CSCE 1035 Lab 09 Fall 2021

1. **zyBooks Labs**

   Please follow the link on Canvas to complete the following zyBooks labs:

   - 9.11 *LAB: Car value (classes)*
   - 9.12 *LAB: Nutritional information (classes/constructors)*
   - 9.13 *LAB: Artwork label (classes/constructors)*
   - 9.14 *LAB: Triangle area comparison (classes)*
   - 9.15 *LAB: Winning team (classes)*
   - 9.16 *LAB*: Warm up: Online shopping cart (Part 1)*

   This portion of the lab will be worth one-half of your Lab 09 grade.

2. **Creating a Clock Class**

   In this lab component you will write a small, but complete Python 3 program called **`Lab9A.py`** that creates our own data type to represent time as a clock. Each `Clock` instance will represent the present time in hours, minutes, and seconds, which means that each instance will have three data attributes: `hours`, `minutes`, and `seconds`. This can be accomplished as follows:

   a. Let's first create a version of the `Clock class` that essentially does nothing using the `pass` keyword, since Python does not allow an empty function/method or `class`.

   ```
   class Clock:
        pass
   ```

   Anything defined inside the `class` must be indented. Let's create an instance of this `Clock class` outside of the `class` to see what it already does:

   ```
   my_clock = Clock()  # calls the constructor
   print('My clock is: ', my_clock)
   print('Type is:', type(my_clock))
   print(dir(my_clock))
   ```

   Run the above code to see what is print out. Even though we did not create a constructor (yet!), a constructor was created for us that essentially made a `Clock` instance. You will recognize that the `dir` command showed a lot of the double underlined elements, such as `'__class__'` that Python automatically provides for us.

   b. Now, instead of `pass`, let's build our own constructor that adds the three attributes to each instance, initializing them after Python creates the default instance as follows:

   ```
   def __init__(self, hours = 0, minutes = 0, seconds = 0):
        self.hours = hours
        self.minutes = minutes
   ```

```
    self.seconds = seconds
```

Python uses the `__init__()` as the constructor. All methods, including the constructor, must have `self` as the first parameter, which allows the `class` to refer to itself. Our constructor also has three *default* parameters so that our default instance will have all attributes set to 0.

The `self.hours = hours` statement means that the default instance, now associated with the variable `self`, will have created at attribute called `hours` within that instance. The value of that attribute will be set to the value of the parameter `hours`. Note that it should be clear that, for the variable `hours`, which is the parameter and which is the attribute. The attribute shows up as part of an instance and is scoped with `self`, while the parameter has no such instance.

If you check out the `dir` command again, you will notice that the `class` now contains the `__init__` constructor as well as the attributes `hours`, `minutes`, and `seconds` listed in the results.

In addition to our `my_clock` instance, let's also create a new `Clock` instance with some non-default values for the attributes to see what is printed:

```
new_clock = Clock(10, 15, 30)
print(new_clock.hours, new_clock.minutes, new_clock.seconds)
```

You should now notice that the `Clock` class provides a little more useful information!

c. We can modify our constructor to deal with clock arithmetic so that no hour will be greater than 23 and no minute or second will be greater than 59. When a sum is larger than the limit, a carry is added (although no carry is needed for `hours` since there is no day represented). Make the following changes to the constructor:

```
self.seconds  = seconds % 60
total_minutes = minutes + (seconds // 60)
self.minutes  = total_minutes % 60
self.hours    = (hours + (total_minutes // 60)) % 24
```

Notice that local variable `total_minutes` was included without prepending `self`, which means that `total_minutes` will be discarded when `__init__()` method goes out of scope, instead of persisting with the object.

We can add another `Clock` instance to check out what happens when adding an extra second to the `seconds` data member when instantiating a `Clock` object.

```
tst_clock = Clock(23, 59, 60)
print(tst_clock.hours, tst_clock.minutes, tst_clock.seconds)
```

You should notice that `tst_clock` prints all 0's. Make sure you understand why!

d. So far, printing our `Clock` objects has been fairly simple and plain. Let's add a member function called `print_clock()` that will print a better "formatted" `Clock` object. Inside the `Clock` class, add the following code:

```
def print_clock(self):
     print('{:02d}:{:02d}:{:02d}'.format(self.hours,
self.minutes, self.seconds))
```

The `print_clock()` method does not take any arguments, but it does have one parameter, the instance that we are printing (i.e., `self`). We grab the three attributes from `self` and print them. Replace your `print()` statements for `new_clock` and `tst_clock` with the object function call to `print_clock()`, passing no arguments.

e. Finally, let's add a new method called `add_clocks()` that allows us to add two `Clock` objects, generating a new `Clock` object. This new method should not modify the existing `Clock` instances, only return a new `Clock` object. This means that we should call the constructor inside the `add_clocks()` method. When we make a new `Clock` instance, we do not have to enforce restrictions on the attribute values. We can let the constructor do that for us. Just like the `print_clock()` method, we add the following method definition to the `Clock` class:

```
def add_clocks(self, clock2):
     seconds = self.seconds + clock2.seconds
     minutes = self.minutes + clock2.minutes
     hours   = self.hours   + clock2.hours
     return Clock(hours, minutes, seconds)
```

Notice here that we build local variables `hours`, `minutes`, and `seconds` using the instance variables from `self` and `clock2` and then use them to create a new `Clock` instance that is returned (i.e., the new `Clock` object).

We can use test our new method out by adding the following to our program:

```
c1 = Clock(10, 59, 59)
c2 = Clock (1, 1, 1)
c3 = c1.add_clocks(c2)
c1.print_clock()
c2.print_clock()
c3.print_clock()
```

Notice how `Clock` objects `c1` and `c2` remain unchanged. Our `c3` `Clock` object was created using the attributes from `c1` and `c2`. Notice that `c1` is `self` and `c2` is acting as `clock2` in the `add_clocks()` method call.

For example, the output should look like this (input shown in **bold**):

```
$ python3 Lab9A.py
My clock is:  <__main__.Clock object at 0xb78bf62c>
Type is: <class '__main__.Clock'>
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'add_clocks', 'hours', 'minutes', 'print_clock', 'seconds']
10:15:30
00:00:00
10:59:59
01:01:01
12:01:00
```

Note that you will submit this file to Canvas.

Now that you have completed this lab, it’s time to turn in your results. Once you've moved the files to your windows machine (using **WinSCP**), you may use the browser to submit them to Canvas for the **Lab 09** dropbox.

You should submit the following files:

- **`Lab9A.py`**
- **(Note that the zyBooks labs are submitted separately through Canvas.)**

You may want to ask your TA to check your results before submission.

Now that you've finished the lab, use any additional time to practice writing simple programs out of the textbook, lectures, or even ones you come up with on your own to gain some more experience.