CS 112, Lab 11 – Exercise - Classes

Due: after break, Sunday November 29th, 11:59pm
You must be present in your scheduled lab time to get credit.

Files:

- Create your own file with our convention (*userID_2xx_L11.py)*.
- You should also download this file for testing: http://cs.gmu.edu/~marks/112/labs/testerL11E.py
- Run the tester as always: `python3 testerL11E.py yourfile.py just these funcs`

As this is an **Exercise**, you can read any and all materials, ask us questions, talk with other students, and learn however you best learn in order to solve the task. Just create your own solution from those experiences, and turn in your work.

Classes allow us to define entirely new types. We might think of each type in Python as a set of values, and we use a class declaration to describe how to build and interact with all the values of this brand new type. We will often need at minimum a constructor (telling us what instance variables to always create), __str__ and __repr__ methods (telling Python how to represent the thing as a string), and then any extra methods that we deem useful ways of interacting with the new values.

We can actually make our own Exception types by making a class that extends any existing Exception type. We'll make one in this lab and raise it a few times. If we made a larger project, we'd probably have some main menu that interacted with the user and would have to figure out how to recover from these raised exception values at some point. This lab has enough going on already, so we aren't doing this today.

Turning It In

Add a comment at the top of the file that indicates your name, userID, G#, lab section, a description of your collaboration partners, as well as any other details you feel like sharing. Please also mention what was most helpful for you. Once you are done, run the testing script once more to make sure you didn't break things while adding these comments. If all is well, go ahead and turn in just your one .py file you've been working on over on BlackBoard to the correct lab assignment. We have our own copy of the testing file that we'll use, so please don't turn that in (or any other extra files), as it will just slow us down.

What can I use?

There are no restrictions on what to use on this lab – use this time to learn how to create classes and create objects of those classes.

Task 1 - Grade

A `Grade` represents a specific score on a specific kind of assessment (and has its own assumedly unique `name`). This is just a "container" type, used to give memorable names to individual grouped sub-values.

`class Grade:` Define the `Grade` class.

- `def __init__(self, kind, name, percent):` `Grade` constructor. All three parameters must be stored to instance variables of the same names (`kind`, `name`, and `percent`). If kind is not one of `"test"`, `"lab"`, `"project"`, or `"final"` (example: `kind=="survey"`), then raise a `GradingError` with the message `"no Grade kind 'survey'"`. (You can skip this exception-raising part until later).
  - `kind` :: `str`. Should be something like `"lab"`, `"test"`, or `"project"`.
  - `name` :: `str`. Could be something like `"L1"`, `"T2"`, or `"P6"`.
  - `percent` :: `int`. (I chose `int` to simplify the assignment, but `float` would have been useful).
- `def __str__(self):` returns a human-centric string representation. If `kind=="test"`, `name=="T1"`, and `percent==97`, then the returned string must be `"test:T1(97%)"` **(no padded zeroes).**
- `def __repr__(self):` returns a computer-centric string representation. As is often the case, we want to construct a string that could be pasted back into the interactive mode to re-create this object. Given the same example as in `__str__`, we need to return the string `"Grade('test', 'T1', 97)"`. Note: we choose to *always* use single-quotes.
- `def __eq__(self, other):` we want to check that two grades are equal (our self and this other grade). We compare each instance variable like so (this is the definition!)

```
return self.kind==other.kind and self.name==other.name and self.percent==other.percent
```

Task 2 - GradeBook

This represents an entire grouping of `Grade` values as a list (named `grades`). We can then dig through this list for interesting things and calculations by calling the methods we're going to implement.

`class GradeBook:` Define the `GradeBook` class.

- `def __init__(self):` `GradeBook` constructor. Create the only instance variable, a list named `grades`, and initialize it to an empty list. This means that we can only create an empty `GradeBook` and then add items to it later on.
- `def __str__(self):` returns a human-centric string representation. We'll choose a multi-line representation (slightly unusual) that contains `"GradeBook:"` on the first line, and then each successive line is a tab, the `str()` representation of the next `Grade` in `self.grades`, and then a newline each time. This means that the last character of the string is guaranteed to be a newline (regardless of if we have zero or many `Grade` values).
- `def __repr__(self):` We will be lazy and tell this to just represent the exact same thing as `__str__`. But don't cut-paste the code! Use this as the entire body of this function:

```
return str(self)
```

- `def add_grade(self, grade):` append the argument `Grade` to the end of `self.grades`
- `def average_by_kind(self, kind):` Look through all stored `Grade` objects in `self.grades`. All those that are the same `kind` as the `kind` parameter should be averaged together (sum their `percent`s and divide by the number of things of that `kind`). If none of that kind exist, return `None`.
- `def get_all_of(self, kind):` create and return a list of references to each `Grade` object in this `GradeBook` that is of that kind.
- `def get_by_name(self, name):` search through `self.grades` in order and return the first `Grade` by the given name. If no such `Grade` value can be found (say, `name=="whatever"`), then `raise` a `GradingError` with the message `"no Grade found named 'whatever'"`. (You can skip this exception-raising part and come back to it).

Task 3 – GradingError

The `GradingError` class extends the notion of an `Exception` (note the `(Exception)` part of the class signature line). We're making our own exception that stores a single string message.

`class GradingError(Exception):` Define the `GradingError` class to be a child of `Exception` class (by putting `Exception` in the parentheses as shown).

- `def __init__(self,msg):` Constructor. Store `msg` to an instance variable named `msg`.
- `def __str__(self):` human-centric representation of `GradingError` is just `self.msg`
- `def __repr__(self):` computer-centric representation (that could be pasted back to interactive mode to recreate the object). We'll always use triple single quotes *as part of the represented string* when including the message. For instance, if `self.msg=="bad grade"`, this method would need to return `"GradingError('''bad grade''')"`

If you skipped the two exception-raising parts above, you should now go back and add those checks/raises. The test cases for `GradingError` will go back and check those other parts correctly did so.

Example Interactions

An example of all needed usage should be present.

```
>>> g1 = Grade("test","T1",80)
>>> g2 = Grade("lab","L12",100)
>>> g3 = Grade("lab","L10",90)
>>> str(g2)
'lab:L12(100%)'
>>> repr(g2)
"Grade('lab', 'L12', 100)"
>>>


>>> gb = GradeBook()
>>> str(gb)
'GradeBook:\n'
>>> gb.add_grade(g1)
>>> gb.add_grade(g2)
>>> gb.add_grade(g3)
>>> str(gb)
'GradeBook:\n\ttest:T1(80%)\n\tlab:L12(100%)\n\tla
b:L10(90%)\n'
>>> print(str(gb))
GradeBook:
      test:T1(80%)
      lab:L12(100%)
      lab:L10(90%)

>>> print(repr(gb))
GradeBook:
      test:T1(80%)
      lab:L12(100%)
      lab:L10(90%)

>>> gb.add_grade( Grade("lab","L3",95) )
>>> print(str(gb))
GradeBook:
      test:T1(80%)
      lab:L12(100%)
      lab:L10(90%)
      lab:L3(95%)

>>> gb.average_by_kind("lab")
95.0
>>> gb.get_all_of("test")
[Grade('test', 'T1', 80)]
>>> gb.get_all_of("lab")
[Grade('lab', 'L12', 100), Grade('lab', 'L10',
90), Grade('lab', 'L3', 95)]
>>> gb.get_by_name("L12")
Grade('lab', 'L12', 100)
>>>

>>> e = GradingError("any message")
>>> str(e)
'any message'
>>> repr(e)
"GradingError('''any message''')"
>>> g4 = Grade("blahblah","L30",99)
Traceback (most recent call last):
   …<snipped>…
__main__.GradingError: no Grade kind 'blahblah'.
>>> gb.add_grade( Grade("blahblah","L4",97) )
Traceback (most recent call last):
   …<snipped>…
__main__.GradingError: no Grade kind 'blahblah'.
>>> gb.get_by_name("L30")
Traceback (most recent call last):
   …<snipped>…
__main__.GradingError: no Grade found named 'L30'
```