Program 2

Classes and Overloaded Operators (and one Iterator)

ICS-33: Intermediate Programming

Introduction

This programming assignment is designed first to ensure that you know how to write classes that overload many of the standard Python operators by defining various double-underscore methods. It also ensures that you know how to write classes that implement iterators, by defining an **__iter__** method that returns an object that we/Python can call **__next__** on. **These Iterators are covered near the end of the due date for this project; skip writing these functions (only in the first class) until the material is covered in class, or read ahead.**

You should download the program2 project folder and unzip it to produce an Eclipse project with two modules. You will write classes in these modules, which can be tested in the script and using the standard driver using the batch self-check files that I supplied. Eventually you will submit each of these modules you write separately to Checkmate.

I recommend that you work on this assignment in pairs, and I recommend that you work with someone in your lab section (so that you have 4 hours each week of scheduled time together). These are just recommendations. Try to find someone who lives near you, with similar programming skills, and work habits/schedule: e.g., talk about whether you prefer to work mornings, nights, or weekends; what kind of commitment you will make to submit program early.

Only one student should submit all parts of the the assignment, but both students' UCInetID and name should appear in a comment at the **top** of **each submitted .py file**. A special grading program reads this information. The format is a comment starting with **Submitter** and **Partner** (when working with a partner), followed by a **colon**, followed by the student's **UCInetID** (in all lower-case), followed by the student's **name in parentheses** (last name, comma, first name -capitalized appropriately). If you omit this information, or do not follow this exact form, it will require extra work for us to grade your program, so we will deduct points. Note: if you are submitting by yourself, and do **NOT** have a partner, you should **OMIT** the partner line and the "...certify" sentence.

For example if Romeo Montague (whose UCInetID is romeo1) submitted a program that he worked on with his partner Juliet Capulet (whose UCInetID is jcapulet) the comment at the top of each .py file would appear as:

```
# Submitter: romeo1(Montague, Romeo)
# Partner  : jcapulet(Capulet, Juliet)
# We certify that we worked cooperatively on this programming
#   assignment, according to the rules for pair programming
```

If you do not know what the terms **cooperatively** and/or **rules for pair programming** mean, please read about Pair Programming **before** starting this assignment. Please turn in each program **as you finish it**, so that I can more accurately assess the progress of the class as a whole during this assignment.

Print this document and carefully read it, marking any parts that contain important detailed information that you find (for review before you turn in the files). The code you write should be as compact and elegant as possible, using appropriate Python idioms.

Problem #1: Bag Class

Problem Summary:

Write a class that represents and defines methods, operators, and an iterator for the **Bag** class. Bags are similar to sets, and have similar operations (of which we will implement just the most important) but unlike sets they can store multiple copies of items. We will store the information in bags as dictionaries (I suggest using a **defaultdict**) whose keys are associated with **int** values that specify the number of times the key occurs in the **Bag**. **You must store Bags using this one data structure, as specified**

Details

1. Define a class named **Bag** in a module named **bag.py**
2. Define an **__init__** method that has one parameter, an iterable of values that initialize the bag. Writing **Bag()** constructs an empty bag. Writing **Bag(['d','a','b','d','c','b','d'])** construct a bag with one **'a'**, two **'b'**s, one **'c'**, and three **'d'**s. Objects in the **Bag** class should store only the dictionary specified above: it should **not** store/manipulate any other **self** variables.
3. Define a **__repr__** method that returns a string, which when passed to **eval** returns a newly constructed bag with the same value (**==**) to the object **__repr__** was called on. For example, for the **Bag** in the discussion of **__init__** the **__repr__** method would print its result as **Bag(['a', 'c', 'b', 'b', 'd', 'd', 'd'])**. Bags like sets are not sorted, so these 7 values can appear in any order. We might require that information in the list is sorted, but not all values we might put in a bag may be ordered (and therefore not sortable): e.g., a bag storing both string and int values, **Bag(['a',1])** which is allowed.

   Note: This method is used to test several other methods/operators in the batch self-check file; so it is critical to write it correctly.
4. Define a **__str__** method that returns a string that more compactly shows a bag. For example, for the **Bag** in the discussion of **__init__** the **__str__** method would print its result as **Bag(a[1], c[1], b[2], d[3])**. Bags like sets are not sorted, so these 7 values can appear in any order.
5. Define a **__len__** method that returns the total number of values in the **Bag**. For example, for the **Bag** in the discussion of **__init__** the **__len__** method would return **7**.
6. Define a **unique** method that returns the number of different (**unique**) values in the **Bag**. For example, for the **Bag** in the discussion of **__init__** the **unique** method would return **4**, because there are four different values in the **Bag**; contrast this method with **__len__**.
7. Define a **__contains__** method that returns whether or not its argument is in the **Bag** (one or more times).
8. Define a **count** method that returns the number of times its argument is in the **Bag**: **0** if the argument is not in the **Bag**.
9. Define an **add** method that adds its argument to the **Bag**: if that value is already in the **Bag**, its count is incremented by **1**; if it is not already in the **Bag**, it is added to the **Bag** with a count of **1**.
10. Define an **__add__** method that unions its two **Bag** operands: it returns a new **Bag** with all the values in **Bag** operands. For example: **str(Bag(['a','b']) + Bag(['b','c']))** should be **'Bag(a[1],b[2],c[1])'** Neither **Bag** operand should change.
11. Define a **remove** method that removes its argument from the **Bag**: if that value is already in the **Bag**, its count is decremented by **1** (and if the count reduces to **0**, the value is removed from the dictionary; if it is not in the **Bag**, raise a **ValueError** exception, with an appropriate message that includes the value that could not be removed.
12. Define **__eq__**/**__ne__** methods that return whether one **Bag** is equal/not equal to another: contains the same values the same number of times. A **Bag** is not equal to anything whose type is not a **Bag** This method should not change either **Bag**.
13. Define an **__iter__** method that that returns an object on which **next** can be called to produce every value in the **Bag**: all **len** of them. For example, for the **Bag** in the discussion of **__init__**, the following code

    ```
    for i in x:
        print(i,end='')
    ```

    would print

    ```
    acbbddd
    ```

    Bags like sets are not sorted, so these 7 values can appear in any order.

    Ensure that the iterator produces those values in the **Bag** at the time the iterator starts executing; so mutating the **Bag** afterwards, or even during the iteration, will **not** affect what values it produces.

    Hint: Write this method as a call to a local generator, passing a copy of the dictionary (covered in Friday's lecture in Week 4).

I have shown only examples of **Bag**s storing strings, because they are convenient to write. But bags can store any type of data. The **__repr__**, **__str__**, and **__iter__**/**__next__** methods must be written independently: neither should call the other to get things done.

Testing

The **bag.py** module includes a script that calls **driver.driver()**. The project folder contains a **bsc1.txt** file (examine it) to use for batch-self-checking your class. These are rigorous but not exhaustive tests. Incrementally write and test your class; check each method as you write it.

Note that when exceptions are raised, they are printed by the driver but the **Command:** prompt sometimes appears misplaced.

You can write other code at the bottom of your **bag.py** module to test the **Bag** class, or type code into the driver as illustrated below. Notice the default for each command is the command previously entered.

```
Driver started
Command[!]: from bag import Bag
Command[from bag import Bag]: b = Bag(['d','a','b','d','c','b','d'])
Command[b = Bag(['d','a','b','d','c','b','d'])]: print(b)
Bag(a[1], b[2], c[1], d[3])
Command[ print(b)]: print(len(b))
7
Command[print(len(b))]: print(b.count('d'))
3
Command[print(b.count('d'))]: quit
Driver stopped
```

Problem #2: Interval Class

Problem Summary:

Write a class that represents and defines operators for **Interval** numbers, which are represented by a pair numeric values: **int**, **float** or mixed. We use intervals to represent approximate numbers, whose exact value we do not know. For example, in physics we might know that the acceleration produced by the the force of gravity (**g**) at sea level is 9.8 m/s^2 +/- .05 m/s^2, which we will write as **9.8(+/-.05) m/s^2**. With this class we can perform numeric calculations on **Interval** objects, which automatically keep track of the precision for each calculated value.

For example, the formula **sqrt(d/(2*g))** computes the amount of time it takes for an object at sea level to fall a given distance (**d**) in a vacuum. Given our approximation for **g**, and a distance that is 100(+/-1) m, we can use the **Interval** class to compute the amount of time it takes for an object to drop this amount as follows, including the precision with which we know the answer.

```
g = Interval.mid_err(9.8,.05)
d = Interval.mid_err(100,1)
t = (d/(2*g)).sqrt()
print(t)
```

So, with **g** known +/-.05 m/s^2, and **d** known +/-1 m, the results print as **2.2587923826805945(+/-0.017056289680373204)**), which indicates that the time will be somewhere between about 2.24 and 2.28 seconds, having a relative error of about .76%. Note that each **Interval** object will store the minimum and maximum value in the interval. So **9.8(+/-.05)** is stored as an **Interval** with a minimum of **9.75** and a maximum of **9.85**.

Details

1. Define a class named **Interval** in a module named **interval.py**
2. Define an **__init__** method that has two parameters; their arguments specify the minimum and maximum values in the interval respectively. Store them in the **self** variables **min** and **max**. Programmers will not use this method directly to construct **Interval** objects; instead, they will use the static **Interval.min_max** and **Interval.mid_err** methods (described below).

   For information about static methods, read the Class Review lecture notes (look for the entry on Static Methods near the bottom, before the problems).
3. Define a static **min_max** method that has two parameters; their arguments specify the minimum and maximum values in the interval. The second parameter is optional, with **None** as its default value. This method should raise an **AssertionError** exception, with an appropriate message, if (a) the first argument is not an **int** or **float** numeric type or (b) if the second argument is not a numeric type or **None**, or (c) the first argument is greater than the second; if the second argument is **None**, use the first argument for both the minimum and maximum value (creating an interval with one value representing exactly that number). Return the appropriate **Interval** object.
4. Define a static **mid_err** method that has two parameters; their arguments specify the middle value and the +/- error for the interval. The second parameter is optional, with **0** as its default value. This method should raise an **AssertionError** exception, with an approprate message, if (a) the first argument is not an **int** or **float** numeric type, or (b) if the second argument is not a numeric type, or (c) if the second argument is negative. Return the appropriate **Interval** object. For example, **Interval.mid_err(9.8,.05)**. would produce the same object as **Interval.min_max(9.75,9.85)**.
5. Define the methods **best**, **error**, and **relative_error**
   - **best** returns the best approximation for the **Interval** (the value at its middle).
   - **error** returns the error for the **Interval**: half the distance between its minimum and maximum values.
   - **relative_error** returns the absolute value of the ratio between the error over the best approximation as a percentage (multiply by 100).
6. Define a **__repr__** method that returns a string, which when passed to **eval** returns a newly constructed **Interval** with the same value as the object **__repr__** was called on.
7. Define a **__str__** method that returns a string, with the best approximation for the **Interval** followed in parentheses by **+/-** followed by error bounds on the **Interval**. For example, **str(Interval.mid_err(9.8,.05))** returns **9.8(+/-.05)**
8. Define a **__bool__** method that returns **True** if the **Interval** represents an interval whose error is not zero: it has different minimum and maximum values.
9. Define all the underscore methods needed to ensure the prefix +/- work correctly: **+** returns the same interval while **-** returns the appropriate negated interval.
10. Define all the underscore methods needed to ensure that the add, subtract, multiply, and true divide (/) operators produce the correct answers when their operands are any combination of an **Interval** object with an **Interval**, **int**, or **float** object.

    Treat **int** and **float** values as exact/precise values (with no error). In all cases, carefully determine from the minimum and maximum values of the operand **Interval**(s), what the minimum and maximum values are in the result **Interval**: doing so requires a bit of deep thinking about arithmetic operators; in some cases it is useful to think about whether **Interval** is all negative, all positive, or contains 0: having a minimum <= 0 and maximum >= 0.

    If Python tries to apply an arithmetic operator to an **Interval** object and any other type of value, return the standard value **NotImplemented** (which will ultimately cause Python to raise the standard **TypeError** exception with the standard messsage about unsupported operand types: see what **1+'a'** produces in the Python interpreter). If a divisor's interval includes zero, it should raise the **ZeroDivisionError**, including an appropriate message with the offending denominator.
11. Define all the underscore methods needed to ensure that the exponentiate operator (**) produces the correct answers when their left operand is an **Interval** object and their right operand is restricted to be an **int** object. Use repeated multiplication to solve this problem. In any other case, return the standard value **NotImplemented** (which will ultimately cause Python to raise the standard **TypeError** exception with the standard message). Note that if **p** is negative, computing **a**p** is equivalent to computing **(1/a)**(-p)**.
12. Define all the underscore methods needed to ensure that we can compare **Interval** objects using the six standard relational operators, with any combination of an **Interval** object and an **Interval**, **int**, or **float** object.

    If Python tries to apply a relational operator to an **Interval** object and any other type of value, return the standard value **NotImplemented** (which will ultimately cause Python to raise the standard **TypeError** exception with the standard message about unsupported operand types: see what **1<'a'** produces in the Python interpreter).

    For **==** two **Interval**s are equal if they have the same minimum and maximum values; they **!=** if either is different. Likewise, an **int** or **float** is **==** to an **Interval** if the minimum and maximum values of the interval are the same as the **int** or **float**.

    For **<**, **<=**, **>**, **>=** the rules are more complex: they depend on whether the **Interval** class's **compare_mode** attribute is set to **'liberal'** or **'conservative'**. For example, when using the **<** operator in **'liberal'** mode, the left argument is **<** the right when its best value is **<** the right's best value (or the right itself, if it is an **int** or **float**). When using the **<** operator in **'conservative'** mode, the left argument is **<** the right when its maximum value is **<** the right's minimum value (or the right itself, if it is an **int** or **float**). The other operators work simliarly in **'liberal'** and **'conservative'** mode.

    In fact, these four operators should raise an **AssertionError**, with an appropriate message, if the **Interval** class has no **compare_mode** attribute or this attribute is bound to anything other than **'liberal'** or **'conservative'**. Initially, the **Interval** class should have no **compare_mode** attribute.
13. Python automatically provides meanings for **+=**, **-=**, ***=**, **/=**, and ****=**.
14. Define
    - an **__abs__** method for **Interval** values (hint: what if the interval is all negative, positive, stradles 0)
    - a **sqrt** method for **Interval** values.
15. Define a **__setattr__** method that ensures objects in the **Interval** class are immutable. The methods that you will write should never bind any instance name (except in **__init__**, which initializes them) but exclusively returns newly constructed **Interval** objects with the correct values. If an attempt is made to mutate an object (by defining a new attribute or rebinding an existing attribute), raise an **AssertionError** with an appropriate message.

Hint: I wrote four helper methods (all starting with _) to perform common tasks (four were static methods) in my class. If you find yourself writing similar code over and over, try to define and call a helper method to do the job. I also made use of the form **f(*g(..))** where function **g** returns a 2-tuple which * turns into 2 arguments for calling **f**.

Testing

The **interval.py** module includes a script that calls **driver.driver()**. The project folder contains a **bsc2.txt** file (examine it) to use for batch-self-checking your class. These are rigorous but not exhaustive tests. Incrementally write and test your class: for example, getting one arithmetic (or relational) operator working correctly will create a pattern for the others.

Note that when exceptions are raised, they are printed by the driver but the **Command:** prompt sometimes appears misplaced.

We can write other code at the bottom of your **interval.py** module, before calling the driver, to test our **Interval** class: e.g., when testing **+** we can just define two **Interval**s and print their sum. You can also type such code into the driver as illustrated below, but if you want to perform the same test over and over again when debugging, it it better to put this code before the driver is called. Notice the default for each command is the command previously entered.

```
Driver started
Command[!]: from interval import Interval as I
Command[from interval import Interval as I]: g = I.mid_err(9.8,1)
Command[g = I.mid_err(9.8,1)]: print(g)
9.8(+/-1.0)
Command[print(g)]: print(2*g)
19.6(+/-2.0)
Command[print(2*g)]: I.compare_mode = 'conservative'
Command[I.compare_mode = 'conservative']: print(9.75<g)
False
Command[print(9.75<g)]: print('9.75'<g)
Traceback (most recent call last):
  ...
TypeError: unorderable types: interval.Interval()>str()
Command[print('9.75'<g)]: quit
Driver stopped
```