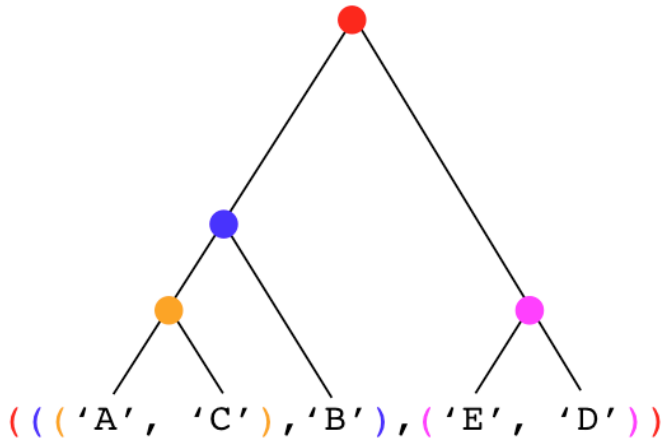
Exam project for Applied Programming 2010: Comparing Trees.

Tree data structures are used in biology to represent evolutionary relationships between species. More often than not the evolutionary relationship of a set of species is ambiguous resulting in different trees depending on what DNA locus that is used for the analysis.

In this exam project you will work on trees represented as nested tuples. The figure below shows the correspondence between tuples and *nodes* in the tree.



The blue tuple represents the *sub-tree* below the blue node. The elements of the blue tuple are the *descendants* of the blue node. Of these descendants, (‘A’, ‘C’) is a tuple and represents a sub-tree whereas ‘B’ is a string and represents a *leaf* denoting the species ‘B’.

The set of leaves of a tree or sub-tree is referred to as a clade. ((‘A’, ‘C’), ‘B’) is a sub-tree and the set of leaves ‘A’, ‘C, ‘B’ is the corresponding clade. Only the leaves of trees or sub-trees are considered clades, the leaf of a singleton is not. E.g. for the sub-tree below the blue node ‘A’, ‘C’, ‘B‘ and ‘A’, ‘C’ are clades but ‘B’ is not.

One goal of this project is to evaluate how different two trees are using a metric called the Robinson-Foulds distance. This distance is calculated here as the number of clades that are unique to either of the two trees.

Trees describing the same set of species may differ if built on different genomic loci. We are thus often faced with a long list of more of less different trees. Such trees may differ completely or may differ only in the relationship of a few species. Another goal of this project is to evaluate to what extent many trees agree with respect to the placement of a given species.

Problem 1:
Before you start to compare trees you may have to check that the trees you want to compare represent the same set of species. To know what species a tree represents you must extract the set of leaves. This can be done recursively in the following manner:

Function: `getLeaves`
Argument: a `tree` in the form a tuple.
Variables: let `descendant` be an element of the tuple `tree` i.e. one of the leaves or sub-trees that the tree immediately divides into.
Returns: a list of all leaves in `tree`.

Base case: if `descendent` is a leaf it is added to the list of leaves in `tree`.

Recursive case: if `descendent` is a sub-tree then `getLeaves` must be called to return the leaves of `descendant`. The leaves returned by this call to the function are then added to the list of leaves in `tree`.

Example usage:
```
getLeaves( ((('A','C'),'B'),('E','D')) )
```
should return
```
['A','C','B','E','D']
```

Hint: the type of a variable `x` is returned by the built in function `type(x)`. The statement, `type(x) is str`, returns `True` if `x` is a string.

Hint: remember that the list methods `append` and `extend` do different things.

Implement the function `getLeaves`.

Problem 2:
Before you can calculate the Robinson-Foulds distance between two trees you need to extract all the clades of each tree in order to compare them. E.g. the tree in the figure above has the following four clades:

‘A’, ‘C, ‘B’, ‘E’, ‘D’
‘A’, ‘C, ‘B’
‘A’, ‘C
‘E’, ‘D’

This problem can be solved recursively in almost exactly the same way you solved problem one. The only difference is that in addition to returning the list of leaves of the given tree the function must also add this list to a list given as argument to the function.

Write a function

```
def getListOfClades(tree, listOfClades):
```

taking two arguments: a tuple `tree`, and a list `listOfClades`. To work recursively the function must return a list of the leaves in `tree` just like `getLeaves` did. The lists of leaves (the clades) must be collected in `listOfClades`.

In the initial call to `getListOfClades` `listOfClades` is empty. In each recursive call to the function `listOfClades` is passed on as argument allowing it to collect a clade each time it is called.

Example usage:

```
listOfClades = []
getListOfClades( (('A','C'),'B'),('E','D')), listOfClades)
```

`listOfClades` should now be
`[['A','C'], ['A','C','B'], ['E','D'], ['A','C','B','E','D']]`

Hint: remember that the value of a list variable is a reference to the list.

Problem 3:
Having obtained a list of clades for each of two trees we need to compare the lists to compute the Robinson-Foulds distance. Two clades are identical if they contain the same set of leaves. Two lists with the same set of leaves, however, may be sorted differently. So to make comparison easier you need to sort all the lists representing clades.

Write a function

```
def sortLists(listOfLists):
```

taking a list of lists, `listOfLists`, as argument. The function returns nothing. After calling the function the lists in `listOfLists` should be sorted.

Example usage:

```
L = [['A','C'],['A','C','B'],['E','D'],['A','C','B','E','D']]
sortLists(L)
```

`L` is now:
`[['A','C'],['A','B','C'],['D','E'],['A','B','C','D','E']]`

Hint: remember that the value of a list variable is a reference to the list.

To make comparison of clades even easier, as you will see later, you also need a function that turns a list of lists into a list of tuples.

Write a function

```
def listsToTuples(listOfLists):
```

taking a list of lists, `listOfLists`, as argument. The function must return a list of tuples.

- Example usage:
- listsToTuples([['A','C'],['A','B','C'],
- ['D','E'],['A','B','C','D','E']])
-
- should return
- [('A','C'),('A','B','C'),('D','E'),('A','B','C','D','E')]
-

Problem 4:
With the functions you have implemented so far you can compute a list of clades represented as sorted tuples. To find the number of clades that are unique to either tree you now need to implement a function that compares two such lists of sorted tuples. A naive solution would be:

```
def findDifferences(list1, list2):
    cladesNotShared = 0
    for clade in list1:
        if clade not in list2:
            cladesNotShared += 1
    for clade in list2:
        if clade not in list1:
            cladesNotShared += 1
    return cladesNotShared
```

This solution works but is slow if the lists are long because testing for membership of a list requires that Python runs through the list. You can do better. If you put `list1` and `list2` together, clades in this new list `bigList` appear either once or twice. If you find the number of different clades in `bigList` you can then find the number of clades that are not shared between the trees in the following manner:

```
nrShared = len(bigList) - countNrDiffClades(bigList)
nrCladesNotShared = len(bigList) - 2 * nrShared
```

Write a function

```
def countNrDiffClades(bigList):
```

that takes a list of sorted tuples `bigList` as argument and returns the number of different tuples (clades) in the list. To compute this number effectively the function is required to make use of a dictionary.

Hint: tuples are valid keys in a dictionary.

Example usage:
`countNrDiffClades([('A','B'),('C','D'),('C','D')])`
should return `2`

Write a function

```
def findDifferences(list1, list2):
```

that takes two lists of tuples as arguments, `list1` and `list2`, and returns the number of tuples (clades) not shared. The two lines of code at the bottom of the previous page make up most of the function.

Example usage:

```
findDifferences( [('A','C'),('A','B','C'),('D','E'),
                  ('A','B','C','D','E')],
                 [('B','C'),('A','B','C'),('D','E'),
                  ('A','B','C','D','E')] )
```

should return `2`

Problem 5:
You are now all set to compute the Robinson-Foulds distance between two trees.

Write a function

```
def rfDistance(tree1, tree2):
```

that takes two trees with the same sets of leaves as arguments and returns the Robinson-Foulds distance between the two trees. Use the functions you have implemented already to do the required computations: compute the list of clades for each tree and turn them into sorted lists of tuples before you compare them to find the differences. The number of clades not shared is your Robinson-Foulds distance.

Example usage:

```
tree1 = ((('A','C'),'B'),('E','D'))
tree2 = ((('A','B'),'C'),('E','D'))
rfDistance(tree1, tree2)
```

should return 2

Problem 6:

Now on to a different issue. Imagine you have have built, say, 100 trees for a set of species using sequences from 100 different genome loci. Due to the stochasticity in the evolution of DNA sequences the true phylogenetic relationship may not be reflected perfectly in the sequences from a locus. Depending on the amount of sequence information available at each locus, to resolve the evolutionary relationship of the species, the 100 trees may be identical or quite different.

The position of a species in a tree is summarized by the clades it is part of. E.g. in the figure species 'A' occurs in the clades ('A', 'C, 'B', 'E', 'D'), ('A', 'C, 'B'), ('A', 'C).

You are more confident in phylogenetic placements that many trees agree on. So you want to know if the clades that a species fall in are almost the same across all trees or if they differ a lot from tree to tree?

Write a function

```
def printCladeFrequencies(treeList, leaf):
```

that takes a list of trees `treeList` as first argument and a string `leaf` as second argument. The function should not return anything. It must compute and print the clades that `leaf` is part of along with the percentage of trees in `treeList` where this clade is found. `leaf` is always part of the clade that represent the entire tree. This is trivial and this clade should not be printed.

Example usage:

```
tree1 = ((('A','C'),'B'),('E','D'))
tree2 = ((('A','B'),'C'),('E','D'))
printCladeFrequencies([tree1, tree2], 'A')
```

should print:

```
('A','B','C') 100%
('A','C')     50%
('A','B')     50%
```

Hint: use the functions you have already implemented to compute lists of clades (tuples)for each tree. Use a dictionary to map each clade (a tuple) to the number of

times it occurs across all the trees in `treeList`. Remove the clade representing the largest tree, and finally print the clades with the their frequencies.