CSC148 Lab#2, winter 2017

learning goals

In this lab you will:

- Practice **designing** and **implementing subclasses**. Resources here include lecture materials from week 2, course notes, chapter 1, and How to think like a computer scientist.
- You'll also re-visit some things you've already done:
  - Get more practice with the class design recipe.
  - Continue using the design recipe for functions (which also works fine for methods).
  - Continue using good programming style by consulting CSC108 style guidelines and pep 8.

You are encouraged to start working on this lab as soon as it is posted. If you feel shaky on the lab, be sure to come in and work through it with your TA. There will be a quiz during the last 20 minutes or so of the lab which you are likely to ace if you have worked through the lab.

where we're headed

You will design and implement **GradeEntry**, **LetterGradeEntry**, and **NumericGradeEntry** below, so that you can create and run a file **test_lab02.py** with code very similar to:

```
if __name__ == '__main__':
    grades = [NumericGradeEntry('csc148', 87, 1.0),
              NumericGradeEntry('bio150', 76, 2.0),
              LetterGradeEntry('his450', 'B+', 1.0)]
    for g in grades:
        # Use appropriate ??? methods or attributes of g in format
        print("Weight: {}, grade: {}, points: {}".format(g.?, g.??, g.???))
    # Use methods or attributes of g to compute weight times points
    total = sum(                    # sum of the list of...
                [g.? * g.??         # ? and ?? are methods or attributes of g
                 for g in grades])  # using each g in grades
    # sum up the credits
    total_weight = sum([g.? for g in grades])
    print("GPA = {}".format(total / total_weight))
```

Important: Notice that the code above never checks whether a particular **g** is a **LetterGradeEntry** versus a **NumericGradeEntry**. You design the classes below so that it just does the right thing! Also notice the list comprehension:

```
[g.? * g.?? for g in grades]
```

Check out list comprehensions if needed.

setup

We assume that you either remember some of the setup techniques from last week, look for them in the lab#1 handout or consult your TA and other students. You?ll need to:

- Log into your cdf account, start up Pycharm or another IDE for Python programs, and navigate to **csc148/Labs/lab2**
- Open a web browser and navigate to the page with **lab2** materials:

  http://www.cdf.toronto.edu/~csc148h/winter/Labs/lab2/

  Here you'll find materials for **lab#2**.
- Download the file specs.txt from among the **lab#2** materials, and save it under your own **lab2** subdirectory. Open **specs.txt** in Pycharm (or some other IDE), and read through it.

Check with your TA before moving on, in order to reassure yourself that you're on the right track.

design GradeEntry

Begin by designing the public interface of class **GradeEntry**, using the instructions below. You are not intended to be able to create instances of class **GradeEntry**, rather through inheritance you will be creating **subclasses** of **GradeEntry**. Here's what you need to do:

1. Create and open a new file with your editor called **grade.py** in the subdirectory **lab2**.
2. Perform an object-oriented analysis of the specifications in specs.txt, following the same recipe we used in class for Point, Shape, Rational, etc.:
   (a) choose class name GradeEntry and write a brief description in the class docstring.
   (b) write some examples of client code that uses your class
   (c) decide what services your class should provide as public methods, for each method declare an API[1] (examples, header, type contract, description)
   (d) decide which attributes your class should provide without calling a method, list them in the class docstring

Show your work to a TA before proceeding, in order to be reassured you are on the right track.

implement GradeEntry

The design is where the hard thinking should take place, so now it's time to implement **GradeEntry**. Remember that there is at least one method in **GradeEntry** that should only be implemented in its subclasses. In that case (or cases) the implementation for the method is an easy one-liner:

```
raise NotImplementedError('Subclass needed')
```

Here's what you need to do:

1. write the body of special methods `__init__`, `__eq__`, and `__str__`
2. write the body of other methods

Notice that there is no practical way to try out instances of **GradeEntry** until you have created one or more **subclasses**. There will be one or more methods that generate those annoying **NotImplementedError**s.

[1] use the CSC108 function design recipe

design NumericGradeEntry

The procedure for designing a subclass is similar to design of class **GradeEntry**, with a few important differences:

1. Rather than **class NumericGradeEntry:**, your declaration will be **class NumericGradeEntry(GradeEntry):**. This tells Python (and human readers) that this class inherits attributes and methods from **GradeEntry**

2. Attributes and methods that you will use unchanged from **GradeEntry** (we say you **inherit** these) need not be mentioned in the class docstring. New attributes of **NumericGradeEntry** should be documented in the class docstring, as usual.

3. If you have new attributes for **NumericGradeEntry** that were not in **GradeEntry**, you will need to redesign the **_init_** method. This will include writing a new docstring for the method that says that **NumericGradeEntry**'s initializer **extends** the initializer of **GradeEntry**, gives a (probably) new header and type contract, a new example of a call to the initializer, and says what the initializer does to the new attributes.

4. You will have (at least) one method that could not be implemented in class **GradeEntry**, but can now be implemented in **NumericGradeEntry**. Your docstring should say that the method in **NumericGradeEntry** (which should have the same name as in **GradeEntry**) **overrides** the corresponding method in **GradeEntry**. You should be careful that the type contract for the method in **NumericGradeEntryEntry** is consistent with the type contract for the corresponding function in **GradeEntry**. A good mental exercise is to convince yourself that any client code that uses an instance of **GradeEntry** without knowing that it is a **LetterGradeEntry** should experience results that are consistent with the public interface of **GradeEntry**.[2]

Show your work to a TA before moving on.

design LetterGradeEntry

This will be very similar to the procedure for designing **NumericGradeEntry**. Once again, devote special attention to the method(s) that were not implemented in **GradeEntry**, as well as the attributes that are new in **LetterGradeEntry**.

implement both NumericGradeEntry and LetterGradeEntry

Here is where we experience the labour-saving features of declaring subclasses. If we designed **GradeEntry** carefully, there should be a lot of code that was written in **GradeEntry**that does not have to be re-written in **NumericGradeEntry** and **LetterGradeEntry** — the code is already there, through inheritance. Avoiding duplicate code also avoids **many** errors. For the attributes and methods of **GradeEntry** that will be used unchanged in these subclasses your implementation consists of... nothing. You did the work in **GradeEntry**. If you don't initialize any new attributes in a subclasses, you don't need to have additional documentation.

We have already discussed how to document an **_init_** method that **extends** the one from **GradeEntry** under design. Here is how you would implement the extended **_init_**:

1. The first statement in your implementation should be

```
GradeEntry.__init__(self, ???)
```

[2] This subtle, yet very important, idea is a consequence of the Liskov Substitution Principle, and is worth taking the time to think through.

. . . where the question marks indicate possible values **GradeEntry**'s initializer needs.

2. after the first line, add code to initialize your new attributes that aren't inherited from **GradeEntry**.

You should have some method(s) that are not inherited from **GradeEntry**. You should implement these as usual, even if they share a name with the corresponding method in **GradeEntry**.

additional exercises

As well as the various grade entries, here are some additional exercises in designing and implementing classes with inheritance. We set up each one so that one appropriate solution involves class **Roster** together with one of its subclasses.

We certainly don't expect you to do this many exercises in the lab, but they are here for additional practice. We'll have a brief quiz in the last 20 minutes or so of the lab, which will involve an exercise similar to one of the additional exercises.

auto-graded exercise on composition of classes

Although this lab has been about **inheritance**, a more common way of using the features of one class in another is **composition**. The second portion of week 2's lab mark is exercise 1 on composition, due Sunday January 22 at 10 p.m.