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 **ddate.py** and **misspelling.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 **Date**, which stores and manipulates dates. 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 may not import/use the **datetime** or other similar modules in Python.

   1. The class is initialized with three **int** values (the **year** first, the **month** second, the **day** third.) If any parameter is not an **int**, or not in the correct range (the **year** must be **0** or greater, the **month** must be between **1** and **12** inclusive, and the **day** must be legal given the **month** and **year**: remember about leap years) raise an **AssertionError** with an appropriate string describing the problem/values. When initialized, the **Date** class should create exactly three **self** variables named **year**, **month**, and **day** (with these exact names and no others **self** variables).

   2. Write the **__getitem__** method to allow **Date** class objects to be indexed by either (a) an **str** with value **'y'** or **'m'** or **'d'**, or (b) any length tuple containing any combinations of just these three values: e.g., **('m','d')**. 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 **'y'**, returns the **year**; if the argument is **'m'**, return the **month**, and if the argument is **'d'**, return the **day**. If the argument is a **tuple**, return a **tuple** with **year** or **month** or **day** substituted for each value in the **tuple**. So if **d = day(2016,4,15)** then **d['y']** returns **2016** and **d['m','d']** returns **(4,15)**. Note that calling **d['m']** will pass **'m'** as its argument; calling **d['m','d']** will pass the **tuple ('m','d')** as its argument.

   3. Write methods that return (a) the standard **repr** function of a **Date**, and (b) a **str** function of a **Date**: **str** for a **Date** shows the date in the standard format: **str(Date(2016,4,15))** returns **'4/15/2016'**; 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 the length of a **Date** as the number of days that have elapsed from January 1st in the year 0 to that date. So **len(Date(0,1,1))** returns **0**; **len(Date(0,12,31))** returns **365**; **len(Date(2016,4,15))** returns **736,434**. Hint: **len(Date(0,3,14))** returns **73**: **31** days in January, **29** days in February (it is a leap month), and **13** days in March, which sum to **73**.

   5. Overload the **==** operator to allow comparing two **Date** objects for equality (if a **Date** 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 **==**.

   6. Overload the **<** operator to allow comparing two **Date** objects. The left **Date** is less-than the right one if it comes earlier 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 **Date** is less-than the right **int**. If the right operand is any other type, Python should raise a **TypeError** exception (hint: **NotImplemented**) with the standard error message. Note that if you define **<** correctly, Python will be able to compute **Date > Date** and **int > Date** by using **<**.

   7. Overload the **+** operator to allow adding a **Date** object and an **int** (which can be positive or negative) producing a new **Date** object as a result (and not mutating the **Date** object **+** was called on). The result is a new **Date** that many days in the future for a positive **int**; in the past for a negative **int**. For example, **Date(2016,4,15)+100** returns **Date(2016,7,24)**, **100** days in the future of **4/15/2016**. If the other operand is not an **int**, Python should raise a **TypeError** exception (hint: **NotImplemented**) with the

standard error message. Both **Date + int** and **int + Date** should be allowed and have the same meaning. Hint: write code that repeatedly adds/subtracts one day at a time to get to get to the required **Date**.

8. Overload the - operator to allow subtracting two **Date** objects, producing an **int** object as a result (and not mutating the **Date** objects - was called on). The difference is the number of days from the left **Date** to the right **Date** (which can be negative). For example, **Date(2016,6,8)-Date(2016,4,15)** returns **54** and **Date(2016,4,15)-Date(2016,6,8)** returns **-54**. Hint: use the **len** function defined above. Also, allow subtracting an **int** from a **Date**. It should produce the same value as adding its negative value to a **Date**: for example, **Date(2016,4,15)-1** should produce the same result as **Date(2016,4,15)+-1**.

9. Write the **__call__** method to allow an object from this class to be callable with three **int** arguments: update the **year** of the object to be the first argument, and the **month** of the object to be the second argument, and the **day** 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 **bscq31W20.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 **Misspellings**, which should be able to correct the misspelling of attribute names: both for getting the value of an attribute and setting the value of an attribute. See the exact specifications below.

The **distance_helper** module defines a **Memoize** class and a **min_dist** function. Leave these definitions exactly as they are. Do not worry what the **Memoize** class does nor how it works: we will study it in a few weeks. Do not worry how the **min_dist** function works, but understand that it recursively computes the **minimum distance** between two strings. The more dissimilar the strings, the bigger the value it returns.

The distance between strings increases by **1** for every addition, deletion, or substitution it takes to convert one string into the other. For example **min_dist ("able", "camel")** is **4**: to transform **"able"** into **"camel"** we can add a **c** at the front, substitute an **m** for a **b**, add an **e** between the **m** and **l**, and delete the **e** at the end Going the other way, we can delete the **c** at the front, substitute a **b** for an **m**, add an **l** between the **m** and **e**, and delete the **l** at the end. There are other ways to convert in **4** changes, but no ways to do it in **3**: so **4** is the minimum distance.

The **__init__** method in **Misspelling** has a parameter (whose default value is **False**) that is used to set the **fix_when_setting** attribute (see below for how this attribute is used). It calls **initialize_attributes** to initialize the attribute names that may be misspelled in tests. For testing purposes, this is easier to do outside of **__init__**. You can change the inside of either method, but **__init__** must call **initialize_attributes** as its last statement.

1. Write a method named **closest_matches**; its parameters are **self** (an object from the class **Misspelling**) and **name** (a **str**); it returns a **list** of all the attribute names in that **Misspelling** object that have the same minimum distance from **name**, so long at that minimum distance is ≤ half of the length of **name** (so, not too distant from **name**). The returned **list** may be empty (no attributes names are close), or contain one or more values (multiple attribute names all having the same minimum distance).
2. Write the **__getattr__** method; its parameters are **self** (an object from the class **Misspelling**) and **name** (a **str** that is not the correct spelling of any attribute in the object); it returns the value of the attribute name in **self** that is closest to **name** (use the **list** of attribute names returned by **closest_matches**). Do this only if there is exactly **one** attribute name that is the minimum distance from **name**: if there are none or more than one, this method should raise a **NameError** exception with an appropriate message. Remember that **__getattr__** is called by Python only if **name** is not an actual attribute in an object.
3. Write the **__setattr__** method; its parameters are **self** (an object from the class **Misspelling**), **name** (a **str** that may or may not be the name of an attribute), and **value** (the new value to be bound to the

attribute **name**). This method should allow (a) the binding of any attribute names in **__init__** and **initialize_attributes**; (b) the rebinding of any attribute name that is already stored in the object. Otherwise (c) if **name** is not an attribute name and the **fix_when_setting** attribute is **True**, then value should be bound to the unique name that most closely matches **name** (use the **list** of attribute names returned by **closest_matches**). if none or more than one attribute names match with the minimum distance (or **fix_when_setting** is **False**) then this method should raise a **NameError** exception with an appropriate message. Hint: Use code like the **History** class example in the notes to ensure all the attribute names in **__init__** and **initialize_attributes** get bound correctly. Read the comment before **initialize_attributes**, which guarantees the last attribute name bound in that method is **last**.

Here is an example of a run of the code in **misspelling.py** (**allowing** fixing spelling in **__setattr__**).

```
Allow fixing mispelling in __setattr__[True]:

{'fix_when_setting': False, 'amoral': 1, 'more': 2, 'babel': 3, 'last': 4}
Enter test: o.babel
3

{'fix_when_setting': False, 'amoral': 1, 'more': 2, 'babel': 3, 'last': 4}
Enter test: o.babe
3

{'fix_when_setting': False, 'amoral': 1, 'more': 2, 'babel': 3, 'last': 4}
Enter test: o.least
4

{'fix_when_setting': False, 'amoral': 1, 'more': 2, 'babel': 3, 'last': 4}
Enter test: o.colorado
('Misspelling.__getattr__: name(colorado) not found; matches =', [])

{'fix_when_setting': False, 'amoral': 1, 'more': 2, 'babel': 3, 'last': 4}
Enter test: o.amorx
('Misspelling.__getattr__: name(amorx) not found; matches =', ['amoral', 'more'])

{'fix_when_setting': True, 'amoral': 1, 'more': 2, 'babel': 3, 'last': 4}
Enter test: o.babel = 5

{'fix_when_setting': True, 'amoral': 1, 'more': 2, 'babel': 5, 'last': 4}
Enter test: o.babe = 6

{'fix_when_setting': True, 'amoral': 1, 'more': 2, 'babel': 6, 'last': 4}
Enter test: o.mora = 7

{'fix_when_setting': True, 'amoral': 1, 'more': 7, 'babel': 6, 'last': 4}
Enter test: o.least = 8

{'fix_when_setting': True, 'amoral': 1, 'more': 7, 'babel': 6, 'last': 8}
Enter test: o.amorx = 9
('Misspelling.__setattr__: name(amorx) not found; matches =', ['amoral', 'more'])

{'fix_when_setting': True, 'amoral': 1, 'more': 7, 'babel': 6, 'last': 8}
Enter test: o.amoralandmuchmuchmore = 10
('Misspelling.__setattr__: name(amoralandmuchmuchmore) not found; matches =', [])
```

Here is an example of a run of the code in **`misspellings.py`** (**not allowing** fixing spelling in **`__setattr__`**).

```
Allow fixing mispelling in __setattr__[True]: False

{'fix_when_setting': False, 'amoral': 1, 'more': 2, 'babel': 3, 'last': 4}
Enter test: o.least
4

{'fix_when_setting': False, 'amoral': 1, 'more': 2, 'babel': 3, 'last': 4}
Enter test: o.least = 5
Misspelling.__setattr__: name(least) not found and spelling correction disabled
```