category topic: CSC 135

csc135-midterm 01

WW09.2 | Tuesday, March 1, 2022 | 10:17 AM

1. Question 1 - `filter`, `len` - 2.5 points possible

Write a function named all_positive that takes a python list of numbers as a parameter and returns False if any of them are less-than or equal-to zero, and True otherwise. (This means the empty list should return True too.)

You will write the function twice. Once here and once in another Question. Both versions should use higher-order functions and no loops or list-comprehensions to solve the problem.

The version you write for this question should use filter to keep either the positive numbers or the non-positive numbers (the choice is yours), and then use the len() function to determine the answer.

```
def all_positive(lst):
    # return True if len(list(filter(lambda x: x > 0, lst,))) == len(lst) or
lst is None else False
        pass
```

2. Question 2 - `map`, `reduce` - 2.5 points possible

Write a function named all_positive that takes a list of numbers as a parameter and returns False if any of them are negative or zero, and True otherwise. (This means the empty list should return True too.)

You will write the function twice. Once here and once in another Question. Both versions should use higher-order functions and no loops or list-comprehensions to solve the problem.

The version you write for this question should use map to create a list of booleans, and then use reduce to determine the answer. To support length 0 and 1 lists, use reduce(func, list, init) with three parameters where the third parameter is the initial value of the reduction (ie, func(init, list[0]) is the first invocatin of func instead of func(list[0], list[1]).

```
from functools import reduce

def all_positive(lst):
        pass
```

3. Question 3 - `list135` sorted (Part 1) - 2.5 points possible

Add to the list135 class a method called sorted that returns a sorted list135 that begins at self. Because we want to maintain persistence, the current version of the list135 should not be altered.

The easiest way to do this is to build a pyhton list from the list135, sort that list with python's sort method, and then build a list135 from the result. Note: lst.sort() sorts lst in-place, and lst.append(x) appends x to the end of lst.

You will do this problem in two parts. In this first part, write sorted so that it returns just a sorted python list.

```
class list135:

    def __init__(self, first_item = None, rest_of_list = None):
        self._first_item = first_item
        self._rest_of_list = rest_of_list

    def cons(self, first_item):
        return list135(first_item, self)

    def first(self):
        return self._first_item

    def rest(self):
        return self._rest_of_list

    def empty(self):
```

```
        return self._rest_of_list == None

    def __str__(self):
        result = "["
        cur = self
        if cur._rest_of_list != None:
            result = result + str(cur._first_item)
            cur = cur._rest_of_list
        while cur._rest_of_list != None:
            result = result + "," + str(cur._first_item)
            cur = cur._rest_of_list
        return result + "]"

    # return a sorted python list of the elements in the list135
    def sorted(self):
                pass
```

4. Question 3 - `list135` sorted (Part 2) - 2.5 points possible

Add to the list135 class a method called sorted that returns a sorted list135 version of the list that begins at self. Because we want to maintain persistence, the current version of the list135 should not be altered.

The easiest way to do this is to build a pyhton list from the list135, sort that list with python's sort method, and then build a list135 from the result. Note: lst.sort() sorts lst in-place, and lst.append(x) appends x to the end of lst.

Building off your solution from the previous question, copy your method from the previous Question and now finish sorted so that it returns the sorted list135.

```
class list135:

    def __init__(self, first_item = None, rest_of_list = None):
        self._first_item = first_item
        self._rest_of_list = rest_of_list

    def cons(self, first_item):
        return list135(first_item, self)

    def first(self):
        return self._first_item
```

```
    def rest(self):
        return self._rest_of_list

    def empty(self):
        return self._rest_of_list == None

    def __str__(self):
        result = "["
        cur = self
        if cur._rest_of_list != None:
            result = result + str(cur._first_item)
            cur = cur._rest_of_list
        while cur._rest_of_list != None:
            result = result + "," + str(cur._first_item)
            cur = cur._rest_of_list
        return result + "]"

    # return a sorted list135 version of the elements in this list135
    def sorted(self):
                pass
```

5. Question 4 - `hamt reduce` - 5 points possible

Add to the hamt class a method called reduce that behaves similarly to reduce on lists. It should take an initial value 'init' and a two-input function 'f(acc, value)' and accumulate the result of applying f to each value in the tree along with the an accumulator that gets updated with each call.

For example if 1, 2, 3, 4 were the values in hamt x, then x.reduce(f, 0) should evaluate to f(f(f(f(0, 1), 2), 3), 4). You may assume that the operation f is associative and commutative, so the order of visiting the values does not matter.

An example use of such a reduce function would be to count elements (ie, like the len function from homework). x.reduce(lambda a, v: a + 1, 0) would start with 0 and add one for each value visited.

Hint: if "acc" holds a value after reducing one of your subtrees, acc = (other child).reduce(f, acc) will update acc with the reduction of the next child's tree.

```
# Hash Array Mapped Trie - Used in CSC 135, Sacramento State
# Written by Ted Krovetz, February 2022
#
```

```
# This implementation assumes that the objects pointed at by the key and
value
# references stored in the HAMT structure do not change during the lifetime
# of the structure.
class hamt:

    DEG = 4       # Children per node (must be power of 2)
    BITS = 2      # log2(DEG), bits needed to select child
    MASK = 0b11   # Bitmask for extracting low BITS bits (DEG - 1)

    def __init__(self, key, value, children=None):
        self._key = key
        self._value = value
        if children == None:
            self._children = [None] * hamt.DEG
        else:
            self._children = children

    def _set(self, key, value, hashbits):
        # Each node encountered during search will get altered.
        # To maintain persistence, each is duplicated, updated, returned.
        if (self._key == key):
            # This node matches key. Return duplicate with new value
            return hamt(self._key, value, self._children)
        else:
            # Continue search using hashbits to pick direction
            child_num = hashbits & hamt.MASK
            # Copy can reuse key/value. Child list gets updated, so duplicate
            copy = hamt(self._key, self._value, list(self._children))
            if (copy._children[child_num] == None):
                # End of search, key not found, add new node
                copy._children[child_num] = hamt(key, value)
            else:
                # Continue by asking appropriate child to set key/value
                copy._children[child_num] = copy._children[child_num].     \
                                    _set(key, value, hashbits >> hamt.BITS)
            return copy

    def set(self, key, value):
        # Pass key/value and hashbits to recursive helper
        return self._set(key, value, hash(key))

    def __str__(self):
        s = "[({},{})".format(str(self._key),str(self._value))
        for i in range(hamt.DEG):
            if (self._children[i] == None):
                s = s + "X"
            else:
                s = s + str(self._children[i])
```

```
        return s + "]"

    # Reduce the tree rooted here with initial value "init"
    def reduce(self, f, init):
                pass
```