Assignment 3

In this assignment we're going to leverage the convenience of a dictionary to power a configuration file, which is simply a file of key-value pairs.

A configuration file is used quite often in programming shops to hold values that don't belong in the Python script itself. Values like SQL queries, email addresses, and other configurable values should be stored outside of the script, because they may change when the code itself doesn't need to be changed.

The structre of a config file could take many forms, and one of them is a simple key=value syntax, with one key/value pair per line. This is simple and straightforward, so we'll use it.

What's great about using a built-in structure like a dictionary as the interface to the configuration file is that any Python programmer will immediately know how to use it. The instructions are so simple you almost don't need documentation: "create a new **ConfigDict** object, then read and write keys and values as desired" -- that's it.

Our config file should looks like this:

sql\_query=SELECT this FROM that WHERE condition

email\_to=me@mydomain.com

num\_retries=5

In order to impelement a dictionary, **ConfigDict** will inherit from **dict** (the dictionary class), so in most respects it will act like a dictionary.

Here's the sample usage:

from assignment3 import ConfigDict # you can put class ConfigDict

# in assignment3.py, or just put the

# class code in the same file as this

# calling code; then this import

# statement is not needed

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

print(cc['query']) # SELECT this FROM that WHERE condition

print(cc['email\_to']) # me@mydomain.com

cc['database'] = 'mysql\_managed' # [ this writes to the config file ]

So in **\_\_init\_\_**, we will:

* open the config file for reading
* parse the keys and values (using split('='))
* load the keys and values from the specified configuration file into the dictionary
* the dictionary object is "self". however, you will be careful not to set keys and values in "self" in the conventional way -- see note below

In **\_\_setitem\_\_**, we will:

* set the key and value in the dictionary
* open the config file for writing
* write the dictionary's key/value pairs to the file
* close the file

It should do this everytime a key and value is set in the dictionary, so that the file will be updated immediately.

In the videos we've discussed the hazards of reimplementing **\_\_setitem\_\_**, but I'll note it here in case you run into it (i.e., see a lot of output followed by a "maximum recursive depth" message):

class DoThis(dict):

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

self[key] = val # OOPS! don't do this

# this calls \_\_setitem\_\_,

# which calls \_\_setitem\_\_, etc.

Think about what happens when we say **self[key] = val**: we're implicitly calling the object's **\_\_setitem\_\_** method, which as you know is called anytime we do this operation. But if we're calling **\_\_setitem\_\_** and we're inside **\_\_setitem\_\_** already, what does this mean? It means we've established a recursive loop, in which **\_\_setitem\_\_** calls **\_\_setitem\_\_** endlessly. Well -- it will end pretty soon, because Python has a "recursive depth" limit, which is reached quickly.

How to resolve this? We can call the parent-class method and it will perform the normal dict behavior. If we pass our own instance as the argument, that's the dict that gets updated.

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

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

# code that writes to the file -- see above

Here's some sample calling code, which will read out the file if called without arguments, but will write to the file if called with key/value arguments

"""

Usages:

./assignment\_3.py (reads out the entire config dict)

./assignment\_3.py thiskey thisvalue (sets 'thiskey' and 'thisvalue' in the dict)

"""

import sys

from assignment\_3 import ConfigDict

cd = ConfigDict('config.txt')

if len(sys.argv) == 3:

key = sys.argv[1]

value = sys.argv[2]

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

cd[key] = value

else:

print('reading data')

for key in cd.keys():

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

Note that because this assignment is potentially involved, I've decided to include an optional supplementary lesson (with accompanying written companion in the working files) with hints and a breakdown of what you need to do to complete this assignment. It's optional so feel free to work without it; or if you'd prefer a real breakdown, check it out. It's the lesson immediately following this one. After that, we'll go over the solution.

Good luck!