Linked List Bignums

CSCI 112, Labs 4 and 5

File names: Names of files, functions, and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

Individual work: All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code. Do not use code written by anyone else, in the class or from the internet.

Documentation: Each file should begin with a docstring that includes your name, the class number and name, the lab number, and a short description of the lab, as well as documentation pertinent to that particular file.

Addition and subtraction standard algorithm: You should be familiar with the standard algorithms for addition and subtraction, at least in base 10. If you are not familiar with other number bases, review them quickly here: `https://www.mathsisfun.com/numbers/bases.html`.

The standard subtraction and addition algorithms work fine in any base. You just have to remember that if you are in base 16, say, and you "borrow one" from the next column, you are borrowing 16, not 10. Likewise, you "carry one" to the next column when you get 16 or more, not 10. Here are some worked examples to get the hang of things:

Base 10 examples:

$$\begin{array}{rcccc} & 3 & 9 & 1 & 5 \\ + & 1 & 2 & 4 & 5 \\ \hline & 5 & 1 & 6 & 0 \end{array} \qquad \begin{array}{rcccc} & 3 & 9 & 1 & 5 \\ - & 1 & 2 & 4 & 5 \\ \hline & 2 & 6 & 7 & 0 \end{array}$$

Base 8 examples:

$$\begin{array}{rcccc} & 7 & 5 & 1 & 3 \\ + & 2 & 3 & 3 & 5 \\ \hline 1 & 2 & 0 & 5 & 0 \end{array} \qquad \begin{array}{rcccc} & 7 & 5 & 1 & 3 \\ - & 2 & 3 & 3 & 5 \\ \hline & 5 & 1 & 5 & 6 \end{array}$$

Base 16 examples:

$$\begin{array}{rcccc} & 15 & 4 & 11 \\ + & 4 & 13 & 13 \\ \hline 1 & 4 & 2 & 8 \end{array} \qquad \begin{array}{rcccc} & 15 & 4 & 11 \\ - & 4 & 13 & 13 \\ \hline & 10 & 6 & 14 \end{array}$$

If you want more examples, just run my program `arithmetic.py` and paste the output into a new project on `https://www.overleaf.com`.

You can get easy base 8 and base 16 examples from python (remember that in base 16 the digits 10 to 15 are: `a, b, c, d, e, f`):

```
>>> hex(0xaaa + 0x123)
'0xbcd'
>>> oct(0o666 + 0o123)
'0o1011'
```

Bignums: Python has bignums (arbitrarily high integers) built in. For example, Python has no problem computing:

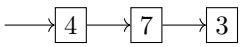
```
>>> 1234567890987654321 * 1234567890987654321
1524157877457704723228166437789971041
```

Even though computers only natively support either 32 bit integers, or 64 bit integers. Hence, the maximum integers supported in most programming languages are either $2^{32} = 4.294.967.296$ or $2^{64} = 18.446.744.073.709.551.616$. Integers larger than this are supported in software.

When Python encounters integers that are too big for the 32 or 64 bit registers, it automatically converts them to bignums. Python's bignums are implemented as dynamic arrays of digits in a large, fixed base.

In this lab we will build a Bignum class of our own that supports arbitrarily large integers using linked lists. Each cell in the linked list will hold a single position in the base. For example, if we choose base 10 for our bignums, each cell in the linked list will hold a Python integer from 0 to 9. If we choose base 16 for our bignums, each cell in the linked list will hold a Python integer between 0 and 15. If we choose 10,000 for our base, each cell in the linked list will hold a Python integer between 0 and 9,999.

Here, for example, is a figure of what the number `0o374` in base 8 looks like:



Note that the smallest digit will be at the head of the list. This will make arithmetic much easier.

Make your Bignum class general, so it will support any base.

Since there's no such thing as an "empty" bignum, we will not need the header class used for linked lists in the book. A bignum will hold the digit, the sign, and a pointer to the next bigger digit.

Signs: Positive and negative numbers will be represented by a separate field in the class called `Bignum.sign`, with the value of either plus or minus one.

Initializing with Python ints: To make testing easy, Bignums will be initialized by Python integers. The initializing integer can be any size, of course, using Python's native integers. You will convert this to your linked list representation of Bignums. Here is the initializing routine for bignums that you should use:

```
    def __init__(self, n, base = 17):
        self.base = base
        if n < 0:
            self.sign = -1
        else:
            self.sign = 1
        self.n = abs(n)
        self.next = None
        self.overflow()
```

Needless to say, the `Bignum.overflow` method will take care of normalizing the integer to the range `[0..(self.base-1)]` and pushing any overflows ("carries") along to the next cell in the list.

Extracting Python ints: Also to make testing easy, include a `Bignum.int` method that converts your bignum into a Python int. This will support easy testing expressions such as:

```
n = 2**100
self.assertEqual(n, Bignum(n).int())
```

String method: While it will be nice to see the bignums converted to Python integers for testing, I find it easier to debug if I can see what the data structure looks like. Build a `__str__` method that shows the contents of bignums like this:

```
>>> print(Bignum(20,8))
Bignum base 8: 4:2:+
>>> print(Bignum(20,2))
Bignum base 2: 0:0:1:0:1:+
>>> print(Bignum(-123,10))
Bignum base 10: 3:2:1:-
```

```
>>> print(Bignum(50,16))
Bignum base 16: 2:3:+
>>>
```

Absolute value addition: Once you've got your Bignum class working, implement the standard algorithm for addition, as illustrated above. These will only have to work for the absolute value of the bignums, we will handle the signs later.

The overflow, or carry, can be handled digit by digit, or you can simply allow each digit to be temporarily larger than the base, and fix them all at once after all the single-digit additions have been made. For example, as a first step, you can add like this (base 10):

$$\begin{array}{rcccc} & 3 & 9 & 1 & 5 \\ + & 1 & 2 & 4 & 5 \\ \hline & 4 & 11 & 5 & 10 \end{array}$$

and then correct the overflows: $4\ \ 11\ \ 5\ \ 10 \Rightarrow 5\ \ 1\ \ 6\ \ 0$

Note that you may have to extend the answer with new cells if the answer has more digits than the summands.

Lab 4: That's it for lab 4! Put `bignum.py` and `bignum_test.py` into a folder called `csci112lab04yourname`, zip it and turn it in by the due date.

Lab 5: For lab 5, complete the rest:

Absolute value subtraction: Once you've got addition working and tested, think about subtraction. The standard algorithm assumes that we're always subtracting a smaller from a larger. Implement this first.

Less than: You will have to implement a less than method in order to do subtraction in general. This can be done recursively. For example, to compare (base 10) 1234 with 3435, we first compare (recursively) 123 with 343. We find that the first is less than the second, so we can return true immediately; it doesn't matter what the smaller digits are.

On the other hand, if we compare 1234 with 1238, we first compare 123 with 123, and, finding them equal, we now have to compare the single digits 4 and 8 to see which is bigger.

Your recursive procedure is going to have to tell you whether the first is less than, equal to, or greater than the second. This is a more general opeation than a simple less than comparison. Traditionally this operator has been called `cmp`. Comparing two items it returns -1 if the first is less than the second, 0 if they are equal, and 1 if the first is greater than the second.

Write this `cmp` method, and then use it to define a `__lt__` method.

You can do this without recursion if you like. You might consider reversing the list of digits, and counting them.

Subtraction of larger from smaller: Now we can handle the case of subtracting a larger absolute value from a smaller, using the relation:

$$a - b = -(b - a)$$

Now fix your absolute value subtraction so that it works with numbers in either order.

General addition and subtraction: To support the `__add__` and `__sub__` methods for your Bignums in general you will have to do a little sign checking at the beginning. For example, to subtract a negative number from a positive number, just convert the negative to a positive and then add. There are a few other cases for you to work out: each operand can be positive or negative, and we might be adding or subtracting. Eight cases in all, right? They will look like this:

$$\begin{aligned} a + b &\Rightarrow a + b \\ a + -b &\Rightarrow a - b \\ -a + b &\Rightarrow b - a \\ -a + -b &\Rightarrow -(a + b) \end{aligned}$$

The subtraction cases are similar. In each case the general problem has been reduced to an addition or subtraction of absolute values, with an optional sign correction at the end.

Leading zeros: Some operatrions may result in leading zeros, for example, $12345 - 12300 = 00045$ in base 10. Your Bignum class should clean up any result by removing these unecessary zeros.

```
>>> print(Bignum(123456, 10) - Bignum(123000, 10))
Bignum base 10: 6:5:4:+
>>>
```

This is another case where you might want to think recursively. Just be sure to leave at least one zero, when the bignum is zero!

Also, you can do it nonrecursively, in which case you again might want to consider reversing the list of digits.

Recursion: Note, if you use recursion in your bignum calculations, you may run into recursion depth limits with very large numbers and very small bases. This will not be considered a bug.

Calculator: Use your infix calculator from the previous lab to make a Bignum calculator!

Turn in: Files `bignum.py` and `bignum_test.py` in folder `csci112lab05yourname`, zipped and turned into Canvas.

Optional additions:

- Multiplication can be done with the standard algorithm, or using the following facts:

$$ab = \begin{cases} 0 & \text{if } b = 0 \\ a + a(b-1) & \text{if } b \text{ is odd} \\ 2(a(b/2)) & \text{if } b \text{ is even} \end{cases}$$

  You will have to implement a divide by 2 function, but the rest can be done with recursion and addition.

- Division, exponentiation, *etc.*
- Run some experiments to see how the timing of your Bignums changes with the size of the base. Note that Python will not use native integers if the size of the integer is greater than the size of a native integer. Since your calculations can result in numbers in the cells up to 2(base), or base^2 if you do multiplication, your base should probably not exceed 2^{16} for 32 bit computers, or 2^{32} for 64 bit computers. (Why?)

  Once you find the best base for your implementation, compare times on your implementation with those for native Python bignums. How much slower are you? Do you think it makes a big-O difference, or not? Why? Can you prove it with some data?

  Write up your experiments and conclustions in a short document.

My implementation headers: You do not have to follow my implementation here, but in my Bignum class I used the following approach. Some of my functions are recursive, and some are iterative.

I have also supplied my unit test file, as a start, for yours.

```
def __init__(self, n, base = 17):
def __str__(self):
def int(self):
    '''convert to Python integer'''
def copy(self, end=None):
def add(self, other):
    '''add absolute values'''
def __add__(self, other):
    '''figure out signs, then add or sub and fix result'''
def sub(self, other):
    '''subtract absolute values, if self < other return negative of other-self'''
def __sub__(self, other):
    '''figure out signs then add or sub and fix result'''
def cmp(self, other):
    '''compare absolute values, return -1,0,1 for less, equal, greater'''
def __lt__(self, other):
    '''less than, using signs and cmp'''
def __eq__(self, other):
    '''equality, using signs and cmp, needed for unit tests'''
def remove_zeros(self):
    '''remove leading zeros'''
def inc_next(self, amt=1):
    '''if amt > 0 increment the next cell.  if absent, create it'''
def overflow(self):
    '''any cell that is larger than the base is reduced and carried to the next cell'''
def dec_next(self, amt=1):
    '''if amt > 0 decrement next cell, if absent, raise error'''
def underflow(self):
    '''any cell that is less than 0 is incremented by borrowing from the next'''
```