New York University
School of Continuing and Professional Studies
Management and Information Technology

Advanced Python

Homework Discussion, Session 4

4.1. **class Config**:

Again thinking about potential code repetition and the work that Python will be expected to do, and considering that we will be reading multiple key/values from the config file, we probably want our **__init__()** constructor to read the file into a dictionary, and then store that dictionary in the instance.

Then when a key's value is requested in the **get()** method, we can simply read from the dictionary.

When **set()** is called, however, we shouldn't just add the key/value to the instance's dictionary: we should also write the key/value to the file so that the file is always up-to-date and we don't have to think about writing to the file later on.

Here are the methods I implemented:

__init__(self, filename, overwrite_keys=True): this method opens the file (traps and then re-raises a **IOError** exception if the file can't be read), loops through the file, splitting out the key/value pairs, and adds them to a new dictionary stored as an attribute in the instance. It also stores the **overwrite_keys** flag in another instance attribute.

get(self, keyname): simply returns the value found for this keyname in the dictionary we stored in the instance in the constructor. If the key can't be found, it raises a **KeyError** exception

set(self, keyname, value): adds the key and value to the instance's dictionary, and then writes the entire key/value set back to the file (I used a separate method called **write_data()** called from this method). **Thus the file is not appended to; it's completely overwritten**. This "wipe and reload" approach is much cleaner than trying to update the file with just one key/value change.

Of course **set()** also needs to see if the key already exists in the dictionary, because if **self.overwrite_keys** is **True** (as set in the **__init__()** constructor) then we should **raise** a **ValueError**.

write_data(self): simply does the work of opening the file and writing each key and value in the dictionary to the file.

4.2. class DatetimeSimple: a class based on the **datetime_simple** module

Not a lot more is required for this class beyond what was done in the **datetime_simple** module.

a. Obviously the object that is constructed in **__init__** has to have an attribute to hold the **datetime.date** object that represents the initialized date.

b. Your object should also have an attribute that specifies its default format (you can choose to store the 'pseudo format' (YYYY-MM-DD or DD-Mon-YY) in this attribute, or for convenience you could store the datetime **strptime()** template string (%Y-%m-%d, etc.). This would default to YYYY-MM-DD but could be changed with the **set_format()** method.

c. The use of the math operators **+** and **-** correspond to the **__add__** and **__sub__** methods and are not challenging to use -- as we discussed, **obj + val** translates to **obj.__add__(val)**. The one gotcha that seems to crop up is understanding that if our interface (the way we use the method) specifies **dts = dts + 5**, we're going to have to return **self** from the **__add__()** method; otherwise, without a **return** statement the method will return **None**, and if we then assign the method call back to **dts**, it will thus become **None**. In cases like these you'll see the error message '**NoneType' object has no attribute 'xxx'**.

d. In the extra credit, we must allow the class to behave as an iterator using **__iter__()**, **next()** and raising the **StopIteration** exception. We'll discuss this more next week, so you can attack this later if you wish -- it is extra credit.