Bryan Jensen's

Programming Assignment #2: BigNumber

Date submitted: 2 / 19 / 2013

Total Number of Pages: 6

Jalanorian / gist:4992707

Created in a few seconds

gistfile1.py — Python

```python
from random import randint
from time import time

class LinkedList(object):
  maxSize = 9
        maxInt = 10**maxSize

        def __init__(self, data=None):
                try:
                        self.data = int(data)
                except TypeError:
                        self.data = data

                self.nextIndex = None

        def append(self, data):
                if data == 0:
                        return

                if self.data == None:
                        self.data = int(data)

                elif self.nextIndex == None:
                        self.nextIndex = LinkedList(data)

                else:
                        self.nextIndex.append(data)

        def __str__(self):
                if self.nextIndex != None:
                        return ( str(self.nextIndex) + "".join(["0" for i in range(self.maxSize-len(str(self.data)))]) + str(self.
                else:
                        return ( "".join(["0" for i in range(self.maxSize-len(str(self.data)))]) + str(self.data) ).lstrip("0")

        def __add__(self, other):
                return self.recurAdd(other)

        def recurAdd(self, other, newList=None, carry_one=0):
                if newList == None:
                        newList = LinkedList()

                listSlot = self.data + other.data + carry_one

                if listSlot >= self.maxInt:
                        newList.append(listSlot%self.maxInt)
                        carry_one = 1

                else:
                        newList.append(listSlot)
                        carry_one = 0

                if self.nextIndex == None and other.nextIndex == None:
                        newList.append(carry_one)
                        return newList
                elif self.nextIndex == None:
                        newList.append_dump(other.nextIndex, carry_one)
                        return newList
                elif other.nextIndex == None:
                        newList.append_dump(self.nextIndex, carry_one)
                        return newList
                else:
                        return self.nextIndex.recurAdd(other.nextIndex, newList, carry_one)

        def append_dump(self, dumpingList, leftOverData=0):
```

```
                self.append(dumpingList.data+leftOverData)

                if dumpingList.nextIndex != None:
                        self.append_dump(dumpingList.nextIndex)

        def shift_left(self):
                takenOff = self.data // 10**(self.maxSize-1)
                self.data = self.data % 10**(self.maxSize-1) * 10
                if self.nextIndex != None:
                        self.nextIndex.shift_left_recur(takenOff)
                elif self.nextIndex == None and takenOff > 0:
                        self.nextIndex.append(takenOff)

        def shift_left_recur(self, addTo):
                takenOff = self.data // 10**(self.maxSize-1)
                self.data = ( self.data % 10**(self.maxSize-1) * 10 ) + addTo
                if self.nextIndex != None:
                        self.nextIndex.shift_left_recur(takenOff)
                else:
                        self.append(takenOff)



        def shift_right(self):
                if self.nextIndex != None:
                        takenOff = self.data % 10
                        self.data = (self.data // 10) + (10**(self.maxSize-1))*self.nextIndex.shift_right()
                        return takenOff
                else:
                        takenOff = self.data % 10
                        self.data = self.data // 10
                        return takenOff

class BigNumber(object):
        maxSize = -1

        def __init__(self, s):

                if type(s) == LinkedList:
                        self.list = s
                        self.maxSize = self.list.maxSize

                else:
                        self.list = LinkedList()

                        self.maxSize = self.list.maxSize

                        if s == "":
                                s = "0"

                        if len(s) % self.maxSize == 0:
                                for i in range(len(s)/self.maxSize, 0, -1):
                                        self.list.append(s[self.maxSize*(i-1):self.maxSize*(i)])

                        else:
                                leftOver = len(s) % self.maxSize

                                for i in range((len(s)-leftOver)/self.maxSize, 0, -1):
                                        self.list.append(s[(self.maxSize*(i-1))+leftOver:(self.maxSize*(i))+leftOver])

                                self.list.append(s[:leftOver])


        def __str__(self):
                return str(self.list)

        def __add__(self, other):
                if self.list.data == 0 and other.list.data == 0:
                        result = 0
                elif (self.list.data != None and other.list.data != None):
                        result = self.list + other.list
                elif self.list.data != None:
                        result = str(self.list)
                elif other.list.data != None:
                        result = str(other.list)
                else:
                        result = 0

                return BigNumber(result)

```

```
143         def shift_left(self):
144                 return self.list.shift_left()
145
146         def shift_right(self):
147                 return self.list.shift_right()
148
149
150 def main():
151         numStr1 = "".join([str(randint(0,9)) for x in range(randint(0,50))])
152         numStr2 = "".join([str(randint(0,9)) for x in range(randint(0,50))])
153
154         #numStr1 = "5489641854681"
155         #numStr2 = "841584138515"
156
157         #print "         |%s|" % numStr1
158         #print "          |%s|" % numStr2
159
160
161         BN1 = BigNumber(numStr1)
162
163         print "\nStart: %s" % BN1
164
165         BN1.shift_right()
166         BN1.shift_left()
167         BN1.shift_right()
168
169         print "End:   %s" % BN1
170         print "Wanted:%s" % numStr1[:-1]
171
172         BN2 = BigNumber(numStr2)
173
174         print "\nStart: %s" % BN2
175
176         BN2.shift_left()
177
178         print "End:   %s" % BN2
179         print "Wanted:%s" % (numStr2+"0")
180
181         n = 1000000
182
183         iTime = time()
184         for i in range(n):
185                 BN3 = BN1 + BN2
186         print "\nTime for %i additions of BigNumber: " %n, time() - iTime
187
188         L1 = long(numStr1)
189         L2 = long(numStr2)
190
191         iTime = time()
192         for i in range(n):
193                 L3 = L1 + L2
194         print "Time for %i additions of Long: " %n, time() - iTime
195
196         if len(numStr1) == 0:
197                 numStr1 = "0"
198         if len(numStr2) == 0:
199                 numStr2 = "0"
200
201         print "\nGot:     %19s" % (BN1 + BN2)
202         print "Wanted:  %19s" % (int(numStr1[:-1]) + int(numStr2+"0"))
203
204         if str(BN1 + BN2) == str(int(numStr1[:-1]) + int(numStr2+"0")):
205                 print "\nSuccess.\n"
206         else:
207                 print "\n\nNOPE.\n\n"
208
209
210
211
212 if __name__ == '__main__':
213         main()
```

Output From Main():

Start: 7730438739270344733104881
End: 773043873927034473310488
Wanted:773043873927034473310488

Start: 8518427422264668723
End: 85184274222646687230
Wanted:85184274222646687230

Time for 1000000 additions of BigNumber: 12.5720000267
Time for 1000000 additions of Long: 0.171999931335

Got: 773129058201257119997718
Wanted: 773129058201257119997718

Success.

Explorations:

1. I implemented a class as a new abstract data type, my envisioning of a LinkedList in this context. The LinkedList is far from modular as most of the functionality of the BigNumber is built in rather than performed thereon. The BigNumber class itself is simply used for a constructor, case-catcher and minimal functionality, with most of the method calls being passed on to the interior LinkedList. I chose this method of storage because, with my LinkedLists being recursive in nature with the next index inside of the previous, I can store an unlimitedly large number in my BigNumber class, with the only limitation being memory space. This is as opposed to a BigNumber class which has a set number of variables to store any incoming data, and therefore can only go so big.
2. Possible Future Limitations:
    a. When considering subtraction there needs to be an easy method of determining which value is larger and whether that will cause the result to be negative. As our BigNumber class was only built to store nonnegative integers, such subtraction would be severely limited. This could be implemented as easily as any of the other operators, with the simple addition of the method for determining relative size.
    b. Implementing multiplication would be completely possible but not nearly as easy as either addition or subtraction. That's not to say it would be insanely complicated either, since with the current setup (especially with the inclusion of left_shift()) we have some

of the tools needed already. It is also easy to run into overflow errors when multiplying two numbers together large enough to warrant the use of a BigNumber class.

c. Division would also be possible with many of the same considerations as multiplication, excluding overflow concerns. I believe that with the current methods, division would be more of a challenge than multiplication but also implementable with the current setup.

3. The largest number possibly stored in my BigNumber class is in the ballpark of 10^1782, and is limited by the maximum recursion depth of Python rather than memory restrictions as I had predicted. (I cannot show the output of that main() version, but the class call was: **BigNumber("".join([str(randint(0,9)) for x in range(1782)]))** and the output began: **795506433491403391056264…**)
4. The time to compute the addition of BigNumbers and longs both are too short for python to display for one run, but for 1 million additions the numbers become a bit more clear: ~12.5 seconds vs. ~.18 seconds. The default long implementation is about 73 times quicker than my created BigNumber class, which is the be expected as many people have spent much time writing and rewriting the code behind everything that Python gives by default. The algorithm behind the long type has been perfected to the extent required for it to no longer be practical to refine it, whereas my implementation is a simple, intuitive approach to the problem.