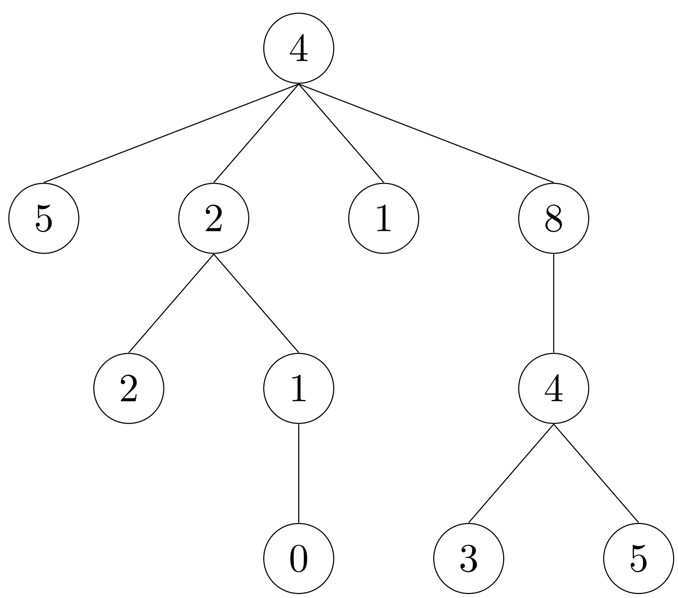
FUNCTION-BASED TREES, MUTABILITY, ITERATORS + GENERATORS

COMPUTER SCIENCE MENTORS 61A

February 19–February 23, 2023

1 Tree ADT

Trees are a kind of recursive data structure. Each tree has a **root label** (which is some value) and a sequence of **branches**. Trees are "recursive" because the branches of a tree are trees themselves! A typical tree might look something like this:



This tree's root label is 4, and it has 4 branches, each of which is a smaller tree. The 6 of the tree's **subtrees** are also **leaves**, which are trees that have no branches.

Trees may also be viewed **relationally**, as a network of nodes with parent-child relationships. Under this scheme, each circle in the tree diagram above is a node. Every non-root node has one parent above it and every non-leaf node has at least one child below it.

Trees are represented by an abstract data type with a `tree` constructor and `label` and `branches` selectors. The `tree` constructor takes in a label and a list of branches and returns a tree. Here's how one would construct the tree shown above with `tree`:

Created by Alyssa Smith, Esther Shen

```
tree(4,
    [tree(5),
     tree(2,
        [tree(2),
         tree(1,
            [tree(0)])]),
     tree(1),
     tree(8,
        [tree(4,
            [tree(3), tree(5)])])])
```

The implementation of the ADT is provided here, but you shouldn't have to worry about this too much. (Remember the abstraction barrier!)

```python
def tree(label, branches=[]):
        return [label] + list(branches)

def label(tree):
        return tree[0]

def branches(tree):
        return tree[1:] # returns a list of branches
```

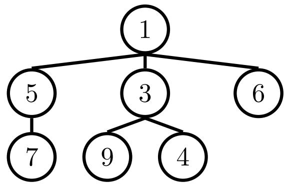
Because trees are recursive data structures, recursion tends to a be a very natural way of solving problems that involve trees.

- The **recursive case** for tree problems often involves recursive calls on the branches of a tree.
- The **base case** is often reached when we hit a leaf because there are no more branches to recurse on.

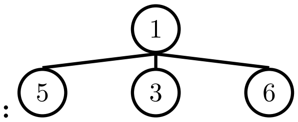
1. Draw the tree that is created by the following statement:

```
tree(4,
     [tree(5, []),
      tree(2,
          [tree(2, []),
           tree(1, [])]),
      tree(1, []),
      tree(8,
          [tree(4, [])])])
```

BEFORE prune(1):



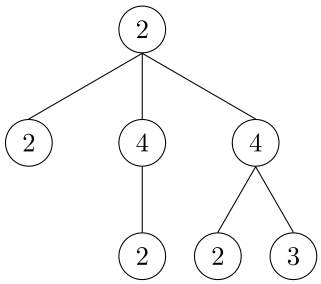
AFTER prune(1):



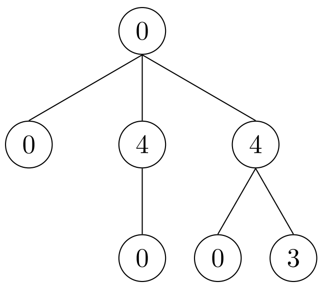
2. Implement `prune`, which takes in a tree `t` and a depth `k`, and should return a new tree that is a copy of only the first `k` levels of `t`. Suppose `t` is the tree shown to the right. Then `prune(t, 1)` returns nodes up to a depth of level 1.

3. Write a function, `replace_x` that takes in a tree, `t`, and returns a new tree with all labels `x` replaced with 0.

   For example, if we called `replace_x(t, 2)` on the following tree:



   We would expect it to return

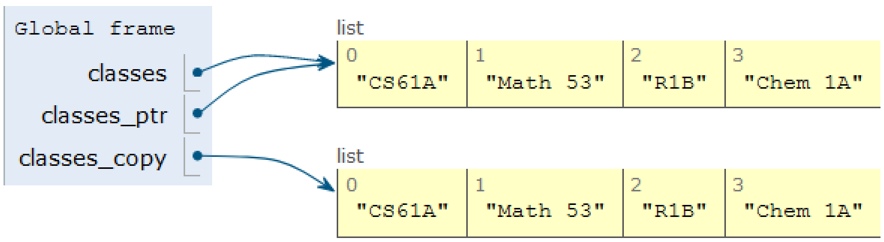


```
def replace_x(t, x):
    """
    >>> t = tree(2, [tree(1), tree(2)])
    >>> replace_x(t, 2)
    tree(0, [tree(1), tree(0)])
    """
    new_branches = []
    for _______ in ______________________________:

        new_branches.________________________________

    if __________________________________________:

        return ___________________________________

    return ___________________________________
```

2 Mutability

Let's imagine it's your first year at Cal, and you have signed up for your first classes!

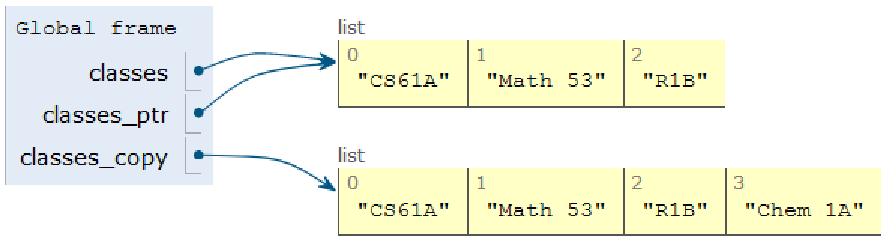
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



After a few weeks, you realize that you cannot keep up with the workload and you need to drop a class. You've chosen to drop Chem 1A. Based on what we know so far, to change our classes list, we would have to create a new list with all the same elements as the original list except for Chem 1A. But that is silly, since all we really need to do is remove the Chem 1A element from our list.

We can fix this issue with list mutation. In Python, some objects, such as lists and dictionaries, are mutable, meaning that their contents or state can be changed over the course of program execution. Other objects, such as numeric types, tuples, and strings are immutable, meaning they cannot be changed once they are created. Therefore, instead of creating a new list, we can just call `classes.pop()`, which removes the last element from the list.

```
>>> classes.pop() # pop returns whatever item it removed
"Chem 1A"
```



Here are more list methods that mutate:

Mutability in Lists

Function	Create or Mutate	Action/Return Value
lst.**append**(element)	mutate	attaches element to end of the list and returns None
lst.**extend**(iterable)	mutate	attaches each element in iterable to end of the list and returns None
lst.**pop**()	mutate	removes last element from the list and returns it
lst.**pop**(index)	mutate	removes element at index and returns it
lst.**remove**(element)	mutate	removes element from the list and returns None
lst.**insert**(index, element)	mutate	inserts element at index and pushes rest of elements down and returns None
lst += lst2	mutates	attaches lst2 to the end of lst and returns None same as lst.extend(lst2)
lst[start:end:step size]	create	creates a new list that start to stop (exclusive) with step size and returns it
lst = lst2 + [1, 2]	create	creates a new list with elements from lst2 and [1, 2] and returns it
list(iterable)	create	creates new list with elements of iterable and returns it

(credits: Mihira Patel)

1. What would Python display? If an error occurs, write ”Error”. If a function is displayed, write ”Function”. If nothing is returned, write ”Nothing”.

```
>>> a = [1, 2]
>>> a.append([3, 4])
>>> a
```

```
>>> b = list(a)
>>> a[0] = 5
>>> a[2][0] = 6
>>> b
```

```
>>> a.extend([7])
>>> a += [8]
>>> a += 9
```

```
>>> a
```

Challenge:

```
>>> b[2][1] = a[2:]
>>> a[2][1][0][0]
```

2. Given some list `lst` of numbers, mutate `lst` to have the accumulated sum of all elements so far in the list. If `lst` is a deep list, mutate it to similarly reflect the accumulated sum of all elements so far in the nested list. Your function should return an integer representing your "accumulated" sum (sum of all numbers in your list). You may not need all lines provided.

   Hint: The **isinstance** function returns True for **isinstance**(l, **list**) if `l` is a list and False otherwise.

```
def accumulate(lst):
    """
    >>> l = [1, 5, 13, 4]
    >>> accumulate(l)
    23
    >>> l
    [1, 6, 19, 23]
    >>> deep_l = [3, 7, [2, 5, 6], 9]
    >>> accumulate(deep_l)
    32
    >>> deep_l
    [3, 10, [2, 7, 13], 32]
    """
    sum_so_far = 0
    for ________________________________________:

        ________________________________________

        if isinstance(____________________, list):

            inside = ____________________________

            ____________________________________

        else:
            ____________________________________

            ____________________________________

    _________________________________
```

3 Iterators & Generators

On a conceptual level, **iterables** are simply objects whose elements can be iterated over. Think of an iterable as anything you can use in a `for` loop, such as ranges, lists, strings, or dictionaries.

On a technical level, iterables are a bit more complicated. An **iterator** is an object on which you can (repeatedly) call `next`, which will return the next element of a sequence. For example, if `it` is an iterator representing the sequence $1, 2, 3$, then we could do the following:

```
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
StopIteration
```

`StopIteration` is an exception that is raised when an iterator has no more elements to produce; it's how we know we've reached the end of an iterator. Iterators that will never produce a `StopIteration` exception are called *infinite*.

Under this regime, an iterable is formally defined as an object that can be turned into an iterator by passing it into the `iter` function. When you iterate over an iterable, Python first uses `iter` to create an iterator from the iterable and then iterates over the iterator. The simple `for` loop syntax abstracts away this fact. f There are a few useful functions that act on iterables that are particularly useful:

- `map(f, it)`: Returns an iterator that produces each element of `it` with the function `f` applied to it.
- `filter(pred, it)`: Returns an iterator that includes only the elements of `it` where the predicate function `pred` returns true.
- `reduce(f, it, init)`: Reduces `it` to a single value by repeatedly calling the two-argument function `f` on the elements of `it`: `reduce(add, [1, 2, 3])` → `6`. Optionally, an initializer may be provided: `reduce(add, [1], 5)` → `6`.

Generators, which are a specific type of iterator, are created using the traditional function definition syntax in Python (`def`) with the body of the function containing one or more `yield` statements. When a generator function (a function that has `yield` in the body) is called, it returns a generator object; the body of the function is not executed. Only when we call `next` on the generator object is the body executed until we hit a `yield` statement. The `yield` statement yields the value and pauses the function. `yield from` is another way to yield values. When we `yield from` another iterable, it yields each element from that other iterable one at a time.

The following generators all represent the sequence $1, 2, 3$:

```
def a():
    yield 1
    yield 2
    yield 3
```

```
def b():
    for x in range(1, 4):
        yield x
```

```
def c():
    yield from b()
```

1. Given the following code block, what is output by the lines that follow?

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

>>> foo()
```

```
>>> foo_gen = foo()
>>> next(foo_gen)
```

```
>>> next(foo_gen)
```

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

```
>>> a = iter(filter(lambda x: x % 2, map(lambda x: x - 1, range(10))))
>>> next(a)
```

```
>>> reduce(lambda x, y: x + y, a)
```

2. Define `all_sums`, a generator that iterates through all the possible sums of elements from `lst`. (Repeat sums are permitted.)

```
def all_sums(lst):
    """
    >>> list(all_sums([]))
    [0]
    >>> list(all_sums([1, 2]))
    [3, 2, 1, 0]
    >>> list(all_sums([1, 2, 3]))
    [6, 5, 4, 3, 3, 2, 1, 0]
    >>> list(all_sums([1, 2, 7]))
    [10, 9, 8, 7, 3, 2, 1, 0]
    """
```