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

Advanced Python

Homework, Session 4

WARMUP EXERCISES

If you are new to user-defined classes, please begin by doing the following practice exercises (not to be submitted) -- the solutions can be found in a new "Exercise Solutions" document on the handouts page. If you attended the Introduction to Python class, you'll recognize these exercises (as well as the homework and extra-credit homework, now demoted to exercises) and can re-solve them at your discretion.

Ex. 4.1 Create a class called **ThisClass** with the statement class **ThisClass(object):** and create one method inside the class with the statement **def report(self):** Inside this method, the instance/object (which is labeled self) should call the special **id()** function to report its own reference id, i.e. **print id(self)**. Create three instances using the constructor (for example, a = **ThisClass()**) and then call the report() method on each of them.

Lastly, **print id()** on each of your objects in the calling code, for example **print id(a)**. Note that the id numbers are the same as those found when calling report().

Expected calls and output:

```
a = ThisClass()
b = ThisClass()
c = ThisClass()

a.report()          # 4299790736  [your ids will of
                    #              course differ]
b.report()          # 4299790544
c.report()          # 4299790800
print               # [blank line]

print id(a)         # 4299790736   (same as a.report() above)
print id(b)         # 4299790544   (same as b.report() above)
print id(c)         # 4299790800   (same as c.report() above)
```

Ex. 4.2 Create a class, **TimeStamp(object):** that can store the current timestamp in an instance attribute.

set_time(self): will set the timestamp. It can do this by setting the attribute in self this way (you will need to **import datetime** at the top of your script containing **class TimeStamp**):

```
self.t = str(datetime.datetime.now())
```

where **t** is the attribute label (you could call it whatever you prefer). (Of course, you'll also need to **import datetime** at the top of your module.)

get_time(self): this returns the timestamp. It simply returns the object attribute time, i.e. **return self.t**.

Expected calls and output (your timestamp will be different of course, but note when the object time is repeated and when it is different):

```
var1 = TimeStamp()
var2 = TimeStamp()
var1.set_time()
var2.set_time()
print var1.get_time()    # 2013-04-07 15:19:31.762220
print var2.get_time()    # 2013-04-07 15:19:31.956881
print                    # [blank line]
var1.set_time()          # change timestamp for var1
print var1.get_time()    # 2013-04-07 15:19:31.956941
                         # (diff from var1 above)
print var2.get_time()    # 2013-04-07 15:19:31.956881
                         # (same as var2 above)
```

Ex. 4.3 Copy the above code, and this time replace the **set_time()** method with the constructor, **__init__(self)**, which does the same work - sets the attribute to the current timestamp. Now the method has only an **__init__(self)** method and a **get_time(self)** method which returns the timestamp.

Expected calls and output (your timestamp will differ of course, but note when the object time is repeated and when it is different):

```
var1 = TimeStamp()
var2 = TimeStamp()
print var1.get_time()    # 2013-04-07 15:19:31.762220
print var2.get_time()    # 2013-04-07 15:19:31.956881
print
print var1.get_time()    # 2013-04-07 15:19:31.762220
                         # (same as previous var1)
print var2.get_time()    # 2013-04-07 15:19:31.956881
                         # (same as previous var2)
```

Ex. 4.4 Create a class, **Square**, each of whose instances (i.e., each **Square** object) have an attribute which is an integer. Each **Square** object has the following methods:

a. **__init__()**: set the integer attribute to 2.
b. **def squareme(self)**: square the integer attribute and store the new squared value in the attribute, replacing the old value (i.e. the first time you call it, it will replace 2 with 4.
c. **def getme(self)**: return the integer attribute.

Sample usage (your code must be able to behave as shown below, including being able to create multiple arguments with their own data -- PLEASE MAKE SURE YOUR OBJECTS HOLD THEIR OWN DATA -- MAKE SURE THEY CAN BEHAVE EXACTLY AS SHOWN BELOW, ESPECIALLY THE FINAL LINES OF THE EXAMPLE USAGE).

Also, make sure you can create any number of objects, not just three!

```
# all output must match!
n1 = Square()
n2 = Square()
n3 = Square()

n1.squareme()
val1 = n1.getme()
print val1                # 4

n2.squareme()
n2.squareme()

val2 = n2.getme()
print val2                # 16
print n1.getme()          # 4

n3.squareme()
n3.squareme()
n3.squareme()
n3.squareme()
n3.squareme()

val3 = n3.getme()
print val3                # 4294967296

print n1.getme()          # 4
print n2.getme()          # 16
print n3.getme()          # 4294967296
```

Expected output:

```
4
16
4
4294967296
4
16
4294967296
```

Ex. 4.5 List of a Maximum Size. Create a class, **MaxSizeList**, that constructs instances that have a list as an attribute (set as an empty list in **__init__**). The constructor takes an integer as an argument, which becomes the maximum size of the list (stored as a second attribute).

Add a method, **push()**, which appends a value to the list. However, if the list has already reached its maximum size (as set in the constructor), remove the first element from the list before appending the new element to the list. (You can use the list method **pop()** with an argument of 0 to remove the first element of the list, or you can use a slice (**alist = alist[1:]**).)

Expected output:

```
a = MaxSizeList(3)
b = MaxSizeList(1)

a.push("hey")
a.push("ho")
a.push("let's")
a.push("go")

b.push("hey")
b.push("ho")
b.push("let's")
b.push("go")

print a.get_list()     # ['ho', "let's", 'go']
print b.get_list()     # ['go']
```

HOMEWORK

NOTE ON ACCESSING CLASS VARIABLES: some variables belong to the class ("class variables") but are not set in individual instances. Such variables are defined in the class and can only be accessed through the class name:

```
class MyClass(object):
    base_dir = '/my/path'

    def store_dir(self, custom_path):
        self.dir = MyClass.base_dir + '/' + custom_path
```

In other words, you would not access **base_dir** through its bare name, because Python will look for it in the global space rather than in the class' data - so you must reference it as an attribute of the class name. See slides on **Class Data**.

NOTE ON EXCEPTIONS: library code usually does not **exit()** on its own; that's because we want to allow the *calling code* to decide what to do in case of error. We document the type of Exception that may occur under certain conditions, and we **raise** the Exception ourselves, usually with a message that is more appropriate to the module:

```
try:
    fh = open('xxxx.txt')
except IOError, e:
    raise IOError("config file can't be opened:  {}".format(e))
```

Some sample output of the above:

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 4, in <module>
IOError: config file can't be opened:  [Errno 2] No such file or
directory: 'xxxx.txt'
```

To be clear, we don't only raise exceptions that Python has already raised and we have trapped; we can raise our own at any time based on our own program's logic.

In this particular case, we are responding to an **IOError** that Python raises when the OS can't find the file we're attempting to open. And, we're going right ahead and re-raising the same exception. But we're adding our own more descriptive error message.

Note the use of variable **e**: in this case this variable contains the error message that was passed from the OS to Python and then to the exception; it is the reason the file can't be opened (might be not found, or might be lacking permissions). So we include this string in our own error message.

Examples shown have placed the class in a module called **mylib**, thus the **from mylib import ...** statement.

4.1. class Config: this class reads and writes configuration text files that may also be hand-editable.

a. It reads a configuration file which can be in one of the following formats:

i. CSV (use the **csv** module)

ii. JSON (use the **json** module)

iii. A simple **key=value** format, as follows:

```
data_query=SELECT this, that FROM mytable WHERE col = 5
db_name=data_db
db_username=dbuser
db_password=458abJk!9
email_to=joe@wilson.com
```

Note the equals sign in the value for **data_query**! Use the **str.split()** method's MAXSPLIT parameter to avoid splitting that string into 3 pieces.

b. Reading from the above config file, the following code provides convenient access through the **get()** method:

```
from mylib import Config

conf = Config('myfile.txt', overwrite_keys=True)

print conf.get('db_username')   # dbuser

query = conf.get('data_query')  # SELECT this, that FROM ...
```

We can also set a value in the config file using the **set()** method. Rather than require an additional **write()** method, we should open, rewrite and close the config

file every time a key/value is set in the file. (Don't think about just changing the one key -- simply overwrite the existing file with the object's current set of keys and values.)

```
conf.set('confkey1', 'newval')
```

If the key already exists, overwrite the value for this key only if overwrite_keys was specified in the constructor arguments, otherwise raise a **ValueError** exception with a custom message.

Exceptions:

If the file cannot be opened, raise an **IOError** exception with a custom message.

If the file cannot be written, raise an **IOError** exception with a custom message.

If the key does not exist, raise a **KeyError** exception with a custom message.

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

Last week's **datetime_simple** module required that we pass **datetime.date** objects between calls. The class **DatetimeSimple** will allow us to create an object that hides **datetime.date** functionality behind its interface -- allowing us to work exclusively with **DatetimeSimple** objects without having to consider **datetime.date** objects at all:

```
from mylib import DatetimeSimple

# creating a DatetimeSimple object
dts = DatetimeSimple('today')
dts = DatetimeSimple()                    # today, same as above
dts = DatetimeSimple('2016-06-06')
dts = DatetimeSimple('06-Jun-2016')
dts = DatetimeSimple('06/06/2016')
dts = DatetimeSimple('some bad format')  # raises ValueError

dts = dts + 1  # add one day to the date: returns the object
dts = dts - 5  # subtract 5 days from the date: returns object

print dts                    # 2016-06-02  (default format)
dts.set_format('MM/DD/YYYY')
print dts                    # 06/02/2016

# extra credit:  looping through successive dates
for date_str in dts.daterange(interval=7, len=5):
    print date_str                                  # '06/02/2016'
                                                    # '06/09/2016'
                                                    # '06/16/2016'
                                                    # '06/23/2016'
                                                    # '06/30/2016'
```