Preparing for Homework 6

Create an empty repostitory on Github.

Add `pa6.py` containing your definitions of Github repository with your changes, and upload your repository to Gradescope.

The goals of this assignment are to give you practice with trees and functional programming, and to give you further experience with recent topics such as recursion.

Problem 1: How many ways to make change?

Given a total amount and a set of coin denominations, there can be several different ways of making change.

For example, 10 cents can be made with

1. one 10 cent piece
2. two 5 cent pieces
3. one 5 cent piece and 5 1-cent pieces , or
4. ten 1-cent pieces.

These represent four distinct sets of coins that add up to 10 cents.

For coins of values `[1, 5, 10, 25, 100]`, (this can be a constant in your function, it does not need to be an argument), write a python function called `make_change(total)` that returns a list of all of the distinct combinations of coins that add up to the total amount. The length of the list should be the number of distinct groups of coins.

Problem 2: Dictionary Filter

Write a function, `dict_filter()`, that takes in a function and a dictionary and produces a new dictionary where a given key and value remain associated with each other in the new dictionary, if and only if the function returns `True` when called with the key and the value. Other entries from the original dictionary are not placed in the new one.

For example, given the input:

```
example = {"Illinois": "IL", "Pennsylvania": "PA", "Indiana": "IN"}
```

and the function:

```
def checker(name, abbrev):
    return abbrev[0] == "I" and name[1] == "l"
```

the result of `dict_filter(checker, example)` should be:

```
{'Illinois': 'IL'}
```

Problem 3: Tree Map

Here is a class for representing trees, similar to the one we saw in class that had the names of countries, states, or cities, and their populations, and a sample tree made with it:

```
class KVTree:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.children = []
    def add_child(self, child):
        self.children.append(child)
samplekv = KVTree("us", 4.6)
pa = KVTree("pa", 1.9)
samplekv.add_child(pa)
pa.add_child(KVTree("Pittsburgh", 0.3))
pa.add_child(KVTree("Philadelphia", 1.6))
il = KVTree("il", 2.7)
samplekv.add_child(il)
il.add_child(KVTree("Chicago", 2.7))
```

Write the function `treemap()` that takes in a function and one of these trees and modifies the tree according to the function. In particular, the `treemap()` function should traverse the entire tree, and call the provided function once per node in the tree. The provided function should take in two parameters, in order, the key and value of the current node, and return a tuple with two elements, in order, the new key and new value for that node. The `treemap()` function should then replace the contents of the current node with those updated values. For instance, applying the following function, via `treemap()`, to the example tree earlier:

```
lambda x, y: (x.upper(), y * 1000000)
```

should modify the tree to have all the names in upper-case and to have the populations in people rather than in millions of people.

Problem 4: Trees Modeling Decisions

We will make a different tree class, which we will call `DTree`, which represents a "flow chart" for a decision someone might make. For instance, will I go for a walk today? The decision might be based on the temperature, humidity, and wind speed. My tolerances may be different depending on the overall details; I may not care about humidity unless it is also warm, or about wind unless it is also cold, etc.

We will represent the kind of information used to make a decision in a tuple. For instance, if I care about temperature, humidity, and wind speed, we could place these into a 3-tuple in that order.

Each of the nodes in the tree represents a step in my decision-making process. Some nodes represent a situation where the outcome still depends upon one or more remaining questions. For these, I will next look at a specific part of the tuple, such as the temperature. I will compare it to a threshold. If it is less than or equal to the threshold, I will then go to a specific child of that node to continue the process; otherwise, I will go to the other child of that node. Eventually, I will get to another type of node, that represents my final answer. For these nodes, there are no thresholds or children, just a copy of the answer, like "yes, I will go for a walk" itself.

Create a class, `DTree.` The constructor should take in, in order:

1. `variable`, the index into the tuple with data about, for instance, the weather, that will be inspected at this step in the process (e.g., perhaps 0 = temperature, 1 = humidity, and 2 = wind speed)
2. `threshold`, a number, the boundary between points at which I will proceed in one way vs. another
3. `lessequal`, another `DTree` that indicates how to continue if the actual data point for that variable is less than or equal to the threshold
4. `greater`, a subtree for when the datapoint is greater
5. `outcome`: when I am ready to make a decision, the answer goes here, and the other inputs are `None`.

In your constructor, store all these parameters in object attributes. Raise a `ValueError` if the following is not true: either all four of the first four arguments are not `None`, or the last argument (`outcome`) is not `None`, but not both.

As an example, supposing tuples provide temperature, followed by humidity, followed by wind speed, here is a tree that captures the idea that I will go for a walk so long as the temperature is at most 66 degrees:

```
DTree(0, 66, DTree(None, None, None, None, "walk"),
      DTree(None, None, None, None, "stay home"),
      None)
```

and here is one that only permits walks when it is at most 66 but the winds are not above 10mph:

```
DTree(0, 66,
      DTree(2, 10,
            DTree(None, None, None, None, "walk"),
            DTree(None, None, None, None, "stay home"),
            None),
      DTree(None, None, None, None, "stay home"),
      None)
```

Write the following methods in this class:

1. `tuple_atleast`: analyzes the tree and determines how many entries there need to be in the tuple. For instance, given the second example tree, we can infer that we need tuples of at least size 3, because there are references to variable 0 and variable 2. It is possible that the tuples were intended to have more entries, but if we don't look at all of them (as the example trees do not – the first only looks at temperature and the second ignores humidity – then this could be imprecise. But we know the tuple must be at least this size.
2. `find_outcome`: takes in a tuple with observations and navigates through the tree to provide the outcome that matches (like "walk")
3. `no_repeats`: analyzes the tree and returns `True` if and only if there are not "repeats", `False` otherwise. We define a *repeat* as a situation in which the tree may ask a question about the same variable twice. For instance, if the tree checks the temperature, and then a descendant node within part of the tree checks the temperature (even with a different threshold) a second time, that is a repeat. Note that it is okay for the tree to check the humidity in the branch of the tree that deals with cold temperatures and also in the branch of the tree that deals with warm temperatures; this is not a repeat. But it will be considered a repeat if it would check the temperature twice for a single datapoint.

To implement `no_repeats`, it would be helpful to have a parameter being passed between recursive calls that tracks which variables we have or have not seen yet as we explore. But, the method is not intended to take any parameters beyond `self.` This is a common situation in recursion: we often want extra parameters to track something, but from the perspective of someone else using the function, they should not need to know about those parameters and what they do. The common solution to this is to use a helper. Your `no_repeats` method must take no extra parameters, but it can immediately call a helper method that is actually recursive and that has the desired additional parameter(s). It can call that helper with an appropriate starting value for that parameter, and return the result of the helper as its own result.