600.112: Intro Programming for Scientists and Engineers

Assignment 7: Classes*

Peter H. Fröhlich
phf@cs.jhu.edu

Joanne Selinski
joanne@cs.jhu.edu

Due Date: 3pm on Wed 10/28 & Wed 11/4

Introduction

The seventh assignment for *600.112: Introductory Programming for Scientists and Engineers* explores how we can work with complex *objects* by defining our own specialized data types through classes. There are three things to do: First you'll write a Time class module that can be used for any situations where we need to keep track of a specific time of day. Note that there are various built-in Python modules related to time, but for a change we will write our own simplified version instead of learning to use an existing one. Second you'll develop an Experiment class module that can be used to keep track of data related to a particular generic form of scientific experiments. Lastly, you'll create an EResults class module which will hold and provide operations to manipulate a collection of Experiment objects. You will also have to complete a transaction processing main program that uses your EResults class to get and manipulate experimental data.

You must submit a complete zip file with all *.py and *.txt files needed to run your solutions as detailed below, and on Blackboard before the deadline. As usual, don't forget to include descriptive docstrings for all your modules and the functions in them, and make your Python files pep8 compliant. You are encouraged (but not strictly required) to add doctests to your class methods as we did in lecture, in addition to testing them with the provided test programs.

1 Time Class [10 points]

We'll begin by creating a Time class which will be used to store and manipulate times of day. Name this class `Time` and define it in file `Time.py` and nothing else. We'll include some functions to set and change Time object attributes with data validation, and to access a Time in varous formats. We want our class to be flexible enough to handle times that are represented in standard format, such as "1:30pm" as well as military format, such as "1530".

If you're not familiar with military format, it is a way of using 4-digit integers to represent time, where the first two digits are the hours, and the last two are the minutes. Midnight is 0, noon is 1200, and the pm hours are represented with the values 13 (1pm) to 23 (11pm). One Python limitation in working with military times is that we cannot use integer constants that begin with 0, such as 0500 which is military speak for 5am. Python interprets integer constants that start with a zero as octal literals - integers in base 8 instead of base 10.

*Disclaimer: This is *not* a course in physics or biology or epidemiology or even mathematics. Our exposition of the science behind the projects cuts corners whenever we can do so without lying outright. We are *not* trying to teach you anything but computer science!

The specifications of the Time class are illustrated with a TimeTest.py program we have written for you. We are going to develop the Time class incrementally, according to the methods that are called in the test program. It is fairly common to develop object oriented solutions this way, essentially writing a module to conform to a specification contract inherent in a test program. In order to get a sense of what is supposed to happen when the TimeTest program is run, we have created a sample run that demonstrates the correct results.

Here are the methods that your Time class must provide, based on the usage in the TimeTest program:

- __init__(self,tm) - a constructor that has a parameter which is either an integer for a military time, or a string in standard time format. Do data validation and initialize the Time object to midnight if either the hours or minutes are invalid.
- __str__(self) - return a string version of the time in standard format.
- milInt(self) - return an integer which is the military format value.
- milStr(self) - return a string which has exactly 4 digits (including leading zeros) representing the military format value.
- elapsed(self,other) - calculate and return the elapsed minutes between self and some other time.

However, in crafting these methods we will create several helper methods to simplify the coding and provide good building blocks with data validation.

The proper way to incrementally develop this class is to begin the Time class definition with the class declaration, and a skeleton with the required method headers. For methods that are supposed to return a particular type of value, we will include a dummy return value of the correct type. For others, we will simply `pass` on the method definitions. It's important to try running the TimeTest program once you have created this skeleton to be sure the files will work together correctly without syntax problems.

```
class Time:
    def __init__(self, tm):
        pass

    def __str__(self):
        return ''standard string''

    def milInt(self):
        return -1

    def milStr(self):
        return ''military string''

    def elapsed(self, other):
        return -1
```

Next we must decide how to represent the attributes of a Time object with instance data members. There are several approaches one could take. The fact that the programmers using our Time class do not need to know these gory details of the internal representation is a great feature of object oriented programming called *information hiding*. One option would be to store the Time as a single string in standard format. The opposite approach would be to store it as an integer in military format. Since we have to work with our Time objects in standard and military forms, we need a set of data members that will let us transition fairly easily. What we absolutely do not want to do (trust me on this) is store the data for an object in both forms! That will create twice the work for every update that occurs.

So consider instead a hybrid approach where we store the hours and minutes separately as integers, using military values for the hours which also allow us to keep track of whether our object is am or pm. Thus we will have an instance variable `self.hour` that can have valid values from 0 to 23 inclusive,

and another one `self.mins` that can range from 0 to 59 inclusive. (Be careful not to call a variable `min` in this project because that is the name of the built-in minimum function in Python.)

The first method to implement should be our constructor. Since the parameter could be an integer for military time or a string for standard time, we will have to test it and then act accordingly. Rather than put all the code to convert from these forms into our instance data members, we will create helper methods `setMil` and `setStandard`. We also need to consider how to approach the data validation aspect, so that we don't allow the hours or minutes to be outside the acceptable ranges. Within the constructor it makes sense to initialize our data members each to 0 before calling our set helper methods. That way, if the passed parameters are invalid, the time object will be created with a default hour of midnight and/or zero minutes. Here is the resulting constructor definition:

```
    def __init__(self, tm):
        self.hour = 0
        self.mins = 0
        if type(tm) is int:
            self.setMil(tm)
        else: # standard format string
            self.setStandard(tm)
```

Now for the implementation of `setMil` and `setStandard`. Since they each need to validate and set the hour and minutes appropriately, once again we will make helper methods to do these jobs independently: `setHour` and `setMins`. We can convert a military integer time value into hours and minutes quite easily using simple integer division and remainder operations:

```
    def setMil(self, tmil):
        self.setHour(tmil / 100)
        self.setMins(tmil % 100)
```

We have to do a little more work in splitting a standard time string into parts and converting them to the appropriate integer values. It's important to remember to adjust the hour from standard time to military time based on whether it is am or pm. Once we get the correct values, we can again use our set methods for the data validation.

```
    def setStandard(self, tstr):
        tstr = tstr.lower()
        colon = tstr.find(':')
        hr = tstr[:colon]
        mn = tstr[colon+1:colon+3]
        hr = int(hr)
        mn = int(mn)
        if 'a' in tstr:
            if hr == 12:
                hr = 0
        else:  # pm
            if hr < 12:
                hr += 12
        self.setHour(hr)
        self.setMins(mn)
```

So now we must write those two setters. The data validation is very straightforward, and we will only set our instance variables with good data. We will print an error message when invalid data is encountered so the users of this class will know that something has gone wrong. Here are the definitions:

```
    def setHour(self, hr):
        if hr >= 0 and hr < 24:
            self.hour = hr
        else:
            print 'ERROR:', hr, 'is invalid hour'

    def setMins(self, mn):
```

```
        if 0 <= mn < 60:  # very cool Python-ism
            self.mins = mn
        else:
            print 'ERROR:', mn, 'is invalid mins'
```

So far we have written four helper methods just to implement the constructor. That might seem ridiculous, but each of those methods has a very specific purpose. In fact, now that we have written them, each of those methods could also be called independently of the constructor to change one or both data members of a Time object. Here's what that might look like; each of these statements changes the Time object `t`:

```
t = Time(``12:30pm'')
t.setMil(430) # 4:30 in the morning
t.setStandard(``11:05am'')
t.setHour(6)  # now 6:05 in the morning
t.setMins(90) # generates error message, no change
```

Now it is rather difficult to know if any of those methods are working correctly without being able to get some useful data out of our Time object. If we try to print one (or run the TimeTest program as you should be doing), we will only get our dummy return values. Let's tackle the military getters first since they will be easier to write than the standard string method. For the `milInt` method, we simply need to do the inverse arithmetic operations to get back to a single integer. The `milStr` method can then call on this one, making sure to format it so that it prints with 4 full digits, including leading zeros. Each method is a simple one-liner:

```
    def milInt(self):
        return self.hour*100 + self.mins

    def milStr(self):
        return '%04d' % self.milInt()
```

Notice the use of '0' in the format string - this is our way of insisting that the integer is printed with 4 digits including leading zeros. Pretty cool, huh?

Next we should tackle the standard string method. This one has lots of little parts related to converting back from military hours to standard hours with an explicit am or pm designator. We need to determine if the object is am or pm. We need to adjust a midnight hour from 0 to 12, and we need to adjust the pm hours greater than 12. All this must be done without disturbing the stored instance variable `self.hour`. Lastly, we'll use a similar format string to make sure that the minutes print with exactly two digits. Here's the full kit and caboodle:

```
    def __str__(self):
        if self.hour >= 12:
            when = 'pm'
        else:
            when = 'am'
        if self.hour > 12:
            hstr = str(self.hour-12)
        elif self.hour == 0:
            hstr = '12'
        else:
            hstr = str(self.hour)
        return hstr + ':' + '%02d' % self.mins + when
```

None of the methods we've written for the Time class so far are particularly complex algorithmically. But they are all very important because they set the stage for working with these objects in various convenient forms. Our last task is a little more challenging - how can we calculate the number of minutes that have elapsed between a start and end time? When the start time is before the end time, both are assumed to be on the same day, and it's not too difficult to do. However, we also must account for

the possibility that the start time is on one day, and the end time is on the next day, for example if we are trying to determine how long we were at a party last weekend.

Your first instinct for coding this method might be to list all the possible conditions you need to consider, particularly since you are working with hours and minutes for two different times. It might seem like a bunch of arithmetic operations in decision statements is the way to go. But there's an easier way - let's first write a method (yes, one more helper) to calculate the minutes since midnight for any given time. This will allow us to then easily subtract the times with fewer moving parts, giving us the elapsed time in minutes as desired.

```
    def minsSinceMidnight(self):
        return self.hour*60 + self.mins
```

Using this in the elapsed method, we now only have to check whether the start time is before the end time or not. If not, then the total time is the number of minutes from start to the end of the day, plus the minutes since midnight for the end time.

```
    def elapsed(self, other):
        first = self.minsSinceMidnight()
        scnd = other.minsSinceMidnight()
        if first <= scnd:
            return scnd - first
        else:
            return 24*60 - first + scnd
```

And there you have it - our first class definition to create a truly customized data type. The only work that remains for you is to **add detailed docstrings** to every method in the class! **IMPORTANT GRADING NOTE: If any Python syntax errors are produced when we try to run the TimeTest program with your Time class, you will get a 0 for this part of the assignment.** If it runs, but produces incorrect results you will get partial credit as usual.

2 Timed Experiments [15 points]

In this part of the assignment we will continue developing classes to define new data types. The eventual goal is to provide a comprehensive set of classes and operations to enable a user to build and manipulate a collection of results from scientific experiments. We have already defined a Time class that will be used for this part of the project as well. The purpose of the next class definition is to hold data for a scientific experiment. Each experiment will be a timed transformation from one solution to another by introducing some type of agent. Probably you have encountered or can imagine many different situations which would produce results in this generic form. Name this second class `Experiment` and define it in file `Experiment.py` and nothing else.

Let's consider the requirements of the Experiment class. The data for one Experiment will consist of a start time, an end time, the starting solution, the final solution, and the agent. The solutions and agents will be strings. The start and end times could given in either standard or military format. Here are two examples of data that might be used to initialize Experiment objects:

```
9:30am 10:15am SolutionA SolutionB Agent1
1440 2204 ice water heat
```

The specifications of the Experiment class are illustrated with a ExperimentTest.py program we have written for you. The expected output from the program appears as a long comment at the end of the file. These are the methods that your Experiment class must provide, based on the usage in the test program:

- __init__ - a constructor that has parameters for all five data members: start time, end time, starting solution, final solution, and agent. Let the Time class handle the data validation for start and end.

- __str__ - return a string version of the Experiment with the times appearing in 4 digit military format and including the elapsed minutes from start to end time, using this exact format:

```
[startTime, endTime] startSol X agent => finalSol (elapsedMins)
```

- __cmp__(self, other) - compare two experiments based on elapsed time, using start time as a tie breaker; return a negative integer if self $<$ other, 0 if the same and a positive integer if self $>$ other.
- minutes - get the number of elapsed minutes from the start time to the end time for this experiment.
- usedAgent - pass a string parameter which is an agent, and return True if the experiment used that Agent, False otherwise.
- hasSolution - pass a string parameter which is a solution and return True if the experiment has the parameter as either the starting or final solution (or both), False otherwise.

In crafting these methods you are welcome to create helper methods to simplify the coding and provide good building blocks with data validation, just as we did for the Time class above. Also, remember to start with a class skeleton that includes dummy return values before you try to implement any of the methods, to be sure your Experiment class is set up correctly to run with the ExperimentTest program. **If any Python errors are produced when we try to run the ExperimentTest program with your Experiment class, you will get a 0 for this part of the assignment.** If the program runs but produces incorrect results, you will get partial credit as usual.

3 Experiments Collection [25 points]

The third class you must write will be used to build and manipulate a collection of experimental results in ways that would not be possible if you simply used a sequence of Experiment objects. Name this class `EResults` and define it in file `EResults.py` and nothing else.

Your EResults class should store a sequence of Experiment objects, and facilitate various operations on that collection. Unlike for the Time and Experiment class, we are not giving you a strict specification through a test program. Instead, look ahead to what the main program for Part 3 is supposed to do, and create methods in the EResults class to support that functionality. As a guideline, you should have a method in EResults corresponding to each different transaction type. Most of the work for processing the transactions below should be done in this EResults class, with the main program primarily just handling input and output.

4 Processing Experimental Results [5 pts]

For the third part of this assignment, you will finish writing a program that uses the classes we've developed above to read and process experimental data that is stored in plain text files. Call your solution to this part `eQueries.py` and nothing else. We have created a starter file for you that does most of the work already. You literally should only need to add about 5 lines of code to this file, and comment out some print statements once you get corresponding operations working. Most of the hard work this program does for the user actually happens in the EResults class itself (which you are writing).

There are two types of input for the program. First we have plain text data files containing the results of many scientific experiments. The other type of input is a plain test transaction file containing a sequence of queries to find out information about the results, such as what was the average time to get from solutionA to solutionB, or to list all the experiments that used a particular agent. Reading and processing this transaction file will guide the main activities when you run the program. Sample input

files are posted on the main assignment webpage (exp1.txt and trans.txt). We have given you one file with experiment data (exp1.txt), but you will need to create a second one named exp2.txt to work with the given transaction file.

Each data input file can contain the results of any number of experiments. The data for each experiment will be all on one line of a file. Specifically, the data will consist of a start time, an end time, the starting solution, the final solution, and the agent. The solutions and agents will be strings; you can assume they do not contain spaces. The start and end times could be in either standard or military format, but must not contain spaces either.

At the start of this program, the user is prompted for the name of the transaction input file. The main program then processes the entire transaction file, one line at a time. For each input line, it writes the transaction request itself to the screen, followed by any results it produces.

The transaction input file contains a sequence of operation requests, one per line. Each line starts with a single letter code from the list below, followed by any data that the transaction needs to do its job. In other words, there is no additional user input to the program once the transaction file name has been given.

Here are the transaction codes and descriptions (useful in writing the EResults class so that you know what is needed):

- 'R' - Read a new experiment data input file, adding each experiment in the file to the current results collection. Additional data for this transaction is the name of this plain text file (eg, "R data1.txt"). Output the number of experiments (lines) that were in this file.
- 'N' - Output the total number of experiments in the collection.
- 'L' - List all the experiments, one per line, in any order. For each experiment use the format specified in the Experiment class above.
- 'S' - Sort the experiments by elapsed time (shortest to longest), breaking ties by start time, then list all in sorted order.
- 'A' - Display a list of the experiments that used a particular agent. Additional data for this transaction includes the agent; assume capitalization is not relevant (eg: "A heat").
- 'T' - Compute and output the average elapsed time (in minutes) for transforming a particular solution to another. Additional input is the two solutions' names; assume capitalization is not relevant (eg, "T startSol finalSol"). Display a -1 if no experiments do that particular transformation.
- 'D' - Delete all the experiments that involve a particular solution as either the starting point or resulting product. Additional input is the name of the solution, case insensitive (eg "D SolutionB"). Output each experiment that got deleted.
- 'Q' - Quit the program.

For any operations that you do not successfully implement (in your EResults class), write "transaction type not supported" to the screen each time they should be executed. It is likely that you will need to add a few methods to your Experiment and/or EResults classes in order to fully support the operation of this program.

Make sure you include all the *.py and *.txt files needed to run your solution in your zip file submission. Also remember to include description docstrings for each method and make all your code pep8 compatible. You are not strictly required to write doctests for this assignment (but always encouraged to do so).