
- `CheckboxQuestion`

As well as defining the text of the question itself, these classes also specify what are valid answers to these questions.

Open up `survey.py` and read through the docstrings for the methods in these question classes.

You might notice that we have not defined any inheritance hierarchy between these classes. You get to decide what it should be. However, in doing so you must follow these rules:

1. The abstract `Question` class should not inherit from any class other than `object`.
2. All other Question classes should inherit from the abstract `Question` class either directly or indirectly.
3. At least one non-abstract Question class should inherit from another non-abstract Question class.

There are many possible inheritance structures you could choose. Remember that one of the requirements for this assignment is to avoid writing duplicate code. Think about which sort of inheritance structure best lets you avoid duplicate code.

Implement each of the methods in the Question classes. You may remove a method that we included in the starter code for a child class if you wish to simply inherit the parent's method rather than to override it.

Remember: you may need to define additional private attributes or private helper methods!

Test your code!

- Write at least one unit test for each method in each of the Question classes. You do not need to write tests for abstract methods or initializers but you do need to write tests for inherited methods.

  For example, even if you structure your code so that a child class inherits its `validate_answer` method without modification from the parent class, you still need to write separate tests for the `validate_answer` method in the parent class and the child class.

- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.

- Run the tests in `example_tests.py`, the tests in the `TestMultipleChoiceQuestion`, `TestNumericQuestion`, `TestYesNoQuestion`, and `TestCheckboxQuestion` class should now pass.

Something to think about!

The `validate_answer` methods ask you to check if an answer is a *valid* answer for this question. Do we have a enough information about the `Answer` class in order to complete this method now? You may need to come back and finish this method after completing step 4.

Step 5: Complete the Answer class

The `Answer` class represents an answer to one of the questions you wrote classes for in Step 3.

The starter code for the `Answer` class can be found in `survey.py`. Open up this file and read through the docstrings for each of the the `Answer` class's methods. Then, implement each of the methods in the `Answer` class.

Remember: you may need to define additional private attributes or private helper methods!

If you have not implemented the `validate_answer` methods in the Question classes, the `Course.all_answered` and the `Student.has_answer` methods yet, go back and finish them now.

Test your code!

- Write at least one unit test for each method in `Answer`. You are not required to write tests for initializers.
- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.
- Run the tests in `example_tests.py`, the tests in the `TestAnswer` class should now pass.

Something to think about!

The `Answer` class is one of the simplest classes that we will implement in this assignment. What is the advantage of creating such a simple class? Are there any disadvantages?

Step 6: Complete the Criterion classes

A criterion is a way of judging the quality of a group based on the group members' answers to a particular question. For example, one criterion could be to want groups with homogeneous answers to a question asking what year they are in.

The starter code defines several Criterion classes in `criterion.py`. Open up this file and read through the docstrings for each of the the Criterion class' methods. The Criterion classes are the classes in this file that have "Criterion" in their name:

- `Criterion`
- `HomogeneousCriterion`
- `HeterogeneousCriterion`
- `LonelyMemberCriterion`

You might notice that we have not defined any inheritance hierarchy between these classes. You get to decide what it should be. However, in doing so you must follow these rules:

1. The abstract `Criterion` class should not inherit from any class other than `object`.
2. All other Criterion classes should inherit from the abstract `Criterion` class, either directly or indirectly.
3. At least one non-abstract Criterion class should inherit from another non-abstract Criterion class.

There are many possible inheritance structures you could choose. Remember that one of the requirements for this assignment is to avoid writing duplicate code. Think about which sort of inheritance structure best lets you avoid duplicate code.

Implement each of the methods in the Criterion classes. You should NOT implement an initializer for these classes.

Remember: You may remove a method defined in a child class if you wish to simply inherit the parent's method directly.

Remember: you may need to define additional private helper methods!

Test your code!

- Write at least one unit test for each method in each of the Criterion classes. You do not need to write tests for abstract methods but you do need to write tests for inherited methods.

  For example, even if you structure your code so that a child class inherits its `score_answers` method without modification from the parent class, you still need to write separate tests for the `score_answers` method in the parent class and the child class.

- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.

- Run the tests in `example_tests.py`, the tests in the `TestHomogeneousCriterion`, `TestHeterogeneousCriterion`, and `TestLonelyMemberCriterion` classes should now pass.

Something to think about!

You are asked not to implement an initializer for the Criterion classes. Why is an initializer not necessary for these classes? If the Criterion classes do not define an initializer, will it be impossible to create instances of these classes?

Step 7: Complete the Group class

The `Group` class represents a collection of one or more students.

The starter code for the `Group` class can be found in `grouper.py`. Open up this file and read through the docstrings for each of the the `Group` class's methods. Then implement each of the methods in the `Group` class.

Remember: you may need to define additional private attributes or private helper methods!

Test your code!

- Write at least one unit test for each method in `Group`. You are not required to write tests for initializers.
- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.
- Run the tests in `example_tests.py`, the tests in the `TestGroup` class should now pass.

Something to think about!

You may find the Python `set` type to be useful.

The `Group.get_members` method asks you to return a shallow copy of the `_members` private attribute instead of simply returning the list that `_members` refers to. A `shallow` copy of an object is a new object (with a different `id`) but whose contents are the same. For example,

```
>>> dicts = [{1:2, 3:9}, {5:18}, {"adieu":7}]
>>> copy = []
>>> for item in dicts:
...     copy.append(item)
...
>>> # dicts and copy are two different objects.
>>> id(dicts)
4485971264
>>> id(copy)
4485971200
>>> # But each item in copy is an alias for an item in dicts.
>>> For example:
>>> id(dicts[2])
4486046896
>>> id(copy[2])
4486046896
>>> # With a Python list, any time we slice we get a new list.
>>> # This provides an easy way to make a shallow copy.
>>> another_copy = dicts[:]
>>> id(another_copy)
4485971008
```

```
>>> id(another_copy[2])
4486046896
```

In contrast, a **deep copy** has new objects at every level, so it contains no aliases.

By returning a shallow copy, the `Group.get_members` method allows the client code to mutate the individual `Student` objects but not the `Group` object itself. The same reasoning holds for the `Grouping.get_groups` method in the next step.

Step 8: Complete the Grouping class

The `Grouping` class represents a collection of `Group` instances. An instance of a `Grouping` class can be used to represent every student in a course, divided up into groups.

The starter code for the `Grouping` class can be found in `grouper.py`. Open up this file and read through the docstrings for each of the the `Grouping` class's methods. Then, implement each of the methods in the `Grouping` class.

Remember: you may need to define additional private attributes or private helper methods!

Test your code!

- Write at least one unit test for each method in `Grouping`. You are not required to write tests for initializers.
- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.
- Run the tests in `example_tests.py`, the tests in the `TestGrouping` class should now pass.

Something to think about!

An instance of the `Grouping` class starts out containing zero groups and more can be added later using the `Grouping.add_group`. On the other hand, an instance of the `Group` class starts out with some members and more cannot be added later. Why might we have chosen to implement these classes differently? What does this design choice tell us about how these classes are intended to be used?

Step 9: Complete the Survey class

The `Survey` class represents a collection of questions. It also associates each question with a criterion (indicating how to judge the quality of a group based on their answers to the question) and a weight (indicating the importance of this question in deciding how to group students).

The starter code for the `Survey` class can be found in `survey.py`. Open up this file and read through the docstrings for each of the the `GroupSurveying` class's methods. Then, implement each of the methods in the `Survey` class.

Hint: Read the documentation for `Survey._get_criterion` and `Survey._get_weight` carefully before you implement the initializer.

Note: The weights associated with the questions in in a survey do NOT have to sum up to any particular amount.

Remember: you may need to define additional private attributes or private helper methods!

Test your code!

- Write at least one unit test for each method in `Survey`. You are not required to write tests for initializers.
- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.
- Run the tests in `example_tests.py`, the tests in the `TestSurvey` class should now pass.

Something to think about!

What is the relationship between the `Survey.score_students` method and the `Survey.score_grouping` method?

Step 10: Complete the helper functions in `grouper.py`

The file `grouper.py` contains helper functions that implement two different ways of dividing a list up into parts.

Open this file and read through the documentation for these functions. Then, implement the functions.

Test your code!

- Write at least one unit test for each function you implemented in this step.
- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.
- Run the tests in `example_tests.py`. The `test_slice_list` and `test_windows` tests should now pass.

Something to think about!

Why are we writing these functions outside of a class? Why not make them instance methods of one of the Grouper classes instead?

Step 11: Complete the Grouper classes

The Grouper classes represent different techniques for deciding how to split all the students in a course into a groups. See the docstrings of each of these classes and their methods for implementation details. The file **Grouper Explanation.pdf (https://q.utoronto.ca/courses/130571/files/5960206/download?wrap=1)**  **(https://q.utoronto.ca/courses/130571/files/5960206/download?wrap=1)** walks through a detailed example for each of the groupers. Make sure you understand the algorithms before you start writing code.

The starter code for the Grouper classes can be found in `grouper.py`. Open up this file and read through the docstrings for each of the the Grouper class' methods.

The Grouper classes are the classes in this file that have "Grouper" in their name:

- `Grouper`
- `AlphaGrouper`
- `RandomGrouper`
- `GreedyGrouper`
- `WindowGrouper`

Unlike the Criterion classes and Question classes, the inheritance structure between the Grouper classes has been given to you. You should NOT change the inheritance structure between the Grouper classes.

Implement each of the methods in the Grouper classes. You may find the function `sort_students` (defined in the `course` module) helpful.

Remember: you may need to define additional private attributes or private helper methods!

Test your code!

- Write at least one unit test for each method in each of the Grouper classes. You do not need to write tests for abstract methods but you do need to write tests for inherited methods.

  For example, even if you structure your code so that a child class inherits a private helper method without modification from the parent class, you still need to write separate tests for the private helper method in the parent class and the child class.

- You should write these tests in the `tests.py` file.
- Once you have finished writing these tests, run all the tests in `test.py`. Make sure your code passes all your tests before moving on.

- Run the tests in `example_tests.py`. The tests in the `TestAlphaGrouper`, `TestRandomGrouper`, `TestGreedyGrouper`, and `TestWindowGrouper` classes should now pass.

A note on writing tests for methods that include randomness

When you test methods that include some randomness, you cannot simply test the return value since it will change from one call to the next. This is where property testing comes in (see the first lecture!).

For example, the tests in `TestRandomGrouper` test the following properties of the `Grouping` instance returned by the `RandomGrouper.make_grouping` method:

- the number of groups in the grouping
- the number of members in each group
- the uniqueness of all members in the grouping

Think about what other properties you could test for when you write your own tests for this class.

Something to think about!

Now that you have written all the code, go back and look at the diagram in Step 1. This is just one possible way to have designed this code. Think about:

- How could we have structured the classes differently?
- What other classes might we want to add to this code in the future?
- Imagine we wanted to add a class that keeps track of all of the courses in a university? How easy would it be to add this new class? Which existing classes (if any) would have to change?

Step 12: Test your code again!

Now that you have finished writing all the code, go back and run the `example_usage.py` file again. This file should now run without errors.

The `example_usage.py` file will create a course (from the data in `example_course.json`) and a survey (from the data in `example_course.json`) and will use that survey to group the students into groups of 2 using the `AlphaGrouper` class. Then it prints the members of each group and an overall score for the grouping.

You may change the group size and the grouper type by modifying the two lines at the bottom of the `example_usage.py` file under this comment:

```
# change the two variables below to test your code with different group
# sizes and grouper types.
```

Why you should write your own tests

The `example_tests.py` file might catch some bugs in your code but it will certainly not catch them all. In order to make sure your code is bug free you *must* write your own unit tests for each method.

Writing tests is also a great way to think about the code in a different way. If you find that you are getting stuck, try writing a test for the method you are stuck on; it just might help!

Also you will be graded on the tests you write so if nothing else you should do it for the marks.

Step 13: Submit your work

You've almost finished and now it is time to hand in your work!

BUT BEFORE YOU SUBMIT:

Take some time to polish up your code This not only will improve your mark, but it also feels good to make your code look its best! Here are some things you can do:

- Pay attention to any violations of the Python style guidelines that PyCharm points out. Fix them!
- In each module, run the provided `python_ta.check_all(...)` code to check for errors. Fix them!
- Make sure the `tests.py` file contains at least one unit test for every method and function that you have implemented in `course.py`, `criterion.py`, `grouper.py`, and `survey.py`.

- Check any docstrings that you have written to make sure they are precise and complete and that they follow the conventions of the **Function Design Recipe (https://q.utoronto.ca/courses/130571/files/5593988/download) *** and the **Class Design Recipe (https://q.utoronto.ca/courses/130571/files/5594041/download) ***.
- Remove any code you added just for debugging, such as print function calls.
- Remove any pass statements where you have added the necessary code.
- Remove any TODO comments where you have added the necessary code.
- Take pride in your gorgeous code! This assignment is a significant piece of software, and you should be proud of the work you've done!

Submission Guidelines:

1. Login to MarkUs and create a group for the assignment, or specify that you're working alone.
2. DOUBLE CHECK ONE LAST TIME THAT YOUR CODE RUNS!!
3. Submit the following files only:
   - `course.py`
   - `criterion.py`
   - `grouper.py`
   - `survey.py`
   - `tests.py`
4. On a Teaching Lab machine, download the file you submitted into a brand-new folder, together with rest of the files you were provided:
   - `example_tests.py`
   - `example_usage.py`
   - `example_course.json`
   - `example_survey.json`
5. Test your code thoroughly one last time! Does it pass all of the tests in `example_tests.py`? Does it pass all of your tests in `tests.py`? Are there any python-ta errors that you should fix? Your code will be tested on the Teaching Lab machines, so it must run in that environment.
6. If you make any changes to your code make sure to submit the updated files to MarkUs before the deadline!

Congratulations, you are finished with your first major assignment in CSC148! You should definitely go and celebrate!

Appendix 1: An explanation of some pieces of the starter code.

There are parts of the starter code that you have been given that have not been explained in class. It is not absolutely necessary to understand them to complete this assignment but a brief explanation of each will be provided below for those of you who are interested:

- `if TYPE_CHECKING:`:

  Pycharm does a lot of magic in order to highlight problematic code and give us hints as to how to improve the code we are writing as we are writing it. In order to do this, Pycharm constantly analyzes

your code by running it though a program called a type checker. The `TYPE_CHECKING` variable is `True` only when a type checker is analyzing our code and `False` when we run our code normally. This lets us define a block of code that we don't want to run normally but is useful for the type checker.

Extra Reading:

```
- https://docs.python.org/3/library/typing.html#typing.TYPE_CHECKING
- https://stackabuse.com/python-circular-imports/
```

- `@pytest.fixture`

  This is a part of the `pytest` library that lets us mark that a function should be treated as a "fixture". In testing, a fixture is an object that can be used in multiple tests. Before each test runs, a fixture object get created if needed by running the fixture function and getting its return value. That fixture object is then passed as a parameter to the test. A test can indicate which fixtures it needs by including a parameter that has the same name as the fixture function.

  For example, the `TestCourse.test_enroll_students` test has a parameter named `empty_course` so before the test is run, the `empty_course` fixture function will run and pass its return value as an argument to the test.

  You do NOT need to use fixtures when writing your tests in the `tests.py` file.

  Extra Reading:

  ```
  - https://docs.pytest.org/en/latest/fixture.html
  ```

- `Tuple[Student, ...]`

  This is a type annotation meaning a tuple that contains any number of Students. The ellipses (three dots) means that the length of the tuple does not matter.

  Extra Reading:

  ```
  - https://docs.python.org/3/library/typing.html#typing.Tuple
  ```

- Classes in pytest:

  You might have noticed that the tests in `example_tests.py` are organized into classes. This is simply a way to keep your tests organized and make it clear which groups of tests logically go together. You are not required to write your tests in `tests.py` in this way but you may if you wish.

  Extra Reading:

  ```
  - https://docs.pytest.org/en/latest/getting-started.html#group-multiple-tests-in-a-class
  ```

  [footnote] You do NOT need to write doctests for the functions and methods in this assignment. The setup required to do the testing would create disruptively long docstrings.