TREES, ITERATORS, AND GENERATORS

COMPUTER SCIENCE MENTORS CS 88

April 19th to April 24th

1 Trees

For the following problems, use this definition for the Tree class:

```
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

Notice that with this implementation we can mutate a tree using attribute assignment:

```
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.label = 5
>>> t.label
5
```

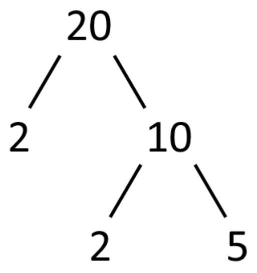
1. What would Python display? If you believe an expression evaluates to a *Tree* object, write *Tree*.

```
>>> t0 = Tree(0)
>>> t0.label


>>> t0.branches


>>> t1 = Tree(0, [1, 2])#Is this a valid tree?

>>> t2 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
>>> t2.branches[0]

>>> t2.branches[1].branches[0].label
```

2. Define a function `square_tree(t)` that squares every value in the non-empty tree `t`. You can assume that every value is a number.

```
def square_tree(t):
    """Mutates a Tree t by squaring all its elements."""
```

3. Define the function `factor_tree` which takes in a positive integer `n` and returns a factor tree for `n`. In a factor tree, multiplying the leaves together is the prime factorization of the root, `n`. See below for an example of a factor tree for `n = 20`.



```
def factor_tree(n):
     """
     >>> factor_tree(20)
     Tree(20, [Tree(2), Tree(10, [Tree(2), Tree(5)])])
     >>> factor_tree(1)
     Tree(1)

    for i in ______________________:

        if _________________________:

            return Tree(_____, ______________________________)

    ________________________________
```

4. Write a function that returns true only if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

```
def contains_n(elem, n, t):
    """
    >>> t1 = Tree(1, [Tree(1, [Tree(2)])])
    >>> contains(1, 2, t1)
    True
    >>> contains(2, 2, t1)
    False
    >>> contains(2, 1, t1)
    True
    >>> t2 = Tree(1, [Tree(2), Tree(1, [Tree(1), Tree(2)])])
    >>> contains(1, 3, t2)
    True
    >>> contains(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif ____________________________________________:

        return ______________________________________

    elif ____________________________________________:

        return ______________________________________

    else:

        return ______________________________________
```

2 Iterators

Introduction to Iterators

An iterable is a data type that contains a collection of values which can be processed one by one in order. Some examples are lists, tuples, strings, and dictionaries.

How do we iterate over an iterable? We use another type of object called an iterator. We create an iterator for an iterable by calling iter on the iterable. It will then keep track of its position in the iterable. Calling next on the iterator gives the current value in the iterable and move the iterator forward until it has gone past the end of iterable and a StopIteration error is produced.

You might wonder why this looks so similar to for loops. As a matter of fact, the for loop uses an iterator. For any iterable you give it, the for loop implicitly creates an iterator to go through its elements.

```
>>> a = [1, 2]
>>> a_iter = iter(a)
>>> next(a_iter)
1
>>> next(a_iter)
2
>>> next(a_iter)
StopIteration Error
```

Related functions:

`range(start, end)` returns an iterable

`map(f, iterable)` returns an iterator containing the values resulting from applying f to every element in the iterable

5. What would Python display? If an error occurs, write ”Error”. If a function is displayed, write ”Function”. If nothing is returned, write ”Nothing”. It might be easier to draw the lists with the iterator as an arrow pointing to the location in the list

```
>>> lst = ['c', 's', 8, 8]
>>> next(lst)


>>> lst_iter = iter(lst)
>>> next(lst_iter)

>>> next(lst_iter)

>>> next(iter(lst))

>>> [x for x in lst_iter]
```

3 Generators

Introduction to Generators

A generator function is a special type of function that uses a yield statement instead of a return statement. When a generator function is called, it returns an iterator. To the right is an example of a generator function that creates an iterator for all the integers 0, 1, 2, 3, 4, 5, ...

The yield statement is like the return statement, except that yield causes the current frame to be saved until next is called again. return simply closes the frame, as we have always seen.

Including a yield statement in a function automatically makes a function a generator function. When the function is first called, it returns a generator object instead of executing the code. But when next is called on the generator, the code is executed until the next yield

```
>>> def gen_nums():
...   current = 0
...   while True:
...     yield current
...     current += 1
>>> gen = gen_nums()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

6. What does the following code block output?

```
def foo():
    a = 0
    if a < 10:
        print("Hello")
        yield a
        print("World")

for i in foo():
    print(i)
```

7. How can we modify `foo` so that it satisfies the following doctests?

```
>>> a = list(foo())
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```