Assignment 4

(Spoiler alert, in case you haven't looked at it: my solution to the previous assignment is below.)

In this assignment, we'll build on the previous solution by adding some exception trapping as well as a custom Exception class to handle a particular kind of error. We'll take care of the following tasks:

1. trap bad filename and bad filepath (i.e., if the user passes a pathname to the constructor, but the path doesn't exist or isn't writeable) by testing for these and raising an IOError with a helpful message

2. trap user access of a non-existent key in the dictionary: if this happens (you'll need to implement **\_\_getitem\_\_** to trap this error in the class code), raise a **ConfigKeyError** (a custom exception you'll create in the next task).

3. create a **ConfigKeyError** class that inherits from **Exception**, and when raised, displays the complete list of available keys.

**Overview:** at this point in our work, when building classes that others will use or that might be used within a larger system, we want to consider not just how our library code can be used, but how it could be misused. What could go wrong? It's important to be able to trap and handle user errors because they are the most common, and we have very little control over what a user will do. If we don't handle an error and Python raises an exception that doesn't make sense to the user -- like "line 137: IndexError: list index out of range" -- the user might be forced to dig through our code trying to find out what happened, finally realizing that his or her input was wrong but that our library never detected it or explained the issue.

What I'd like to challenge you to do is to think about your **ConfigDict** class from a usability perspective, and ask what you might do to handle any errors that can occur.

I'd also like to see if you can create a custom error class that can handle the dict **KeyError** by listing out the keys from the config file that do exist.

I'm supplying the test program for you to use to test the class if you wish. This time, there are three ways to call this script: with two arguments to set a key and a value in the dict; with one argument to get the value for a particular key; and with no arguments to see all the keys and values in the dict.

[in a file called assignment3.py]

class ConfigDict(dict):

def \_\_init\_\_(self, filename):

self.\_filename = filename

if os.path.isfile(self.\_filename):

with open(self.\_filename) as fh:

for line in fh:

line = line.rstrip()

key, value = line.split('=', 1)

dict.\_\_setitem\_\_(self, key, value)

def \_\_setitem\_\_(self, key, value):

dict.\_\_setitem\_\_(self, key, value)

with open(self.\_filename, 'w') as fh:

for key, val in self.items():

fh.write('{0}={1}\n'.format(key, val))

[in a file called test.py]

#!/usr/bin/python

"""

Usages:

./test.py (reads out the entire config dict)

./test.py thiskey thisvalue (sets 'thiskey' and 'thisvalue' in the dict)

"""

import sys

from assignment3 import ConfigDict # assumes "assignment3.py" holds a

# class caleld ConfigDict

cd = ConfigDict('config\_file.txt')

# if 2 arguments on the command line,

# set a key and value in the object's dictionary

if len(sys.argv) == 3:

key = sys.argv[1]

value = sys.argv[2]

print('writing data: {0}, {1}'.format(key, value))

cd[key] = value

# if 1 argument on the command line, treat it as a key and show the value

elif len(sys.argv) == 2:

print('reading a value')

key = sys.argv[1]

print('the value for {0} is {1}'.format(sys.argv[1], cd[key]))

# if no arguments on the command line, show all keys and values

else:

print('keys/values:')

for key in cd.keys():

print(' {0} = {1}'.format(key, cd[key]))

**Trap bad filename and bad filepath.** In our previous solution, a nonexistent filename is not a problem since a new file will be created if needed. But what if we supply a \*path\* that doesn't exist? At that point we'll get an exception, and the first thing you'll want to do is find out what sort of exception results from trying to write to a path that doesn't exist.

This creates a slight awkwardness, since in our script as currently written, if the file is new, we won't know if it can't be written to until we actually set a key and a value in the dictionary, which is a bit late to trap this sort of error. Really, we ought to check on the writeability of the file as soon as we construct the object. So my first suggestion is that you alter this solution so that in **\_\_init\_\_**, if the file doesn't exist (you can use **os.path.isfile(filename)**) we open the file or filepath for writing, and then immediately close the file.

**Trap user access of a non-existent key in the dictionary**. Now, without us doing anything, this will result in a **KeyError**, right? And that's a reasonably descriptive error. The user could infer from that error that the config file simply doesn't have that key. But, I'll take this error as an excuse to have you construct a custom Exception class.

Please create a new Exception class called **ConfigKeyError** that is constructed with two arguments: the **ConfigDict** object, and the key that was requested. This error will be raised in the class itself, in the **\_\_getattr\_\_** method, and the **raise** statement will look like this:

def \_\_getitem\_\_(self, key):

if not key in self:

raise ConfigKeyError(self, key)

return dict.\_\_getitem\_\_(self, key)

And if this does happen, the user should see the following message:

key "x" not found. Available keys: (a, c)

**Create the ConfigKeyError class.** This class will accept the **ConfigDict** object and the key that was requested; in **\_\_init\_\_** it should store the key and the list of keys that can be found in the class; and it should also implement a **\_\_str\_\_** method so that when it is raised, **\_\_str\_\_** will print the message as shown above.

Good luck!