For my SymbolTable I implemented one HashTable which can be used for both identifiers and constants.

The method that I chose to solve collisions is SeparateChaining. To implement that I needed a SLL data structure that in my code is represented by the class SLL.

def insert(self,symbolkey):

- This method is used to insert an element into the SLL (an element is going to be inserted at the beginning of the list for a complexity of O(1) whenever there is more than one element having the same hashcode)

An important aspect of the SLL's implementation is the class Node that represents one entry in the SLL. Each node has a value and a pointer to the next node in the list.

m – the dimension of the hashtable

n – the number of elements from the hashtable

def hash_function(self,symbolkey):

- This method represents a hash function; each element of the hashtable will be added in the hashtable using the corresponding hash value calculated using this function

def resize_and_rehash(self) :

- This method is used to resize and rehash the hashtable when the load factor (n/m) is greater than 0.7
- For the resize the value of m is doubled and a new table with the new dimension is created
- All the elements from the previous table are copied into the new one but with a different hash because the value of m has changed
- The value of the table is changed to the new hashtable of double dimension

def search(self,k):

- Using the key k a value is searched in the table and if the key is in the table the method returns true else it returns false

def insert(self,symbolkey):

- This method is used to insert elements in the hashtable by finding the SLL corresponding to the hash value and inserting the key in that SLL
- Here is performed the checking of the load factor to be > 0.7

def main():

- Here I display the hashtable using some test values (key1,key2,key3,z,2)