Writing a concurrent, pure functional sorting program in Python.

Assignment 3 for CSC 580, Spring, 2009
Dr. Dale E. Parson, http://faculty.kutztown.edu/parson
Due date is 11:59 PM on March 26

The goal of this assignment is for students to write a multithreaded, purely functional program in Python.

Perform the following steps to copy and inspect my initial code handout.

```
cp -pr ~parson/ThryLang/pfunk ~/ThryLang
cd ~/ThryLang/pfunk
```

Student changes go into file pfunksort.py. This file, which does not exist at present, takes 3 command line arguments.

```
python pfunksort.py INFILE OUTFILE THRESHOLD
```

INFILE is the path to a file containing a sequence of integer values, exactly one per line, separated by newline characters, possibly including negative values. There may be 0 or more lines. Program pfunksort.py must construct an array of integers to be sorted using this sequence. Any non-integer character sequences should trigger an exception handler that simply ignores these strings. The process should continue processing subsequent lines.

OUTFILE is the file to which to write the final, sorted sequence of integers read from INFILE, one per line.

THRESHOLD is a positive limit value, > 8, on the size of a subarray as defined below. The program should raise a **ValueError** with an appropriate message and terminate if this command line argument is not an integer > 8.

The program should also raise a **ValueError** with an appropriate message and terminate if it is invoked with an invalid number of command line arguments, or if the INFILE or OUTFILE cannot be opened for reading or writing respectively. Invalid integers within INFILE should not be treated as errors. They should be ignored. Floating point values should be truncated to integers and used. INFILE may contain no numbers to sort, in which case OUTFILE would be an empty file. This condition is not an error.

The program must use a pure functional, concurrent version of quicksort. You will design this algorithm. Attached below is a non-concurrent, non-functional C++ version of quicksort. For a given subarray, housed in a Python tuple, your program must determine whether the length of the subarray is <= **THRESHOLD**. If it is > than **THRESHOLD**, then the thread sorting that subarray first partitions the subarray into two new subarrays that contain elements that are respectively <= a pivot value and > that pivot value. The thread then starts a second thread to sort the > subarray, during which time the initial thread sorts the <= subarray concurrently. After both

threads have completed their work, the new thread terminates, and the initial thread constructs and returns a sorted subarray that is the concatenation of the two <= and > parts.
When the length of the subarray to be sorted is <= the THRESHOLD, use the current thread to sort both halves and combine them. Do not start a helper thread in this condition.

Selection of the pivot value must occur using the same enhancement as the C++ code below, i.e., selecting the median of the first, middle and final array element of the subarray to be sorted.

In addition to being concurrent, the ***ENTIRE PROGRAM*** must be a pure functional program. Python's interpreter has no way of enforcing pure functional programming. That is up to the programmer.

Pure functional programming means that there is no reassignment to variables and no mutation of constructed objects. All "changes" require construction of new objects from existing ones. This includes construction of the initial array from the input file through program terminated.

1. You may **not** use locks, timers, condition variables or semaphores as documented in PY chapter 20 on "Threads," because acquiring and releasing locks and similar operations on other synchronization objects constitute mutation. You may use the Thread start() and join() methods documented in PY 20. However, there is a logistical problem in calling join() directly, illustrated by the following code.

```
>>> from threading import *
>>> def valadd1(val):
...     print "valadd1 received ", val, " returns ", (val+1)
...     return val + 1
...
>>> valadd1(5)
valadd1 received  5  returns  6
6
>>> th = Thread(target=valadd1,args=(9,))
>>> x = th.start()
>>> valadd1 received  9  returns  10
y = th.join()
>>> print x, y
None None
```

Notice that join() does not return the return value from its startup function, valadd1()'s 10 in this case. Check out the following code, however.

```
>>> class MonadicThread(Thread):# Step 0: A miracle occurs! YOU MUST WRITE IT.
        # definition not shown
>>> mth = MonadicThread(target=valadd1,args=(9,))
>>> mx = mth.start()
>>> valadd1 received  9  returns  10
my = mth.join()
```

```
>>> print mx, my
None 10
```

See Rule #2 below for a hint on how to design **your MonadicThread class** that implements a value-returning join() method, where the value returned is the return value of the target function for the Thread constructor.

FUNCTIONAL PROGRAMMING RULES. Please do not violate these commandments.

#1. You may assign a value into a variable, but you may not reassign a value into a variable. The initial assignment is like the Scheme "let" operation. It binds a value to a symbolic name. Any subsequent assignment would be like the Scheme "set!" operation or one of its "!" suffixed kin. It would mutate a symbol binding. The Python compiler does not have distinct "let" versus "set!" operations; you must use assignment into a given variable only once. You may not reassign the binding of a variable after its initial assignment.

#2. You may assign a value into a field of a Python object, but you may not reassign a value into a field that is already bound. This unique assignment to a given field can occur within the object constructor or within any other method of that object.

#3. You may not mutate existing data structures in order to extend them. For example, you may not invoke .append() or .extend() on a list. One implication is that your "quicksort()" will not sort a list in place. Instead, it could construct subarrays by constructing new lists, constructing new, partially ordered lists, and merging lists into new lists using list concatenation. For example:

```
>>> lista = [3, 4, 5]
>>> listb = lista + [4, 5, 6]
>>> listc = [0, 1, 2] + listb
>>> print listc
[0, 1, 2, 3, 4, 5, 4, 5, 6]
```

There is no mutation is lists occurring in this example. All data structuring occurs through construction. List slicing to access a sublist is OK. You may not mutate a slice of an existing list, nor a single element, by placing its name on the left side of an assignment statement. The ONLY thing that may appear on the left side of an assignment state is a previously unbound variable or object field, or a parameter name in a keyword parameter assignment within a function call.

HOWEVER, The Python compiler does disallow mutation of tuples. Therefore, instead of using lists to represent unsorted and sorted arrays, **please use tuples**.

```
>>> tuplea = (3, 4, 5)
>>> tupleb = tuplea + (4, 5, 6)
>>> tuplec = (0, 1, 2) + tupleb
>>> tuplec
(0, 1, 2, 3, 4, 5, 4, 5, 6)
```

```
>>> (1, 2, 3) + (4, 5, 6)[1:4]
(1, 2, 3, 5, 6)
>>> ((1, 2, 3) + (4, 5, 6))[1:4]
(2, 3, 4)
>>>
>>> tuplea.append(tupleb)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> tuplea.extend(tupleb)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'extend'
```

Given the fact that Python disallows mutation of tuples (although not their elements), use tuples instead of lists for your arrays of ints.

#4. You may not use iteration constructs such as “for” loops or “while” loops, because they entail using reassignment of state variables in their tests. You must use recursion to achieve looping, passing parameters and return values that take the place of mutable variables in imperative programming.

Not that the following constructs are legal:

```
>>> vv = 10 # initial assignment
>>> range(1,vv)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [l+2 for l in range(1,vv)] # This is a *list comprehension*.
[3, 4, 5, 6, 7, 8, 9, 10, 11]
```

But the following is not legal because it reassigns into variable “i”:

```
>>> for i in range(1,vv):
...     print i
...
1
2
3
4
5
6
7
8
9
```

You can use recursion to get the same effect:

```
>>> def spill(l):
...     if (l):
...             print l[0]
...             spill(l[1:])
...
>>> spill(range(1,vv))
1
2
3
4
5
6
7
8
9
```

You could also use *map* for this example.

```
>>> from sys import stdout
>>> map(lambda x : stdout.write(str(x)+"\n"), range(1,vv))
1
2
3
4
5
6
7
8
9
[None, None, None, None, None, None, None, None, None]
```

The *lambda*, *map*, *zip*, *reduce* and *filter* functions and *list comprehensions* of PY 6 are all OK as long as you never reassign the value of a bound variable or field.

Below is the single-threaded, imperative C++ quicksort algorithm that you must rewrite as a multithreaded, purely functional Python program. Use **gmake turnitin** after it passes all regression tests. **Please adds some additional tests to the makefile**. Have fun.

```
1   // If there are at least 3 elements, pick a value that is >= one
2   // of them and <= the other as the pivot. This decreases the probability
3   // of hitting a degenerate case where all or almost all elements partition
4   // to one side of the pivot.
5   // The return value is the pivot, also swapped into iarray[left].
6   static int pickpivot(int iarray[], int left, int right) {
7      int result ;
8      int count = right - left + 1 ;
9      int mid = (left + right) / 2 ;
10     if (count < 3) {
11        result = iarray[left];  // [1] element or 50/50 odds for [2]
12     } else if ((iarray[mid] <= iarray[left] && iarray[left] <= iarray[right])
13           || (iarray[right] <= iarray[left] && iarray[left] <= iarray[mid])){
14        result = iarray[left];
15     } else if ((iarray[left] <= iarray[mid] && iarray[mid] <= iarray[right])
16           || (iarray[right] <= iarray[mid] && iarray[mid] <= iarray[left])){
17        result = iarray[mid] ;
18        iarray[mid] = iarray[left] ;
19        iarray[left] = result ;
20     } else if ((iarray[mid] <= iarray[right] && iarray[right] <= iarray[left])
21           || (iarray[left] <= iarray[right] && iarray[right] <= iarray[mid])){
22        result = iarray[right] ;
23        iarray[right] = iarray[left] ;
24        iarray[left] = result ;
25     }
26     return result ;
27  }
28
29  // partition is a helper function for quicksort().
30  // partition partitions an array into a left "half" and a right "half,"
31  // where all values in the left "half" or <= some "pivot" value, and
32  // all values in the  right "half" are > that "pivot" value.
33  // partition may not divide the array exactly in half; left and right
34  // "part" would be a better term.
35  // RETURN index of the element around which the array is partitioned.
36  static int partition(int iarray[], int left, int right) {
37     int pivot = pickpivot(iarray, left, right);
38     int low = left + 1 ;        // scan up looking for a too-high value
39     int high = right ;          // scan down looking for a too-low value
40
41     while (low < high) {      // scan from both sides of array
42        while (low <= high && iarray[low] <= pivot) { // in correct side
```

```
43              low++;
44          }
45          while (low <= high && iarray[high] > pivot) { // in correct side
46              high--;
47          }
48          // If there are two elements on the wrong side, swap them.
49          if (low < high) {
50              int temp = iarray[high];
51              iarray[high] = iarray[low];
52              iarray[low] = temp ;
53          }
54      }
55      // We still need to place the pivot in the correct spot.
56      while (high > left && iarray[high] >= pivot) {
57          high--;
58      }
59      // swap pivot with a lower value if there is one
60      if (pivot > iarray[high]) {
61          iarray[left] = iarray[high];
62          iarray[high] = pivot ;
63          return high;
64      } else {
65          return left ;
66      }
67  }
68
69  static void quicksort(int iarray[], int left, int right) {
70      if (left < right) {
71          // 1. Partition array so all values <= iarray[pivotlocation] are left
72          // of pivotlocation, and all values > pivotlocation are right of it.
73          int pivotlocation = partition(iarray, left, right);
74          // Sort the left partition.
75          quicksort(iarray, left, pivotlocation-1);
76          // Sort the right partition.
77          quicksort(iarray, pivotlocation+1, right);
78      }
79  }
```