Exercise 11 – Classes and OO

Objective

To build a simple class using Python's object orientation. The first exercise will take you through the process of creating a simple file class step-by step. Most of the code is provided for you. The second exercise uses inheritance and is a little more challenging – you will have to figure out some of the code yourself.

Subsequent exercises are optional and are suitable only for experienced programmers.

Questions

1. This exercise will take you step-by-step into building a class based on simple file IO.

    a. Create two files in the same directory:
        myfile.py   this will contain the class **MyFile**
        runfile.py   this will import myfile and test the class

        In **myfile.py**, create a dummy class:

        ```
        class MyFile:
            pass
        ```

        In **runfile.py**, import the module, create an object, and print it:

        ```
        from myfile import MyFile

        filea = MyFile()
        print(filea)
        ```

        Execute **runfile.py**. Did you get the output you expected?

    b. Now we will add a constructor to the MyFile class. It will require a file name to be input, which will be stored as an object attribute. In **myfile.py** alter the class to say:

        ```
        class MyFile:

            def __init__(self, filename):
                self.__fname = filename
        ```

Now alter **runfile.py** to construct the object so that it passes the name of a text file. The file should exist, preferably in the current directory.

```
from myfile import MyFile

filea = MyFile('country.txt')
print(filea)
```

Execute **runfile.py**. The output should look the same as before.

c. Now we will make the print do a little more work, by providing a __str__ special function. In **myfile.py** add a __str__ function that will read the file into a string, and return it:

```
class MyFile(object):

    def __init__(self, filename):
        self.__fname = filename

    def __str__(self):
        s = open(self.__fname).read()
        return s
```

Test the code by executing **runfile.py**, the text file should be displayed.   Note: this is only an exercise! In reality it would be foolhardy to blindly read a file into a string – it might be so large that it exceeds python's memory allocation.

d. If we wanted to check the size of the file, we might wish to use the **len()** built-in. For that we can provide a __len__ method, to be added to **myfile.py**. This will require the **os.path** module, which we should import at the start of the module (outside the class).

We will also add a simple *getter* function to the class to return the filename, called get_fname. The class should now look something like this:

```python
import os.path

class MyFile(object):

    def __init__(self, filename):
        self.__fname = filename

    def __str__(self):
        s = open(self.__fname).read()
        return s

    def __len__(self):
        return os.path.getsize(self.__fname)

    def get_fname(self):
        return self.__fname
```

Add a new test to **runfile.py**:

```python
from myfile import Myfile

filea = Myfile('country.txt')
print(filea)

print(filea.get_fname(), "is", len(filea), "bytes")
```

Execute **runfile.py** and check the output.

2. Create a new module called **inherit.py**. In the new module, import the **MyFile** class from the **myfile** module. We are going to create two further classes derived from **MyFile**.

Class **TextFile** will be for processing text files, and **BinFile** for processing binary files. In **inherit.py** create these classes as being empty, inherited from **MyFile**. For example:

```
class TextFile(MyFile):
    pass
```

Do the same with the BinFile class.

We can now add some tests. This time we will incorporate these tests in the **inherit.py** module itself, and run it as main – a common pattern.

Add the "main" trap after the code for the new classes and do some simple tests. We have provided two *empty* files for these tests, **file1.txt** (text) and **file2.dat** (binary). Let's first check that the inheritance works, for example:

```
if __name__ == '__main__':
    file1 = TextFile('file1.txt')
    print(file1, len(file1))

    file2 = BinFile('file2.dat')
    print(file2, len(file2))
```

Assuming we have that working, we can move on to some more advanced operations using *properties*. Renew your acquaintance with properties by reviewing the slides if necessary.

Start by removing the "pass" from both classes.

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, and should add a newline to the end of the text if none is present.

Both methods will need to open the file, so we need to call `get_fname()`. For example:

```
@property
def contents(self):
# Return the contents of the file
    return open(self.get_fname(),'rt').read()

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

b. BinFile

Has a property called **contents**.

The getter method returns the whole contents of the file as a decoded string. It should be fairly straightforward to write, so *have a go yourself*.

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 (so don't forget to `import struct`). If the data is of any other class then write it and an encoded string. That is a little complicated, so here is the setter method:

```
@contents.setter
def contents(self,value):
    # Append to the file
    if isinstance(value,int):
        out = struct.pack('i',value)
        open(self.get_fname(),'ab').write(out)
    else:
        open(self.get_fname(),'ab').write(value.encode())
```

The `str.encode()` is required because, being binary, we need to write a bytes object.

Both methods will need to open the file as *binary*.

Write suitable tests to write and read data to and from files. Here are some suggestions:

```python
if __name__ == '__main__':
    file1 = TextFile('file1.txt')
    print(file1, len(file1))

    file1.contents = 'hello'
    file1.contents = 'world'

    print(file1.contents)
    print("Size of file1:", len(file1))

    file2 = BinFile('file2.dat')
    print(file2, len(file2))

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

    print(file2.contents)
    print("Size of file2:", len(file2))
```

If time allows:

3. In an earlier exercise we created a generator called `frange` in a file called `gen.py`. The `frange()` function tried to emulate the standard built-in range(), but it falls well short of the task. The built-in `range()` function returns `range` objects on which you can perform various tasks.

    In this exercise we will put our function into a class and create **Frange** objects. Note that the name of the class is capitalized to conform to PEP008, which strictly speaking `range()` does not.

    Take a copy of `gen.py` (or whatever file your `frange()` solution resides in) to create a new file called `range.py`.

    Of the stages that follow, consider stages 2 onwards to be optional.

    Stage 1

    We can't just slap a class statement around the existing frange() function, we need to split it into two. We have the initialisation part and the generator part.

    Create a class called Frange and within it create two functions:

    `__init__` will take the same parameters as `frange()` did, plus the initial `self`. It will perform initialisation (conversion to Decimal) and add the stop, start and step attributes.

    `__iter__` will only take the parameter self. It will consist of the iterator part of the old `frange()` function, that is the loop including the `yield`.

    Alter the tests so that `Frange()` is called rather than `frange()`.

Stage 2

Support the `len()` built-in. In the constructor (`__init__`) we will add a length attribute. This can be done in the constructor because a range is immutable, and there are a number of operations which require it. The length of the range can be determined with:

```
self.len = math.ceil((self.stop - self.start)/self.step)

if self.len < 0: self.len = 0
```

Now add a `__len__` method which returns the length attribute. Write a suitable test, here is a suggestion:

```
if __name__ == '__main__':
    def printit(r):
        print(r, len(r), list(r))

    printit(Frange(1.1, 3))
    printit(Frange(1, 3, 0.33))
    printit(Frange(1, 3, 1))
    printit(Frange(3, 1))
    printit(Frange(-1, -0.5, 0.1))
    printit(Frange(1, 3, 0))
```

Stage 3

Right now, when we print an `Frange` object we don't get anything which is useful. So implement the `__repr__` method into your `Frange` class. The string returned should be a command suitable for recreating the object, for example `Frange(3.0, 1.0, 0.25)`. This assumes that `start`, `stop`, and `step` are attributes of `self`, so you might have to alter the constructor.

The class name should be obtained from `self`, not hard-coded (hint: `self` has an attribute called `__class__`, which has an attribute called `__name__`).

Either write new tests, or use those written previously.

Optional Extension:

<u>Stage 4</u>

Comparing two Frange objects could be done in by brute force, i.e. by comparing each element, but there is a simpler way. Simply compare the start, stop, and end positions used by the constructor. This assumes that they are attributes of `self`, which we added in the previous stage.

Add a `__eq__` method with suitable tests. We could also add the others, `__ne__`, `__gt__`, and so on, but consider if these would be appropriate.   If we just implement `__eq__`, do we need a `__ne__`? Try both `==` and `!=` with only an `__eq__` method.

If time allows:

4. 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 **`print()`** 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 built-in `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 `print` 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` decorator – 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 `print` 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*).

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 **format**. 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.

Solutions

1. This is the final myfile.py:

```python
import os.path

class MyFile:
    def __init__(self,filename):
        self.__fname = filename

    def __str__(self):
        s = open(self.__fname).read()
        return s

    def __len__(self):
        return os.path.getsize(self.__fname)

    def get_fname(self):
        return self.__fname
```

   runfile.py:

```python
from myfile import MyFile

filea = MyFile('country.txt')
print(filea)

    print(filea.get_fname(), "is", len(filea), "bytes")
```

2. This is the complete inherit.py file:

```python
import struct
from myfile import MyFile

# Text file
class TextFile(MyFile):

    @property
    def contents(self):
        # Return the contents of the file
        return open(self.get_fname(),'rt').read()

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

```python
# Binary file
class BinFile(MyFile):

    @property
    def contents(self):
        # Return the contents of the file
        value = open(self.get_fname(),'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.get_fname(),'ab').write(out)
        else:
            open(self.get_fname(),'ab').write(value.encode())

if __name__ == '__main__':
    file1 = TextFile('file1.txt')
    print(file1, len(file1))

    file1.contents = 'hello'
    file1.contents = 'world'

    print(file1.contents)
    print("Size of file1:", len(file1))

    file2 = BinFile('file2.dat')
    print(file2, len(file2))

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

    print(file2.contents)
    print("Size of file2:", len(file2))
```

3. Here is the complete Frange class. We have included a __getitem__ method for your
    interest which you were not asked (or expected) to provide.

```python
import math
import decimal

class Frange:

    def __init__ (self, start, stop=None, step=0.25):
        self.step = decimal.Decimal(str(step))

        if stop is None:
            self.stop = decimal.Decimal(str(start))
            self.start = decimal.Decimal(0)
        else:
            self.stop = decimal.Decimal(str(stop))
            self.start = decimal.Decimal(str(start))

        # Calculate length
        if self.step == 0:
            self.len = 0
        else:
            self.len = math.ceil((self.stop - self.start)/self.step)
            if self.len < 0: self.len = 0

    def __len__(self):
        return self.len

    def __getitem__(self, index):

        if index < 0:
            index = self.len + index

        if index >= self.len or index < 0:
            raise IndexError("index out of range")

        item = self.start + (index * self.step)
        return item

    def __repr__(self):
        retn = \
        f"{self.__class__.__name__:}({float(self.start):}," + \
        f" {float(self.stop):}, {float(self.step):})"
        return retn


        def __eq__(self, rhs):
```

```python
        if self.start == rhs.start and \
           self.stop  == rhs.stop  and \
           self.step  == rhs.step:
            return True
        else:
            return False


    def __iter__(self):
        # Will return None on an empty list
        if self.step != 0:
            curr = self.start
            while curr < self.stop:
                yield float(curr)
                curr += self.step
```

If time allows:

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

```python
import copy
class Country:
    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']])

    # 1c) Overloaded stringification
    def __str__(self):
        #return self.__attr[Country.index['name']]
        # 1g) Formatting the output
        return "{0:<32} {1:>010}".
            format(self.__attr[Country.index['name']],
                self.__attr[Country.index['population']]))

    # Getter methods, using the @property decorator
    # See below for a non-decorator solution
    # 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)
```

Further, if time allows…

2, 3

```python
import os.path
import struct

class File:
    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(afile):
    @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.decode()

    @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.encode())
```

```
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)
```