

## Solutions, Week 3, Day 3

1. Take the graph class from the previous lesson, and make it “safe”, with assertions. For example, the `deleteNode` function should complain if it is given a non-existent node.

Using the `deleteNode` function as an example from the adjacency-based graph (with a dictionary where node IDs are the key and the values are lists of the nodes they attach to), it should check if the node exists before executing. `addNode`, `addEdge` etc. can be proofed in similar ways:

```
class Graph:
    [...]

    def deleteNode (self, node):
        """
        Remove a node and any edges that connect to it.

        This has to remove the node and its presence in any
        other node's adjacency list.
        """
        # Preconditions:
        assert (self._nodeDict.has_key(node)), "this node doesn't exist"
        # Main:
        del self._nodeDict[node]
        for k, v in self._nodeDict.iteritems():
            if node in v:
                v.remove (node)
```

2. Write a class `Time`, with a constructor that takes hours, minutes and seconds.

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

    def __add__ (self, other_time):
        theNewTime = Time (
            hours= self.hours+other_time.hours % 24,
            minutes= self.minutes+other_time.minutes % 60,
            seconds= self.seconds+other_time.seconds % 60,
        )
        return theNewTime

    def __str__ (self):
        return "%s:%s:%s" % (self.hours, self.minutes, self.seconds)
```

3. Give the `Time` class methods such that two time objects can be added and subtracted. It should also have a `__str__` method that prints itself out in an `HH:MM:SS` format. Of course, it should be bullet-proofed with assertions.

We create a class that can be initialised with hours, minutes and seconds. We use the magic methods to provide addition and subtraction abilities. It makes sense in arithmetic to not make changes in place, on the original `Time` object, but to return a new `Time` object:

```
class Time:
    def __init__(self, hr=0, mn=0, sc=0)
        # Preconditions:
        assert (0 <= hr) and (hr <= 24), "invalid value for hour"
        assert (0 <= mn) and (mn <= 60), "invalid value for hour"
        assert (0 <= sc) and (sc <= 60), "invalid value for hour"
        # Main:
        self.hr = hr
        self.mn = mn
        self.sc = sc

    def __sub__(self, other):
        return Time (hr=self.hr-other.hr, mn=self.mn-
other.mn, sc=self.sc-other.sc)

    def __add__(self, other):
        return Time (hr=self.hr+other.hr, mn=self.mn+other.mn, sc=self.sc+other.sc)
```

This does not allow for times adding up to roll over to the next time period. For example, a `Time` with 50 minutes and one with 30 minutes could (arguably) add up to 20 minutes and an additional hour. (What happens when the hours add up to 24 or more is undefined.) So we could modify the methods along the lines:

```
def __sub__(self, other):
    theSc = theMn = theHr = 0
    theSc = self.sc-other.sc
    theMn = self.mn-other.mn
    theHr = self.hr-other.hr
    if (60 <= theSc):
        theSc = theSc - 60
        theMn = 1
    if (60 <= theMn):
        theMn = theMn - 60
        theHr = 1
    return Time (hr=theHr, mn=theMn, sc=theSc)
```

4. Create a class with two methods, `f()` and `g()`. In `g()`, throw an exception of a new class that you define. In `f()`, call `g()`, catch its exception and, in the `except` clause, throw a different exception (of a second type that you define).

```
import exceptions

class MyError (exceptions.Exception):
    def __init__(self, msg):
        exceptions.Exception.__init__(self, msg)

class MyDummyClass:
```

```
def f (self):
    try:
        self.g()
    except:
        print "caught the g error"

def g (self):
    raise MyError, "this is the error from g"
```