Class versus Static Methods

Question 1: Referring to Class Variables

In Homework 7, you used a class variable to keep track of the next index to use, so that all objects are numbered from 0 upwards. To refer to the class variable inside of an object method, you can preface it with the class name, as the following code does. This code also has a second class variable to keep track of all of the instances.

```python
class A:
    next_index = 0
    all_indices = []
    def __init__(self,a,b):
        self.a = a
        self.b = b
        self.index = A.next_index
        A.all_indices.append(self.index)
        A.next_index += 1

x = A(1,2)
y = A(2,3)
print(x.index)
print(y.index)
print(A.next_index)
print(A.all_indices)
```

However, as we will see in later in this homework, you might not want to hardcode in the name of the class. So, you might be tempted to do the following, which actually runs but does not give the right answer.

```python
class A:
    next_index = 0
    all_indices = []
    def __init__(self,a,b):
        self.a = a
        self.b = b
        self.index = self.next_index
        self.all_indices.append(self.index)
        self.next_index += 1

x = A(1,2)
y = A(2,3)
print(x.index)
print(y.index)
print(A.next_index)
print(A.all_indices)
```

Explain how the programmer thought they could refer to the class name.
Explain what actually happens in the code.
Question 2: Referring to the Class

If you do not want to hard code the name of the class in your object methods, you can use `self.__class__`. Rewrite the code from the previous question so that it does not refer to class variables using `self`, but always uses `self.__class__`.

Hand in your code.

Question 3: Class Methods

`hw8_dependencies.txt` is instructions for installing a TimeView, a program that my previous TA worked on with Professor Kain. It lays out of the dependencies between the parts. Consider the following part of the file:

```
black
  - appdirs
  - attrs
  - click
  - toml
```

This says that ‘black’ python library depends on the appdirs, attrs, click, and toml python libraries. So, before working on ‘black’, it should first finish appdirs, attrs, click, and toml.

Review the code in `hw8.py`. This code reads in a list of file dependencies. `hw8.py` has the `Dependency` class, with static method `LoadFile` that will read in the text file. For each program/module read in, it will call `CreateIfNecessary`, which will check if an object has already been created for the program/module, and if not, `CreateIfNecessary` will call the initializer.

`LoadFile` also keeps track of the children of each object, and after it has read in all of the children, will update the object’s attribute `children` to a list of all of its children’s objects.

The problem with this code is that it is using static methods when it would be better to use class methods. By using a class method, the `Dependency` class name does not need to be hardcoded in the methods. Also, change the initializer so that it does not need the class name hardcoded. When you are done, the only type `Dependency` should occur is when it is declared, and from the calls to `Dependency.LoadFile` and `Dependency.PrintAll`.

Hand in your code.

Question 4: Inheritance

Now that you have changed the `Dependency` class so that it uses class methods, let’s not change it any further. It might be that that class is used by a number of different programs, to provide basic functionality for reading in a dependency list.

In the next question, you will be doing a topological sort on the dependency data. But first, we need to convert it into an adjacency list, with vertices numbered from 0. To do this, we will not modify `Dependency`, but create a new class called `MyDependency` that will inherit `tt Dependency`.

`MyDependency`’s initializer should call `Dependency`’s initializer using the ‘super()’ function. Similar to the last homework, have the initialize number all of the objects starting at 0 as they are created, using the class variable `next_index` and create a class variable to map the indices to the objects `index_to_obj`. Remember, do not modify the code in `Dependency`; only put this code in the `MyDependency` class.
Also, MyDependency should have its own version of LoadFile. It should first call LoadFile from Dependency, and then create an object attribute called children_indices that parallels the children attribute, but is a list of the children’s indices instead.

Finally, add a new class method called create_adjacency_list that returns a list of adjacency lists. Note that this adjacency list has an edge (u,v) if u depends on v, in order words if v must come before u. This is actually opposite of how the textbook defines the dependency relationship. But, we can overcome this by not taking the reversal of the finish times from DFS.

Turn in your code.

**Question 4a**

When the LoadFile class method in Dependency calls the class initializer, which one is it calling: the one in Dependency or in MyDependency? Explain why this is the case. Explain why this is a good design decision.

**Topological Sort**

**Question 5: DFS**

Staying pretty close to the textbook code, write a DFS and DFS_VISIT functions. These functions should not know anything about the Dependency or MyDependency classes. Do not use global variables for passing information between DFS and DFS_VISIT. For my version, I used separate lists for keeping track of discovery time, finish time, color and pi, which I passed between them.

DFS should return a list of the discovery times, finish times, and pi.

Turn in your code.

**Question 6: Topological Sort**

Create a function called FindOrder that reads in the dependencies, creates an adjacency list, calls DFS, and then uses the finish times to do a topological sort. Note that TimeView should be the last item returned.

Turn in your code.

**Web Resource: Trie Trees**

These questions will continue the trie tree that we did in Homework 7. You must start with the solution given in homework 7 for the methods for __init__, add_child, find and walk.
Question 7: Print Tree

To help with debugging your trie tree code, make a method `print_tree` that prints the trie tree showing all of the values at each node. Each node should follow its parent, but indented the length of its parent’s value. Below is a sample for the tree in the figure.

```
r
om
 an
e
us
 ulus
ub
 e
 ns
  r
ic
 on
 undus
```

Hand in the code.

Question 8: Split Node

To add a new word to a trie tree, you might need to split a prefix. For example, in the figure, if you want to add the word ‘robust’, the ‘om’ node needs to be split into a ‘o’ and ‘m’ node so that ‘bust’ can be added to ‘o’. Here you will make the routine to take the first step, which is to split a node.

Make a method called `split_node` that will take a node in a trie tree and a proper prefix of the node’s character sequence, it will split the node into a parent and a child node with the parent having the given prefix, and the child having the remainder.

For the subsequent questions, it will be easier if you make created node to have the remainder of the string, rather than the prefix. For the example above, in which we are splitting ‘om’, change the ‘om’ node to be just ‘o’, and have the new node have value ‘m’. This way, the node that the method is called on does not change with respect to nodes closer to the root.

Here is some sample code to show how split node should work, and its output.
def test_split_node():
    top = Node"
    top.add_child(chars("roman"))
    top.find("roman").add_child(chars("e"))
    top.find("roman").add_child(chars("us"))
    top.print_tree()
    top.find("roman").split_node("rom")
    top.print_tree()
    top.find("rom").add_child(chars("e"))
    top.print_tree()

roman
    e
    us
rom
    an
    e
    us
rom
    an
    e
    us
    e

Hand in the code.

**Question 9: Find Partial**

Make a method called `find_partial` that, when given the top node of a trie tree and a word, will search for the word in the trie tree. It should return a tuple of a node and a string.

If it does not find the word in the tree, it should return (a) the node that covers as much of the prefix as is in the trie tree, and (b) as well as the suffix of the word that is not in the tree. If the word is in the tree, it should return the find node, just like `find`, but also the empty string (so we know it was successful). Make this routine as similar to `find` as possible.

For example, in the sample tree, if you give it the word ‘robust’ should return the ‘r’ node, and the string ‘obust’. If you give it ‘romain’, it should return the ‘om’ node along with ‘ain’.

Hand in the code.