FUNCTION-BASED TREES, MUTABILITY, ITERATORS & GENERATORS Meta

COMPUTER SCIENCE MENTORS 61A

October 7 – October 11, 2024

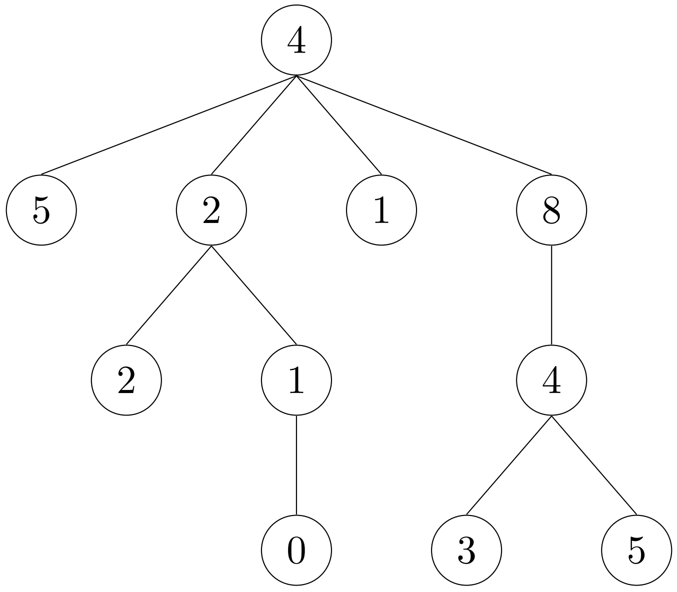
Example Timeline
General: There is a lot of problems in this worksheet - The timings below are suggested times: The minutes given in each numeric header sums up to 50 Mins; The suggested times next to each problem do not – those are a note for you in case you do choose to do that problem.

1. Tree ADT [17 Minutes]
   - Mini-Lecture [5 Mins]
   - Q1 - Prune - Difficulty: Medium [7 Mins]
   - Q2 - Replace X - Difficulty: Medium [10 Mins]
   - Q3 - Contains N - Difficulty Hard, Exam Level [13 Mins]
2. Mutability [17 Minutes]
   - Mini-Lecture [5 Mins]
   - Q1 - WWPD - Difficulty: Easy, challenge is medium [5 Mins]
   - Q2 - Accumulate - Difficulty: Medium [7 Mins]
   - Q3 - Scrabble! - Difficulty: Medium+, Exam Level [10 Mins]
3. Iterators & Generators [15 Minutes]
   - Mini-Lecture [5 Mins]
   - Q1 - WWPD - Difficulty: Easy - Medium [5 Mins]
   - Q2 - All Sums - Difficulty: Medium [5 Mins]

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:

Created by Jim Fang, Bryan Adolfo Ventura Benitez. Worksheet feedback: links.jimfang.dev/feedback
Provided by CSM 61A contributors under CC BY 4.0/3-clause BSD.



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`:

```
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!)

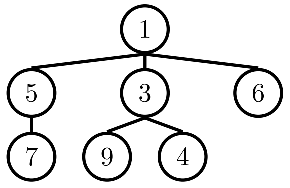
```
def tree(label, branches=[]):
        return [label] + list(branches)

def label(tree):
        return tree[0]

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

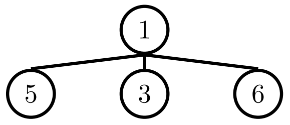
Because trees are recursive data structures, recursion tends to a be a very natural way of solving problems that involve trees.

- 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.

Teaching Tips

- Please make sure to check in with your students before mini-lecture so that you don't go over too much content that that they already feel comfortable with.
- While it is typically true that your make the recursive calls on the branches of a tree and stop recursing when you reach a leaf, this is by no means always true, and you should make it clear that there will be exceptions to this rule of thumb.
- Common Misconceptions:
  - Students often have trouble with the idea that branches is a list of trees. Try to be specific when explaining, focusing on types. (Branches are lists, saving trees in them.)
    - Try using the tree functions to build up different trees.
    - Write out all the functions on the board and clearly define the types of the output and input.
  - Data Abstraction and Trees
    - Although `t[0]` returns the label from the tree, students should be using `label(t)`. This is because `t` is not a list, it is a tree which is a data abstraction!
    - It's important to explain why indexing branches (e.g. `branches(t)[0]`) doesn't violate an abstraction barrier (since branches returns a list of trees).
- The objectives for students are to:
  - Draw trees as graphical representations given Python code
    - Mention to students that empty branches `[]` is the default argument, so `tree(5)` is the same as `tree(5, [])`.
    - Emphasize variable types.
    - It may be helpful to mark pairs of parentheses to help in understand the nesting relationships.
      - Branches is a function that returns a list of trees.
      - Label values are numbers.
  - Construct Python code given a graphical representation of a tree

BEFORE `prune(1)`: 

AFTER `prune(1)`: 

1. 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.

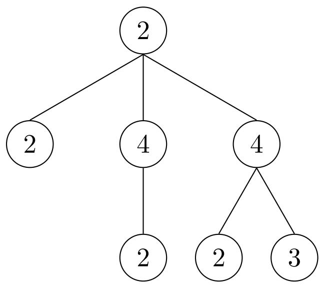
```
def prune(t, k):
    if k == 0:
        return tree(label(t))
    else:
        return tree(label(t), [prune(b, k - 1) for b in branches(t)])
```

Teaching Tips

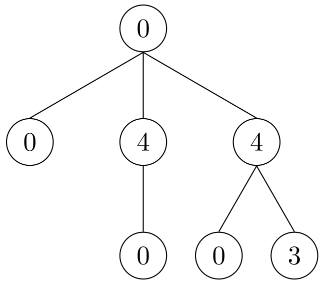
- Emphasize what the base case is/draw it out.
- Emphasize how we can use a tree constructor to create the recursive case. What do we increment down by in the arguments of `prune` to get the base case?
- Tree problems lend themselves to list comprehensions. How are list comprehensions utilized in this problem?

2. 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 _________________________________
```

```
def replace_x(t, x):
    new_branches = []
    for b in branches(t):
        new_branches.append(replace_x(b, x))
    if label(t) == x:
        return tree(0, new_branches)
    return tree(label(t), new_branches)
```

An alternate solution using list comprehensions is as follows:

```
def replace_x(t, x):
   new_branches = [replace_x(b, x) for b in branches(t)]
   if label(t) == x:
        return tree(0, new_branches)
   return tree(label(t), new_branches)
```

Here, we construct and return a new tree. First, we construct a new list of branches where each branch is the same as the previous branch but all occurrences of x have been replaced with 0 as per our recursive function. The if statement guarantees that if our root node's label is an occurence of x, we replace the subtree we're on starting at its root – keeping all else before it the same while replacing the specific subtree's root node label to be zero.
We do not need a base case here, as if we are at a leaf, the list comprehension we use to create the new branches will evaluate to an empty list. Then we will either return `tree(0, [])` or `tree(label(t), [])` as appropriate.

Teaching Tips

- Draw out a tree and ask them to play out the algorithm
  - If you were a computer, how would you replace all the `x`'s? (Answer: check the value of the current tree, then each of the branches)
  - If they're struggling with playing out the algorithm, make sure to draw out how recursion would work in these cases.
  - Can we somehow "simplify" all of this repeated work? (A/N: This simplification begs for list comprehensions to be used. The list comprehension isn't intuitive so have them first work through the for loop skeleton code.)
- Make sure they respect abstraction barriers
  - If there isn't a `set_value` function, how can we return a tree with an updated value? (Answer: create a new tree with `0` and the new branches)
  - Another way to phrase this: What is the only way in which we can change values within the Tree ADT?
- Warn them against trying to evaluate branches
  - What is the simplest replacement we can do?
  - How can we delegate branch replacements to recursive calls?
- If we have multiple branches, how do we make the recursive call on each branch? (Answer: a for loop)
  - What happens in the for loop if there aren't any branches? (Answer: nothing)
  - This is why we don't need an explicit base case (ex. if len(branches) == 0)

3. [Exam Level] Write a function that returns `True` if and 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_n(1, 2, t1)
    True
    >>> contains_n(2, 2, t1)
    False
    >>> contains_n(2, 1, t1)
    True
    >>> t2 = tree(1, [tree(2), tree(1, [tree(1), tree(2)])])
    >>> contains_n(1, 3, t2)
    True
    >>> contains_n(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif ___________________________________________:

        return _____________________________________

    elif label(t) == elem:

        return _____________________________________

    else:

        return _____________________________________
```

```
def contains_n(elem, n, t):
    if n == 0:
        return True
    elif is_leaf(t):
        return n == 1 and label(t) == elem
    elif label(t) == elem:
        return True in [contains_n(elem, n - 1, b) for b in
          branches(t)]
    else:
        return True in [contains_n(elem, n, b) for b in
          branches(t)]
```

Teaching Tips

1. We have purposely left one line return statements to imply that we are using list comprehension for our solution, so please emphasize to your students that hint when walking through the

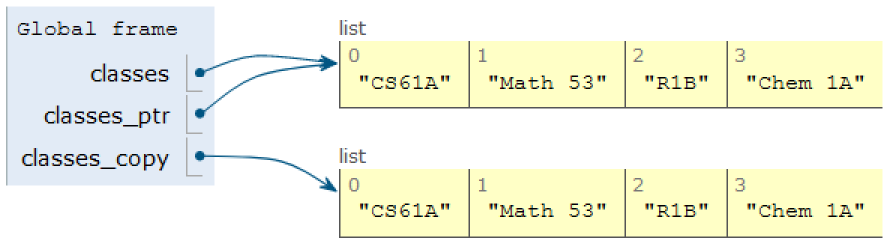
problem.

2. Feel free to use the `any` Python built-in instead, which takes in a list of values and returns `True` if any of the values are truthy and `False` otherwise.
3. Illustrate how `n` can be updated in our recursive calls in order to keep track of how many instances we've seen so far.
4. The second base case is slightly tricky, so you're advised to start with the recursive calls first, which will make that base case make more sense.

2 Mutability

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

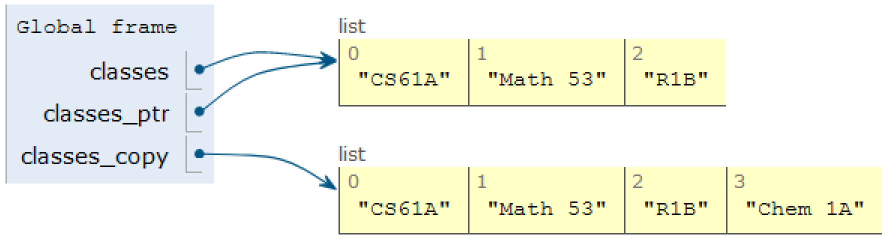
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



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)

Teaching Tips

- Common Misconceptions:

  - Students may be confused about the return value of mutation functions
    * Try contrasting `pop` with `remove`, showing them how only `pop` returns the element
    * Tell them to reference the list mutability table
- The objectives for students are to:
  - Distinguish between mutable and non-mutable objects
  - The effects and return values of mutation functions
  - Become comfortable with pointers and how to copy objects

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
```

```
[1, 2, [3, 4]]
```

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

```
[1, 2, [6, 4]]
```

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

```
TypeError: 'int' object is not iterable
```

```
>>> a
```

```
[5, 2, [6, 4], 7, 8]
```

Challenge:

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

```
6
```

- Draw a box and pointer diagram
- Discuss shallow vs. deep copying.
  - Shallow copying is when you copy each element as is; i.e. elements which were pointers to a list still point to the same list in the copy.
  - Deep copying is when you copy each element within each sublist; i.e. the new elements which are pointers point to brand-new created lists.
  - In general, most operators involving Python lists perform shallow copying: i.e. slicing, list(...), etc.
- If you have enough time, it is helpful for students to make a chart which the different operators and identify when to mutate or create a new list.
- Remind students of the difference between `a += b` and `a = a+b`. The former is essentially `a.extend(b)`, while the latter creates a new list consisting of all the elements of `a` and `b` combined and binds it to `a`.

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:
            ____________________________________

            ____________________________________

    _________________________________
```

```
def accumulate(lst):
    sum_so_far = 0
    for i in range(len(lst)):
        item = lst[i]
        if isinstance(item, list):
            inside = accumulate(item)
            sum_so_far += inside
        else:
            sum_so_far += item
            lst[i] = sum_so_far
    return sum_so_far
```

Teaching Tips

- To keep track of the accumulated sum we need to create a variable that we update every time we see a new element.
- Make sure to emphasize the distinction between `for item in lst` and `for i in range(len(lst))`.
  - We need `i` in order to mutate the list. Why does using `for item in lst` not work when mutating? (Answer: because we're using a copy of the element, not modifying the original list).
  - Perhaps allow your students to first make the mistake of using the former, so that they may realize this difference on their own. Granted, if they aren't able to catch this on their own, do nudge them in the right direction.
- Why do we need a conditional in the for loop? What do we do when we have a nested list?
  1. Integers: For integers we just add the value to the ongoing sum and then mutate the current index of the list to be the cumulative sum
  2. Lists: We need to break down the list and get the values, both so that we can update them and so that we can add it into our sum. However, we don't know how many levels of nesting we have in our list
     - We could have something like `[1, [2, [3]]]`, so we need a function that will sum up the values from a potentially nested list. Do we have a function that does this?
     - **Emphasize the recursive leap of faith when calling accumulate on the inner list**
     - Remind students that they can use **`isinstance`** to check if an element is a list.
- We return the accumulated sum of the list which includes all values, even the nested ones because of the recursive call.

3. **Scrabble!** [Exam Level - Adapted from CS61A Su19 Midterm Q7(b)]

Implement `scrabbler` which takes in a string `chars`, a list of strings `words`, and a dictionary `values` which maps letters to point values. It should return a dictionary where each key is a word in `words` which can be formed from the letters in `chars` and each value is the point value of that word.

For this problem, we will only consider words we can form using letters in `chars` in the same order they appear in the string. Assume values contains all the letters across valid words as keys.

Furthermore, you have access to the function `is_subseq` which returns `true` if a string is a subsequence of another string. A string is a sequence of another string if all the letters in the first string appear in the second string in the same order (but they do not need to be next to each other).

```
def scrabbler(chars, words, values):
    """ Given a list of words and point values for letters, returns a
    dictionary mapping each word that can be formed from letters in chars
    to their point value. You may not need all lines
    >>> words = ["easy", "as", "pie"]
    >>> values = {"e": 2, "a": 2, "s": 1, "p": 3, "i": 2, "y": 4}
    >>> scrabbler("heuaiosby", words, values)
    {'easy': 9, 'as': 3}
    >>> scrabbler("piayse", words, values)
    {'pie': 7, 'as': 3}
    """
    result = ____________________________________

    for ___________________________________________:

        if ________________________________________:

            ____________________________________

            ____________________________________

    return result
```

```
    def scrabbler(chars, words, values):
        result = {}
        for w in words:
            if is_subseq(w, chars):
                total = sum([values[c] for c in w])
                result[w] = total
        return result
```

Teaching Tips

1. The original exam problem had 4 lines after the **`if`** statement, so a solution that doesn't use list comprehension is acceptable. You might want to go over that case with your students.
2. I did not leave space for the non-list comprehension solution here because it seems like list comprehension is becoming almost an ”expectation” of MT2 & the final, so it is good to get students used to using them where ever possible.

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])` $\rightarrow$ `6`. Optionally, an initializer may be provided: `reduce(add, [1], 5)` $\rightarrow$ `6`.

Technically, `map` and `filter` are not functions but classes, but that is not a distinction we need to make.

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()
```

Something to really emphasize here is the difference between regular function execution and generator function execution. When you call a generator function, you do not begin executing the function body! You only begin executing the function body when **next** is called on the generator object. You then pause when you hit a `yield` statement. I like to tell my students that this is an "abuse of notation": they're coopting function syntax to do something completely different from what a function normally does.

Another thing I like to emphasize is that it is impossible to go "backward" with iterators and generators. After all, we only have a **next**, not a `prev`!

You might find it advantageous to go over some of the examples more in depth.

You may or may not find it useful to present students with an example of how iteration works behind the scene:

```
for x in "Hello":
    print(x)
```

```
it = iter("Hello")
while True:
    try:
        x = next(it)
        print(x)
    except StopIteration:
        pass
```

It's possible this may confuse some students, so be cautious if you attempt to use this or a similar example. In particular, students may be confused by the infinite looping and the **try** and **except** blocks. While error handling isn't something super important in CS 61A, they should be able to use it specifically for dealing with iterators, so it might be a good idea to go over this a bit with your students.

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()
```

```
<generator object>
```

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

```
Hello
0
```

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

```
World
StopIteration
```

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

```
Hello
0
World
```

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

```
-1
```

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

```
16
```

Teaching Tips

- Emphasize heavily the fact that when generators are called, they return a generator object. They do NOT start executing their function body until after `next` is called! (So what does that first line return? A generator object!)
- Remind students that generator objects are independent from one another; if you create a new one from calling the same function again, it starts from the beginning again. Each generator on its own, however, remembers where it stopped after the previous `next` call, so that it can resume the next time you call `next`.
- What happens when there are no more `yield` statements, like in the second `call` on `foo_gen`? The generator has reached the end of all possible values to iterate over, and so it returns a StopIteration error.
- If you stick a generator object inside a for loop (or a list, for that matter), it will go all the way through from start to finish, outputting each yield value after another.
  - Careful, however: 'start' doesn't necessarily mean the very first lines of the function or the first yield call; if you feed in a generator on which you've already called `next`, its "start" will be where it last left off.

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]
    """

    if len(lst) == 0:
        yield 0
    else:
        for sum_rest in all_sums(lst[1:]):
            yield sum_rest + lst[0]
            yield sum_rest
```

Teaching Tips

- This is a classic tree recursion problem but now in generator form!
- A tree diagram of how the list splits is a good visualization to draw
- Students may have trouble with this because the order in which they're dealing with the recursive case is a bit different than usual.
- If students are struggling to understand the problem, start from the base case of an empty list and work your way up with the sums of a length-1 list, length-2, etc.
- As always, the recursive leap of faith is helpful in understanding what `all_sums(lst[1:])` returns.
- Even though this is a generator problem, we iterate over the call in a for loop so we can treat the function like it returns a list!