Clock Example

Consider the logic for a digital 24 hour clock object, type `Clock`, that shows hours and minutes, so 03:45 would be three forty-five. Note that there is no AM or PM: The hours go from 00, starting at midnight, through hour 23, the 11PM hour, so 23:59 would be a minute before midnight, and 13:00 would be 1PM.

Assume there is some attached circuit to signal when a new minute starts.

This class could have just a few methods: `tick`, called when a new minute is signaled, and `getTimeString` to return the time in the format illustrated above, and `setTime` specifying new values for the hours and minutes. We can start from a constructor that just sets the clock's time to midnight.

We can imagine demonstration code containing:

```
clock = Clock()
print('midnight', clock.getTimeString())
clock.setTime(23, 58)
print('before midnight', clock.getTimeString())
for i in range(4):
    clock.tick()
    print('tick', clock.getTimeString())
```

It should print

```
midnight 00:00
before midnight 23:58
tick 23:59
tick 00:00
tick 00:01
tick 00:02
```

A `Clock` object will need instance variables. One obvious approach would be to have `int` instance variables for the hours and minutes. Both can be set and can advance unitl they roll over, and will need to be read.

These actions are common to both the hours and minutes, so we might think how we can avoid writing some things twice. There is one main difference: The minutes roll over at 60 while the hours roll over at 24. Though the limits are different, they are both numbers, so we can store the limit for each, 60 or 24. Then the same code could be used to advance each one, just using a different value for the rollover limit.

How would we neatly code this in a way that reuses code? The most significant thing to notice is that dealing with minutes involves data (the current count and the limit 60) and associated actions: being set, advanced and read. The same is true for the hours. The combination of *data and tightly associated actions*, particularly used in more than one situation, suggests a new class of objects, say `RolloverCounter`.

Notice the shift in this approach: The instance variables for hours and minutes would become instances of the `RolloverCounter` class. A `RolloverCounter` should know how to advance itself. Hence the logic for advancing a counter, sometimes rolling it over, would not be directly in the `Clock` class, but in the `RolloverCounter` class.

So let's think more about what we would want in the `RolloverCounter` class. What instance variables? Of course we have the current count, and since we want the same class to work for both minutes and hours, we also need to have the rollover limit. They are both integers.

The limit should just be set once for a particular counter, presumably when the object is created. For simplicity we can just assume the count is 0 when a `RolloverCounter` is first created. Of course we must have a method to let the count advance, rolling over back to 0 when the limit it reached.

Throw in a getter and a setter for the count and we can have the following class:

```
class  RolloverCounter:
    '''count - current value
       limit - rolls over before this
    '''

    def __init__(self, limit):
        self.limit = limit
        self.count = 0

    def getCount(self):
        return self.count

    def setCount(self, count):
        self.count = count

    def advance(self):
        '''Advance by one time tick; eventually roll over at limit'''
        self.count = (self.count + 1) % self.limit
```

Note how concise the `advance` method is! With the remainder operation, we do not need an `if` statement. Check examples by hand if this seems strange.

Finally we introduce the `Clock` class itself. We display one version of the entire code first, and follow it with comments about a number of new features:

```
class Clock:
    '''simulate 24 hour clock using RolloverCounters hours, minutes'''

    def __init__(self, nHours=0, nMinutes=0): # midnight with no parameters
        self.hours = RolloverCounter(24)
        self.minutes = RolloverCounter(60)
        self.setTime(nHours, nMinutes)

    def setTime(self, nHours, nMinutes):
         self.hours.setCount(nHours)
         self.minutes.setCount(nMinutes)

    def tick(self):
        '''advance by one time tick'''
        self.minutes.advance()
        if self.minutes.getCount() == 0:
            self.hours.advance()

    def getTimeString(self):
        return (str(self.hours.getCount()).rjust(2, '0') + ':' +
                str(self.minutes.getCount()).rjust(2, '0'))
```

1. First the principal reason for this example: We illustrate writing a class where the instance variables are objects of a different user-defined type. Because the instance variables `hours` and `minutes` are objects, we initialize them using the constructor for the `RolloverCounter` class.
2. We provide optional parameters, so we can concisely set the initial time to midnight, a particular hour, or with both the hour and minutes set explictly.
3. The `tick` method has a bit of logic to it: while the minutes always advance, the hours only advance when the minutes roll over to 0.
4. The complication in `getTimeString` is that both the number of hours and minutes are always displayed as two digits, with a leading 0 as necessary. The string method `rjust` (mentioned in the string methods section) can handle this. The integers values from the hours and minutes must be explicitly changed to strings to apply the `rjust` method.

See `clock.py` for all the code together.

Admittedly, with this exact functionality and such a concise line to advance a count, it would actually have shorter to have done everything inside the `Clock` class, with no `RolloverCounter` , but we were looking for a simple illustration of combining user-defined types this way, and a `RolloverCounter` is a clear unified concept that can be used in other situations.

Clock With Seconds Exercise

Modify `clock.py` , assuming the tick is for each second, and the time also show the seconds, like 55 seconds before midnight would be 23:59:05.

Twelve Hour Time Exercise

Modify `clock.py` so a `GetTimeString12` method returns the 12 hour time with AM or PM, like 11:05PM or 3:45AM. (The hours do not have a leading 0 in this format.) This could be done modifying a lot of things: keeping the actual hours and minutes that you will display and remembering AM or PM (with the hours being more complicated, not starting at 0). We suggest something else instead:

This is a good place to note a very useful pattern for programming, called *model-view-controller*. The *model* is the way chosen to store the state internally. The *controller* has the logic to modify the model as it needs to evolve. A *view* of a part of the model is something shown to the user that does not need to be in the exact same form as the model itself: A view just needs to be something that can be *easily calculated* from the model, and presumably is desired by the user.

In this case a simple (and already coded!) way to store and control the time model data is the minutes and up to 23 hours that do happen to directly correspond to the 24 hour clock view.

The main control is to advance the time in the `tick` method, and with just two 0-based counts we manage this concisely with the very simple remainder formulas.

So the suggestion is to keep the *internal* attributes the same way as before. Just in the method to create the desire 12-hour *view* have the logic to do the *conversion* of the internal 24-hour model data.

You could leave in the method to provide the time in the 24 hour format, giving the `Clock` class user the option to use either view of the shared model data. To be symmetrical in the naming, you might change the original name `getTimeString` to `getTimeString24` .
