Classes and OOP

Objectives

To build a simple class using Python's object orientation, with particular emphasis on special functions and overloaded operators.

Reference Material

Chapter 11 Classes and OOP. Chapter 04 String Handling might also be useful.

Question 1

We are going to build a class for the records in `country.txt`. Each object will represent one country. The class will be called Country in **`Country.py`**.

The **`Country.py`** module has been started for you. It contains an index list giving the fields for each record in the `country.txt` file. You do not have to use this, there are many valid approaches. The comma-separated fields are as follows:

0 – Country name
1 – Population
2 – Capital city
3 – Population of the capital city
4 – Continental area
5 – Date of independence
6 – Currency
7 – Official religion (can be > 1)
8 – Language (can be > 1)

a) Take a look at the script **`user.py`**. This imports the Country module and reads the `country.txt` file. It creates a Country object for each record in the file, and stores it into a list called **`countries`**.

   Each part of this question has a test in **`user.py`**, prefixed by a suitable comment. Currently all except the first one are commented out. Remove the # symbols at the front of the tests as you progress.

   So, to get the **`user.py`** code to run as it is your first task is to implement a constructor (**`__init__`**) within the **`Country.py`** module.

   We suggest that you keep it simple, and implement the object as a list of fields, created using `split()`.

b) The next task is to implement a **`printit`** method to print just the country name. Now, in **`user.py`**, remove the comments from the start of the second '**`for`**' loop and the first method call.

c) It would be easier for the user if the normal Python `print` could be used to print our object instead. Implement the **`__str__`** special method in

`Country.py` to return the country name.

In **`user.py`** replace the call to the `printit` method call with **`print country`**.

d) It is rather unwieldy to access the elements of the list in our methods. To make things simpler, implement two **getter methods**, one for the country name and another for the population field.

You can use the **`@property`** decoration – refer to the "**Properties and decorators**" slide in the course material.

Alter your **`__str__`** special method to use the getter method for name.

Remove the comments for this question from **`user.py`**, and the `printit` statement for part c), then test.

e) The population totals for these countries are out of date as soon as the raw data is generated, so we need to be able to add or subtract numbers to these countries.

Write the special methods **`__add__`** and **`__sub__`** to add or subtract the required number to the country's population (we are ignoring the capital city for this exercise).

That sounds easy! Actually there is a sting in the tail of this one. You might have to alter the population field in the constructor to get this to work.

Hints:

when we read fields from a file they are *strings*.
we don't want to alter `self`, we want to alter (and return) a copy of `self`.

Uncomment the appropriate tests in **`user.py`** and run it. We are manipulating the population of Belgium in the test for no particular reason (apologies to Belgians – but we had to pick *somewhere*).

If time allows:

f) The **`user.py`** script holds a list of country objects, but we have no way of finding a particular country without printing it. We would like to use an **`index()`** method to search for a country name. We do not actually write an `index()` method for this but instead overload the **==** operator with the **`__eq__`** special method.

   Again, uncomment the tests for this question and run them.

g) The fickle users have now decided that the **`__str__`** special function should output the population as well, and in a nice formatted string. The country name should be left justified, with a minimum field wide of 32 characters, and be followed by a space, then the population right justified, zero padded, with a minimum width of 10 characters.

   Remember formatted strings? There were in the String Handling chapter.

Question 2

Construct a class called **File**. The constructor should take one parameter, a file name, and should create the file if it does not exist. Store the filename as an object attribute which is private to the *module*.

The **File** class should also have a property called **size** which gives the size of the file in bytes (use `os.path.getsize()`).

Question 3

Within the same module, add two further classes derived from **File**.

a. TextFile

Has a property called **contents**.
The getter method returns the whole contents of the file as a string.
The setter method appends text to the file.
Both methods will need to open the file

b. BinFile

Has a property called **contents**.
The getter method returns the whole contents of the file as a decoded string.
The setter method appends data to the file. If the data is of class int (use `isinstance()` to check) then use `struct.pack()` to pack the integer into a binary format. If the data is of any other class then write it and an encoded string.
Both methods will need to open the file as *binary*.

Write suitable tests to write and read data to and from files, and check that the size method can be called from objects of the derived class.

Question 1 solution

As always there are many possible implementations, but here is ours:

```
import copy
class Country(object):
    index = {'name':0,'population':1,'capital':2,'citypop':3,
             'continent':4,'ind_date':5,'currency':6,
             'religion':7,'language':8}

    # Insert your code here
    # 1a) Implement a constructor
    def __init__(self, row):
        self.__attr = row.split(',')

        # 1e) Added to support + and -
        self.__attr[Country.index['population']] = \
            int(self.__attr[Country.index['population']])

    # 1b) Implement a print method
    def printit(self):
        print self.__attr[Country.index['name']]
        return self.name

    # 1c) Overloaded stringification
    def __str__(self):
        #return self._attr[Country.index['name']]
        # 1g) Formating the output
        return "%-32s %010s" % \
             (self.name, self.population)

    # Getter methods
    # 1d) Implement a getter method for country name
    @property
    def name(self):
        return self.__attr[Country.index['name']]

    @property
    def population(self):
        return int(self.__attr[Country.index['population']])

    # 1e) Overloaded + and -
    def __add__(self,amount):
        retn = copy.deepcopy(self)
        retn.__attr[Country.index['population']] += amount
        return retn

    def __sub__(self,amount):
        retn = copy.deepcopy(self)
        retn.__attr[Country.index['population']] -= amount
        return retn

    # If time allows:
    # 1f)  Overloaded == (for index search)
    def __eq__(self,key):
        return (key == self.name)
```

Non-decorator solution

Getter methods for name and population without using properties

```
def name_get(self):
    return self.__attr[Country.index['name']]

name = property(name_get)

def population_get(self):
    return int(self.__attr[Country.index['population']])

population = property(population_get)
```

If time allows:

Questions 2 & 3 solutions

```
import os.path
import struct

class File(object):
    def __init__(self,filename):
        self._filename = filename

        # If the file does not exist, create it
        if not os.path.isfile(filename):
            open(filename,'w')

    @property
    def size(self):
        return os.path.getsize(self._filename)

# Text file
class TextFile(File):
    @property
    def contents(self):
        """ Return the contents of the file """
        return open(self._filename,'rt').read()

    @contents.setter
    def contents(self,value):
        """ Append to the file """
        if not value.endswith("\n"):
            value += "\n";
        open(self._filename,'at').write(value)

# Binary file
class BinFile(File):
    @property
    def contents(self):
        """ Return the contents of the file """
        value = open(self._filename,'rb').read()
        return value

    @contents.setter
```

```
        def contents(self,value):
            """ Append to the file """
            if isinstance(value,int):
                out = struct.pack('i',value)
                open(self._filename,'ab').write(out)
            else:
                open(self._filename,'ab').write(value)


    if __name__ == '__main__':
        file1 = TextFile('file1.txt')
        file1.contents = 'hello'
        file1.contents = 'world'

        print file1.contents
        print "Size of file1:",file1.size

        file2 = BinFile('file2.dat')

        file2.contents = 42
        file2.contents = 34
        file2.contents = 'EOD'

        print file2.contents
        print "Size of file2:",file2.size
```