When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download the **q3helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **ttime.py** and **trackhistory.py** modules online by Thursday, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on Friday morning.

---

1. (20 pts) Complete the class **Time**, which stores and manipulates times on a 24-hour clock. As specified below, write the required methods, including those needed for overloading operators. Exceptions messages should include the class and method names, and identify the error (including the value of all relevant arguments). Hint see the **type_as_str** function in the **goody**.py module. You can't use any of Python's other time-related modules.

1. The class is initialized with three **int** values (the **hour** first, the **minute** second, the **second** third; all with default values of **0**). If any parameter is not an **int**, or not in the correct range (the **hour** must be between **0** and **23** inclusive, the **minute** and **second** must be between **0** and **59** inclusive) raise an **AssertionError** with an appropriate string describing the problem/values. When initialized, the **Time** class should create exactly three attributes/**self** variables named **hour**, **minute**, and **second** (with these exact names and no others attributes/**self** variables).

2. Write the **__getitem__** method to allow **Time** class objects to be indexed by either (a) an **int** with value **1** or **2** or **3**, or (b) any length tuple containing any combinations of just these three values: e.g., **(1,3)**. If the index is not one of these types or values, raise an **IndexError** exception with an appropriate string describing the problem/values. If the argument is **1**, returns the **hour**; if the argument is **2**, return the **minute**, and if the argument is **3**, return the **second**. If the argument is a **tuple**, return a **tuple** with **hour** or **minute** or **second** substituted for each value in the **tuple**. So if **t = Time(5,15,20)** then **t[1]** returns **5** and **t[2,3]** returns **(15,20)**. Note that calling **t[1]** will pass **1** as its argument; calling **t[1,2]** will pass the **tuple (1,2)** as its argument. In fact, **t[1,1]** will pass the **tuple (1,1)** and return the **tuple (5,5)**.

3. Write methods that return (a) the standard **repr** function of a **Time**, and (b) a **str** function of a **Time**: **str** for a Time shows the time in the standard 12 hour clock format: **str(Time(13,10,5))** returns **'1:10:05pm'**; **str(Time(5,6,3))** returns **'5:06:03am'**; **str(Time(0,0,0))** returns **'12:00:00am'**. It is critical to write the **str** method correctly, because I used it in the batch self-check file for testing the correctness of other methods.

4. Write a method that interprets midnight as **False** and any other time as **True**.

5. Write a method that interprets the length of a **Time** as the number of seconds that have elapsed from midnight to the that time. So **len(Time(0,0,0))** returns **0** and **len(Time(23,59,59))** returns **86,399**; there are **86,400** seconds in a day: midnight to midnight.

6. Overload the **==** operator to allow comparing two **Time** objects for equality (if a time object is compared against an object from any other class, it should return **False**). Note that if you define **==** correctly, Python will be able to compute **!=** by using **not** and **==**.

7. Overload the **<** operator to allow comparing two **Time** objects. The left **Time** is less-than the right one if it comes earlier in the day than the right one. Also allow the right operand to be an **int**: in this case, return whether the length (an **int**, see above) of the **Time** is less-than the right **int**. If the right operand is any other type, return **NotImplemented**. Note that if you define **<** correctly, Python will be able to compute **Time > Time** and **int > Time** by using **<**.

8. Overload the **+** operator to allow adding a **Time** object and an **int**, producing a new **Time** object as a result (and not mutating the **Time** object **+** was called on). If the other operand is not an **int**, return **NotImplemented** (you may assume without checking that this argument is non-negative). Both **Time + int** and **int + Time** should be allowed and have the same meaning. Hint: write code that adds one second to a **Time**; then iterate over this code the **int** number of times: there are faster ways to do this, but this way is correct.

9. Write the **__call__** method to allow an object from this class to be callable with three **int** arguments: update the **hour** of the object to be the first argument, and the **minute** of the object to be the second argument, and the **second** of the object to be the third argument. Return **None**. If any parameter is not legal (see how the class is initialized), raise an **AssertionError** with an appropriate string describing the problem/values.

The **q3helper** project folder contains a **bscq31S21.txt** file (examine it) to use for batch-self-checking your class, via the **driver.py** script. These are rigorous but not exhaustive tests. Incrementally write and test your class.

2. (5 pts) Complete the class **TrackHistory**, which should (a) use a dictionary (hint: I used **defaultdict**) to remember a complete history of all the values binding to all the attribute names in the **TrackHistory** class; (b) be able to retrieve the current and previous values of an attribute, by appending a **_prev** and (optionally a number) suffix to the attribute's name; for example **_prev** and **_prev1** goes back to the previous value, **_prev3** goes back three previous values; **_prev0** is another way to get the current value; (c) be able to retrieve a dictionary of the current and historical binding using indexing (**0** means current values, **-1** means previous values, **-2** means previous to previous values, etc.). So **o.x_prev3** returns the same value as **o[-3]['x']**.

The class is initialized with no arguments (it creates a dictionary with no items).

1. Write the **__setattr__** method to (a) disallow any new attributes containing the string **'_prev'**: raise a **NameError** with an appropriate string describing the problem/values; (b) update both the dictionary of historical values and current values (in **self.__dict__**) with the new value for an attribute name (except for the name you use to store the historical dictionary itself). See the lecture notes. Hint: my method body has about a half-dozen lines.
2. Write the **__getattr__** method to allow the names of actual attributes to be followed by **_prev** or **_prevN** where **N** is one or more digits specifying a non-negative number (e.g., **_prev5**, **_prev0**, **_prev2355**, **_prev001**); hint: use a regular expression. If the name before the first **_prev** suffix is not an attribute of the class, or the **_prev** suffix occurs anywhere but the very end of the name, raise a **NameError** exception with an appropriate string describing the problem/values. If the **_prev** is followed by a number too big (the attribute doesn't have that many previous values) this method returns the value **None**. Hint: my method body has about a half-dozen lines with a few conditional statements/expressions.
3. Write the **__getitem__** method to allow the class objects to be indexed by a non-positive **int**: **0** as an index returns a dictionary of the current values; **-1** as an index returns a dictionary of the previous values. **-2** as an index returns a dictionary of the values previous to the previous values; etc. If the level of previous values doesn't exist for a name according to the supplied index, the dictionary should show the value **None** associated with that name. If the index is positive, raise an **IndexError** exception with an appropriate string describing the problem/values. Hint: use a comprehension to compute the returned dictionary; my method body has about a half-dozen lines.

For example, assume that we first declare **x = TrackHistory ()** and then execute **x.a = 1** then **x.a = 2** then **x.b = 1**. Then **x.a** and **x.a_prev0** evaluates to **2**; **x.a_prev** evaluates to **1**; and **x.a_prev2** evaluates to **None**. Also, **x[0]** evaluates to **{'a': 2, 'b': 1}**; **x[-1]** evaluates to **{'a': 1, 'b': None}**; and **x[-2]** evaluates to **{'a': None, 'b': None}**. Generally **x[-n]['name']** is the same as **x.name_prevN**.

The **q3helper** project folder contains a **bscq32S21.txt** file (examine it) to use for batch-self-checking your class, via the **driver.py** script. These are rigorous but not exhaustive tests.