Program 2

Classes, Overloaded Operators, and Iterators

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 (one in each 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

- Define a class named **Bag** in a module named **bag.py**
- Define an **__init__** method that has one parameter, an iterable of values that initalize 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.
- 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.
- 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.
- 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**.
- 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__**.

- Define a **__contains__** method that returns whether or not its argument is in the **Bag** (one or more times).
- 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**.
- 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**.
- 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.
- 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.
- 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 this method should not change either **Bag**.
- 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** during 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**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; 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[len(b)]: print(len(b))
7
Command[print(len(b))]: print(b.count('d'))
3
Command[print(b.count('d'))]: quit
Driver stopped

Problem #2: Sparse Matrix Class

Problem Summary:

A **Matrix** is a 2-dimensional type storing values indexed by a row and a column. For example, the following is a 3x3 Matrix (its 3 rows and columns are numbered starting at 0; so 0-2). Here, the value indexed by **Row 1** and**Column 2** is **3**.

Indexes	Column 0	Column 1	Column 2
Row 0	1	0	0
Row 1	0	5	**3**
Row 2	0	0	1

We could choose to store this Matrix by using a tuple of rows; where each row is a tuple of values in its columns.

((1, 0, 0), (0, 5, 3), (0, 0, 1))

But in this problem, we will use a more interesting data structure.

A **Sparse Matrix** stores the row and column size of a Matrix, and a dictonary whose keys are 2-tuples (row and column index) and their associated **non-zero values**: any index in the Matrix that is not a key in its dictionary stores **0** implicitly; no key in its dictionary should store a **zero** value.

A Sparse Matrix would store the 9 values in the Matrix above in a dictionary of 4 index keys (and their associated **non-0** values). The remaining 5 **0**s are stored implicitly: any index key in the Sparse Matrix but not in the dictionary (e.g., the **(0,1)** tuple) implicitly stores **0**.

{(0,0): 1, (1,1): 5, (1,2): 3, (2,2): 1}

Write a class named **Sparse_Matrix** that represents and defines operators for Sparse Matrix objects, which are represented by a dictionary whose keys are 2-tuples representing a row and column of the Sparse Matrix and whose associated values are the non-zero matrix values stored at that row and colum. Operators will typically be defined to work on **Sparse_Matrix** objects, or between a **Sparse_Matrix** and a numeric (**int** or **float**) value.

The **Sparse_Matrix** class will be be immutable except for the **__call__** method, which can change the number of rows and columns in a matrix (and of course the **__setitem__** and **__delitem__** methods). All other methods will not mutate their **Sparse_Matrix** objects; methods that must return a **Sparse_Matrix** (like the arithmetic operators) will return a newly_constructed **Sparse Matrix** storing the appropriate values.

Details

1. Define a class named **Sparse_Matrix** in a module named **sparsematrix.py**.
2. Define an **__init__** method that has two required parameters (the number of rows and columns specifying the size of the Sparse Matrix), followed by any number (zero or more) of 3-tuple arguments: each will contain a (1) row and a (2) column index, and a (3) value that is to be stored at that index.

3. For example, writing **Sparse_Matrix(2, 2, (0,0,5), (1,1,5))** represents a Sparse Matrix whose value at indexes **(0,0)** and **(1,1)** is **5**, and whose values at all other indexes are **0**. The **__str__** method (read its definition below) displays such a Sparse Matrix as
4. **2x2:[5 0**
   0 5]
5. showing it size and all its values: both **non-0** and **0**. The dictionary representing this **Sparse_Matrix**stores only the two **non-0** values: **{(0,0): 5, (1,1): 5}**
6. **IMPORTANT:** Store a Sparse Matrix using the attributes named **rows**, **cols**, and **matrix** (which is a dictionary (**dict**) whose keys are 2-tuples (row,column), each of which is associated with a **non-zero**value). **Store only these attributes: no others.** You must set these attributes as **self.rows**, **self.cols** and**self.matrix**, because I have defined the **__str__** method and various **batch self-tests** to refer to these attributes, by these names.
7. This dictionary should **never store** an associated value of **0**: if the third value of a 3-tuple supplied to **__init__** is **0** ignore it: **do not store it in self.matrix**. So **Sparse_Matrix(2, 2, (0,0,5), (1,1,5), (0,1,0))**still stores only the two **non-0** values in its dictionary: **{(0,0): 5, (1,1): 5}**.
8. The **__init__** method should raise an **AssertionError** with an appropiate message if the arguments violate any of the following conditions:
   - The row and column arguments must be integers that are strictly greater than **0**.
   - The row and column in each triple must be non-negative integers that are strictly less than the number of rows and columns in the Sparse Matrix: e.g., if the Sparse Matrix specifies 3 rows, then all the row indexes must be 0, 1, or 2.
   - The row and column in each triple must be unique (not repeated). Raise an exception on just the first duplicate index (if there are any): you don't have to find all the duplicates, just raise an exception on the first.
   - The value in each triple must be numeric (an **int** or **float**).
9. For example, in my code writing **Sparse_Matrix(3,2, (0,0,1), (0,0,2))** raises **AssertionError** with the message **Sparse_Matrix.__init__: repeated index (0, 0)**.
10. Define a **size** method that returns a 2-tuple containing the row and column size of a Sparse Matrix. For **m = Sparse_Matrix(3,2)** (a Sparse Matrix filled with **0**s, so its dictionary is empty) **m.size()** returns **(3,2)**. You will find this method useful when you write later methods, to check for compatible sizes for Sparse Matrices.
11. Define a **__len__** method that returns the number of values (counting all **non-zero** and **0** values) in the Sparse Matrix: it returns the product of the number of rows and columns.
12. Define a **__bool__** method that returns **False** if the **Sparse_Matrix** object stores all zero values; it returns**True** if it returns any non-zero values. Recall that the dictionary stores only non-0 values.
13. Define a **__repr__** method that returns a string, which when passed to **eval** returns a newly constructed**Sparse_Matrix** object that represents the same **Sparse_Matrix** object on which **__repr__** was called.
14. For example, if we define **m = Sparse_Matrix(3,3, (0,0,1), (1,1,1), (2,2,1))**, calling **repr(m)** returns something like **'Sparse_Matrix(3, 3, (0, 0, 1), (1, 1, 1), (2, 2, 1))'**, although the order of the 3-tuples doesn't matter.
15. **Hint:**I used Python's **join** function to construct a string of all the 3-tuple values that appear near the end of what **repr** returns.
16. The following three methods are all similar in structure. You will find it useful to call these methods implicitly (using **[]** indexing) in other methods/operators you are asked to define below; but doing so is not necessary. **Hint 1**: **d.get(key,default)** returns **d[key]** if **key** is in the dictionary **d** and returns **default** if**key** is not in the dictionary **d**. **Hint 2**: writing **m[0,0]** calls **Sparse_Matrix.__getitem__(m,(0,0))**: i.e., the arguments **0,0** in **m[...]** are passed as the 2-tuple **(0,0)** to **__getitem__**.
    - Define a **__getitem__** method whose argument is row,column 2-tuple; it returns the value of the Sparse Matrix at that row and column. If the type of the argument is not a 2-tuple, whose first and second indexes are integers specifying a legal row then column in the Sparse Matrix, then raise a**TypeError** exception with an appropriate message.

- Define a **__setitem__** method whose arguments are a row,column 2-tuple and a value; it updates the value for that row,column. If the type of the first argument is not a 2-tuple whose first and second indexes are integers specifying a legal row then column in the Sparse Matrix, then raise a**TypeError** exception with an appropriate message. Also, if the value argument is not numeric (not an **int** or **float**), then raise a **TypeError** exception with an appropriate message. Otherwise update the Sparse Matrix to store the value at the row,column key. **Hint**: if the value argument is equal to**0**, update the dictionary only if it currently stores a non-zero value at that row,column key by removing the the row,column key from the dictionary. This method can mutate the state of a Sparse Matrix.
- Define a **__delitem__** method whose argument is row,column 2-tuple; it deletes that row,column key from the Sparse Matrix; it is equivalent to storing a zero at that row,column key. If the type of the argument is not a 2-tuple, whose first and second indexes are integers specifying a legal row then column in the Sparse Matrix, then raise a **TypeError** exception with an appropriate message. This method can mutate the state of a Sparse Matrix.

17. The following two methods are similar in structure. You will find it useful to call these methods in other methods/operators you are asked to define below (specifically, in the **details** and **__mul__** methods); but doing so is not necessary. In the examples directly below we define **m** so that **str(m)** is
18. **3x3:[0 2 3**
    4 0 6
    7 8 0]
19. Define the following two methods:
    - The **row** method that takes one **int** argument: it returns a tuple containing all the values in that row (from left to right). Using **m** defined above, **m.row(0)** returns **(0, 2, 3)**; **m.row(1)** returns **(4, 0, 6)**; and **m.row(2)** returns **(7, 8, 0)**. If the **row** argument is not an integer whose values specify a legal row in the Sparse Matrix, raise an **AssertionError** with an appropriate message.
    - The **col** method that takes one **int** argument: it returns a tuple containing all the values in that column (from top to bottom). Using **m** defined above, **m.col(0)** returns **(0, 4, 7)**; **m.col(1)** returns **(2, 0, 8)**; and **m.col(2)** returns **(3, 6, 0)**. If the **col** argument is not an integer whose values specifies a legal column in the Sparse Matrix, raise an **AssertionError** with an appropriate message.
20. Define the **details** method that returns a string of three pieces of information about a **Sparse_Matrix**object, separated by **->** : the size, the dictionary, and a tuple of all the rows (see the **row** method above): row 0, row 1, etc. If we define **m = Sparse_Matrix(3,3, (0,0,1), (1,1,5), (2,2,1))**, then the of calling**m.details()** is
21. **'3x3 -> {(0, 0): 1, (1, 1): 5, (2, 2): 1} -> ((1, 0, 0), (0, 5, 0), (0, 0, 1))'**
22. Of course, the keys in the dictionary (2nd part) can appear in any order, but the tuples (3rd part) must occur in the exact order shown.
23. I have defined a **__str__** method that returns a string that nicely displays a Sparse Matrix in a 2-dimensional form. I wrote this method to work even if **__getitem__** is not implemented correctly; it calls**self.matrix.get((r,c),0)**, but if **__getitem** were correctly implemented it would call only **self[r,c]**.
24. Define a **__call__** method that takes two **int** arguments, specifying the new number of rows and columns in a Sparse Matrix; these must each be integers that are strictly greater than **0**; if not, raise an**AssertionError**, similarly to what you did in the **__init__** method. It modifies the Sparse Matrix by resetting its row and column size and **deleting** all values from the dictionary whose indexes lie outside the new size of the re-sized Sparse Matrix.
25. For example if we define **m = Sparse_Matrix(2, 2, (0,0,1),(0,1,1),(1,0,1),(1,1,1))** (a 2x2 Sparse Matrix filled with 1s) and we call **m(1,1)** then **m** becomes a 1x1 Sparse Matrix whose **dict matrix** stores just**{(0,0): 1}** because all the other indexes (that used to be in the **dict matrix**) are now outside the size of Sparse Matrix.
26. This method mutates the state of a Sparse Matrix. **Hint:** You cannot delete any keys in a dictionary while iterating over it: iterate over a copy.

27. Define an **__iter__** method that produces 3-tuples containing the row, column, and value for each index in the Sparse Matrix that stores a **non-0** value. These values must be **sorted by the value (from smallest to biggest)**. For example, if we define **m = Sparse_Matrix(2, 2, (0,0,1),(0,1,3), (1,0,4), (1,1,2)))** which prints as
28. **2x2:[1 3**
4 2]
29. then iterating through it produces the tuples **(0,0,1)**, **(1,1,2)**, **(0,1,3)**, **(1,0,4)** in this order.
30. **Hint**: Write this method as a generator (covered in Friday's lecture in Week 4).
31. Define all the underscore methods needed to ensure the prefix +/- and **abs** work correctly: **+** returns a new Sparse Matrix with the same values; **-** returns a new Sparse Matrix with negated values (not that **-0 == 0**);**abs** returns a new Sparse Matrix with all non-negative values. **Hint**: I wrote each in 1 line, using a comprehension, and using ***** to turn a list into arguments for a function call.
32. Define all the underscore methods needed to ensure that the add, subtract, and multiply operators produce the correct answers when their operands are any combination of a **Sparse_Matrix** object with a**Sparse_Matrix**, **int**, or **float** object.
33. If Python tries to apply an arithmetic operator to a **Sparse_Matrix** object and any other type of value, raise the standard **TypeError** exception with the standard messsage about unsupported operand types: see what **1+'a'** produces in the Python interpreter.
34. Recall that none of these operators mutate their operands: each produces a new Sparse Matrix and returns it as a result. Note that both **+** and ***** (but not **-**) are commutative when applied to a Sparse Matrix and **a numeric value**. Commutivity with numeric values can make programming these methods simpler; so can using the infix **-** operator.
35. **Hint:** Create a Sparse Matrix that is the appropriate size, with no initial **non-0** values; then fill in all its indexes with the correct values: using **__getitem__** and **__setitem__** is simplest, for computing the resulting Sparse Matrix, ensuring that the dictionary contains no **0** values.
    - To add two **Sparse_Matrix** objects, their sizes must be the **same**; the the new resulting Sparse Matrix will be the same size as both its operand Sparse Matrices, and its values are the pair-wise addition of the two Sparse Matrix values. If we define **m1** (on the left) and **m2** (on the right) as follows
    - **2x3:[1 2 3** **2x3:[-1 2 -3**
      4 5 6] **4 -5 1]**
    - Then the result of adding these two Sparse Matrix objects (either **m1+m2** or **m2+m1**) is the new Sparse Matrix object
    - **2x3:[0 4 0**
      8 0 7]
    - For example, the value (**0**) in the resulting Sparse Matrix at row 0 and column 0 is the sum of the value (**1**) in **m1** at row 0 and column 0 plus the value (**-1**) in **m2** at row 0 and column 0. This pattern repeats for all the other rows and columns in the resulting Sparse Matrix.
    - To add a numeric value to a Sparse Matrix, the new resulting Sparse Matrix will be the same size as the one operand that is a Sparse Matrix, and have that numeric value added to each of the values in the argument Sparse Matrix. If we define the Sparse Matrix **m** as follows
    - **2x3:[-1 0 1**
      -2 0 2]
    - Then the result of adding **1** to this Sparse Matrix (**1+m** or **m+1**: adding **1** on the left or on the right) is the new Sparse Matrix
    - **2x3:[0 1 2**
      -1 1 3]
    - If the argument is not a Sparse Matrix or numeric value, raise a **TypeError** with the appropriate information; if the argument is a Sparse Matrix of a different size (see the **size** method), raise an**AssertionError** with the appropriate information.
    - To subtract two **Sparse_Matrix** objects, use the identity that **m1-m2** = **m1 + -m2**: the difference between **m1** and **m2** is the sum of **m1** and the negation (see part 13) of **m2**.

- To multiply two Sparse Matrix objects, their sizes must be **compatible**: the number of **columns in the left** Sparse Matrix must be the same as the number of **rows in the right** Sparse Matrix; the new resulting Sparse Matrix will have the number of **rows of the left** Sparse Matrix and the number of**columns of the right** Sparse Matrix. The value in the resulting Sparse Matrix at row **r** and column**c** is computed by adding together the pairwise products of all the values in **row r in the left** Sparse Matrix and in **column c in the right** Sparse Matrix.
- If we define **m1** (left) and **m2** (right) as follows
- **2x3:[1 2 1 3x2:[0 1**
  2 1 2] 2 0
  1 1]
- Then the result of multiplying these two Sparse Matrix objects is the new Sparse Matrix object
- **2x2:[5 2**
  4 4]
  - The value at index (0,0) is the sum of the pairwise multiplication of row 0 in **m1** (1, 2, 1) and column 0 in **m2** (0, 2, 1): 1*0 + 2*2 + 1*1 = 5.
  - The value at index (0,1) is the sum of the pairwise multiplication of row 0 in **m1** (1, 2, 1) and column 1 in **m2** (1, 0, 1): 1*1 + 2*0 + 1*1 = 2.
  - The value at index (1,0) is the sum of the pairwise multiplication of row 1 in **m1** (2, 1, 2) and column 0 in **m2** (0, 2, 1): 2*0 + 1*2 + 2*1 = 4.
  - The value at index (1,1) is the sum of the pairwise multiplication of row 1 in **m1** (1, 2, 1) and column 1 in **m2** (1, 0, 1): 1*1 + 2*0 + 1*1 = 4.
- **Hint**: Use the **row** and **col** method specified in part 8.
- To multiply a numeric value by a Sparse Matrix, the new resulting Sparse Matrix will have that numeric value multiplied by each of the **non-0** values in the Sparse Matrix. If we define the Sparse Matrix **m** as follows
- **2x3:[-1 0 1**
  -2 0 2]
- Then the result of multiplying **2** by this Sparse Matrix (either **2*m** or **m*2**; multiplying on the left or right) is the new Sparse Matrix
- **2x3:[-2 0 2**
  -4 0 4]
- Note that if the numeric value is **0** the Sparse Matrix will store all **0**s.
- If the argument is not a Sparse Matrix or numeric value, raise a **TypeError** with the appropriate information; if the argument is a Sparse Matrix that is not of a compatible size (see the **size**method), raise an **AssertionError** with the appropriate information.

36. Define all the underscore method needed to ensure that the power operator (**) produces the correct answers when its left operand is a square **Sparse_Matrix** object (same number of rows as columns) and its right operand is restricted to be a non-negative **int** object. Compute the answer by using repeated multiplication. If the power is not an **int** raise a **TypeError**; if the power is less than **1** or the Sparse Matrix is not square, raise an **AssertionError**.
37. Python automatically provides meanings for **+=**, **-=**, ***=**, and ****=**.
38. Define the relational operator **==** and **!=**. These operators must work correctly when one operand is a Sparse Matrix and the the other operand is a Sparse Matrix or a numeric value (**int** or **float**). For any other operands, return **False** for **==** and **True** for **!=**.
39. Two **Sparse_Matrix** objects are considered equal if they have the **same** sizes (see the **size** method) and pair-wise the same values: i.e., the value in row r,column c in one Sparse Matrix is equal to the value in row r,column c in the other Sparse Matrix, for all rows and columns. A Sparse Matrix is considered equal to a numeric value if the Sparse Matrix is any size, but all its values are all equal to the numeric value; otherwise they are unequal.
40. **Hint for the numeric value comparison with ==**:
    - If the value to check is **0**, there must be no values stored in the dictionary representing the **non-0**values in the Sparse Matrix.

    - If the value to check is **non-0**, that value must be stored in every index in the Sparse Matrix with this value: that is, every index must appear in the **matrix dict**, because there must be no **0** values in the Sparse Matrix.
41. Define a **__setattr__** method that ensures objects in the **Sparse_Matrix** class cannot store new attributes: they store only **rows**, **cols**, and **matrix**. The methods you will write should never bind any instance names (except in **__init__**, which initializes **rows**, **cols**, and **matrix**) but exclusively returns newly constructed**Sparse_Matrix** objects with the correct values. If an attempt is made to add new attributes to an object (by defining a new attribute or rebinding an existing attribute), raise an **AssertionError** with an appropriate message.
42. Do not attempt to solve this part of the problem until all other parts are working correctly.
43. **Important:** To solve this problem correctly, you must also find a way to allow the **__call__** method (which mutates **rows** and **cols**) to work correctly; this is hard to do. If you cannot, it best just to not write the **__setattr__** method.

Testing

The **Sparse_Matrix.py** module includes a script that does some simple matrix calculations and then 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: for example, getting one arithmetic 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.
You can also test code you type 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 in the script before the driver is called. Notice the default for each command (printed in the square brackets) is the command previously entered.

```
Driver started
 Command[!]: from sparse_matrix import Sparse_Matrix as SM
 Command[from sparse_matrix import Sparse_Matrix as SM]: m =
SM(2,2,(0,0,0),(0,1,1),(1,0,2),(1,1,3))
 Command[m = SM(2,2,(0,0,0),(0,1,1),(1,0,2),(1,1,3))]: print(m)
 2x2:[0  1
    2  3]
 Command[print(m)]: print(m.details())
 2x2 -> {(0, 1): 1, (1, 0): 2, (1, 1): 3} -> ((0, 1), (2, 3))
 Command[print(m.details())]: print(m+m)
 2x2:[0  2
    4  6]
 Command[print(m+m)]: print(m+1)
 2x2:[1  2
    3  4]
 Command[print(m+1)]: print(m*m)
 2x2:[ 2   3
     6  11]
 Command[print(m*m)]: quit
 Driver stopped
```