PIC 16: Homework 4 (due 5/4 at 10pm)

How should your answers be submitted?

- Download `hw4.py`. Do NOT change the filename.
- Replace `mjandr` by your name.
- Edit the bits that say `UNCOMMENT THIS`, `DEFINE THIS`, `FIX THIS`.
  Do NOT change the class or function names.
- Make sure any test cases are enclosed within `if __name__ == '__main__':`
  Otherwise, they will hinder the grading process and might result in `0` points.
- Submit `hw4.py` to CCLE.

In this assignment you will:

- learn how to create a wrapper for a dictionary using appropriate magic methods;
- define `__contains__` and `__iter__`;
- define your own `exceptions`;
- develop your understanding of inheritance.

1. What if we wanted to use Python to make something more like a `struct` in C++? We've learned that classes in Python behave quite a bit like dictionaries. In fact, we may as well replace all instance variables by a single dictionary. However, we've learned that:

   - we can add keys to dictionaries;
   - we can delete keys from dictionaries;
   - Python is dynamically typed and this means that the values in a dictionary could change type at any moment.

   All of this is in stark contrast to a `struct` in C++.

   In this assignment, we're trying to make a dictionary such that:

   - its keys are fixed;
   - the values corresponding to a specific key always have the same type.

2. Open up `hw4.py`.

   (a) The class `StructuredDict` currently serves as a wrapper for a dictionary. Look at the magic methods `__str__`, `__repr__`, `__len__`, `__getitem__`, `__delitem__`, `__setitem__`. They all just do the relevant thing with `self.__d`, the "private" instance variable which stores the underlying dictionary.

   You can run the code

   ```
   d = StructuredDict({1:1, 2:2.0, 3:'3'})
   print(d[1], d)
   d[4] = 'four'
   print(d)
   del d[4]
   print(d)
   ```

   and from the second line onwards, `d` may as well be a true Python dictionary.

   (b) We see `StructuredDictError`, `DeleteError`, `UpdateValueError`, `InitializationError`.

   These classes all define exceptions. I have structured them so that `StructuredDictError` is the most imprecise error. `DeleteError`, `UpdateValueError`, `InitializationError` are all examples of a `StructuredDictError`. This is a good way to arrange exceptions.

   You can ignore the details of these exceptions for now since you'll edit them later.

   (c) The `Rectangle` and `Student` class start showing how I wish to use the `StructuredDict` class. Both inherit from `StructuredDict` and have a class variable called `key_to_type`. This class variable is supposed to tell us what the keys in our dictionary should be, and what types the corresponding values should have. However, the class variable is currently unused. These classes are attempting to recreate the following C++ structs.

   ```
   struct Rectangle {
       double len1;
       double len2;

       Rectangle(double _len1, double _len2) : len1(_len1), len2(_len2) {}
       double area() { return len1 * len2; }
   };

   struct Student {
       string firstName;
       string lastName;
       double GPA;

       Student(string first, string last, double gpa)
                           : firstName(first), lastName(last), GPA(gpa) {}
   };

   ostream& operator<<(ostream& out, const Student& s) {
       out << "Name: " << s.firstName << " " << s.lastName << ", ";
       out << "GPA: " << s.GPA;

       return out;
   }
   ```

You can see that their initializers work by creating a dictionary and then giving this to the initializer of `StructuredDict`. The initializer of `StructuredDict` currently has a line commented out. Later on, we'll uncomment this line, and define `__check`, so that various type checking is performed at initialization.

(d) Under `if __name__ == '__main__':` is some test code.

When you run it now, all `try` statements run without error. However, you can see:

i. `f1` deletes a key;

ii. `f2` tries to update a `float` to an `int`;

iii. `f3` tries to initialize with an `int` and a `str`, even though both should be `floats`;

iv. `f4` doesn't specify values for all keys, uses additional keys that should not be used, and gets types incorrect.

When I run my solution code it prints.

```
In [1]: runfile('/Users/mjandr/Documents/TeX/Pic16A/classes/hw4_sol.py', wdir='/Users/mjandr/Documents/TeX/Pic16A/classes')
area = 8.0

DeleteError: You cannot delete from a StructuredDict

UpdateValueError: The type of 2 is <class 'int'>, but the value corresponding to the key 'len1' should have type <class 'float'>

InitializationError: the type of d['len1'] is <class 'int'>, but it should be <class 'float'>;
the type of d['len2'] is <class 'str'>, but it should be <class 'float'>;

InitializationError: the following keys are missing from d: {0, 1};
the following keys were supplied in error: {5, 6};
the type of d[3] is <class 'int'>, but it should be <class 'float'>;
the type of d[4] is <class 'int'>, but it should be <class 'str'>;
```

That's our goal!

3. The easiest part! (Although you'll have to wait until after 4/24/2020 to understand `__iter__`.)

   Define `__contains__` and `__iter__`.

   - `__contains__` should just check containment in the underlying dictionary (which checks key containment).

     Adding to the code in 2.(a), the following should result in `True` and `False` being printed.

     ```
     print(3 in d)
     print('3' in d)
     ```

   - `__iter__` should allow us to loop through the underlying dictionary.

     Adding to the code in 2.(a), the following should work as if `d == {1:1, 2:2.0, 3:'3'}`.

     ```
     for k in d:
         print(k, repr(d[k]))
     ```

4. From this point on, we no longer intend to have instances of `StructuredDict`. We only intend to have instances of classes which inherit from `StructuredDict`. We can express this in the language of computer science by saying that `StructuredDict` serves as an *abstract base class*.

   Everything we create(d) for `StructuredDict` is really intended to be made use of by `Rectangle` and `Student`, and if you like `StructuredDict`, Python `structs` of your own design.

5. Fix `__delitem__` and `DeleteError`.

   We don't want to be able to delete items from our `StructuredDict`s, so `__delitem__` should always `raise` an error. Update `DeleteError.__str__` so that the correct error message is displayed: `You cannot delete from a StructuredDict`.

   (If you prefer the approach at the end of `Week4Wed.ipynb` inspired by Kevin's question about `super().__init__`, I am happy with that too.)

6. Fix `__setitem__` and `UpdateValueError`.

   Our concern here is checking that `value` has the correct type. You'll need to access the relevant class variable... First, chase through the function calls that happen when you run:

   ```
   r = Rectangle(2.0, 4.0)
   r['len1'] = 2
   ```

   `r['len1'] = 2` calls `StructuredDict.__setitem__(r, 'len1', 2)`.

   We need to know `Rectangle.key_to_type`.

   To code this, you'll want to use `self.__class__.key_to_type`.

   When the type of value is not correct, you should raise an error. Pass along `key`, `value`, and `self.__class__.key_to_type` to the initializer of `UpdateValueError` and override `__str__` so that the correct error is printed: look at the screenshot above.

   If we were making the best possible `StructuredDict` ever, we would handle key errors (writing to a key that `key_to_type` has told us should never exist) using another exception inheriting from `StructuredDictError`. Let's just let `key_to_type` raise such an error, i.e. let's not worry about this. Notice, similarly, that we haven't "fixed" `__getitem__`: the underlying dictionary is responsible for raising any key errors.

7. Fix `__check` and `InitializationError`.

   My version of `__check` stores as three sets:

   - keys that are missing from `d`;
   - additional keys that `d` has;
   - the keys in `d` associated with a type error.

   (Bear in mind that sets have useful operations: `-`, `&`, and `|`.)

   If there are such problems, it passes this information along to `InitializationError`'s initializer, so that it can print the correct error message.

   Mess with everything until you get the same errors as me. However, don't fluke (British people can make nouns into verbs as they please apparently) the same errors as me. You should think about how your code would work in other cases and if it would still run approrpiately. (Since we're using sets, some lines may appear in a different order.)