Mutability, Iterators, and Generators Meta

Computer Science Mentors 61A

March 6–March 10, 2023

Recommended Timeline

- Mutability
  - Mutability Mini Lecture: 6 minutes
  - What Would Python Do: 5 minutes
  - Nice Ice Cream: 10 minutes
  - Accumulate: 13 minutes
  - There are a lot of questions and topics, so pick and choose between these problems based on your students comfortability.
- Iterators & Generators:
  - Iterators & Generators Mini Lecture: 7 minutes
  - Skip Machine: 16 minutes (worth it though, in content's humble opinion)
  - Foo: 8 minutes
  - In Order: 5 minutes
  - All Sums: 10 minutes

As a reminder, these times do not add up to 50 minutes because no one is expected to get through all questions in a section. This is especially true this week, because this worksheet is rather long. You should use the worksheet as a problem bank around which you can structure your section to best accommodate the needs of your students. Both before and during section, consider which questions would be most instructive and how you should budget your time. I also highly recommend directly asking your students what they would like to focus on.

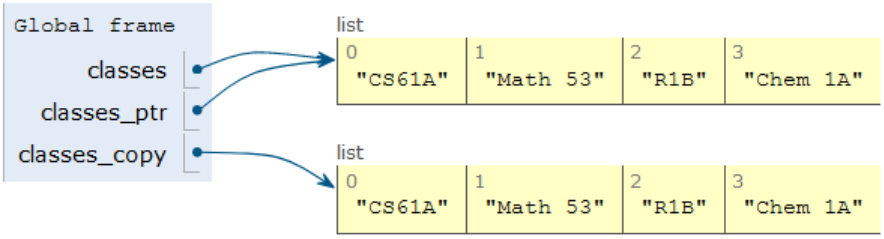
In general, though, I'd recommend planning at the start of section how long you're going to spend on each section (say, 25 minutes on mutability and 25 minutes on iterators and generators) and then restrict yourself to that time budget so that you can get good coverage of different topics.

1 Mutability

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

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang
Provided by CSM 61A contributors under CC BY 4.0/3-clause BSD.

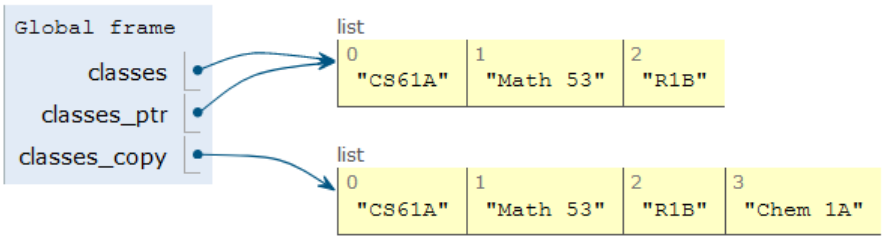
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



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. Produce the environment diagram and output that result from executing the code below.

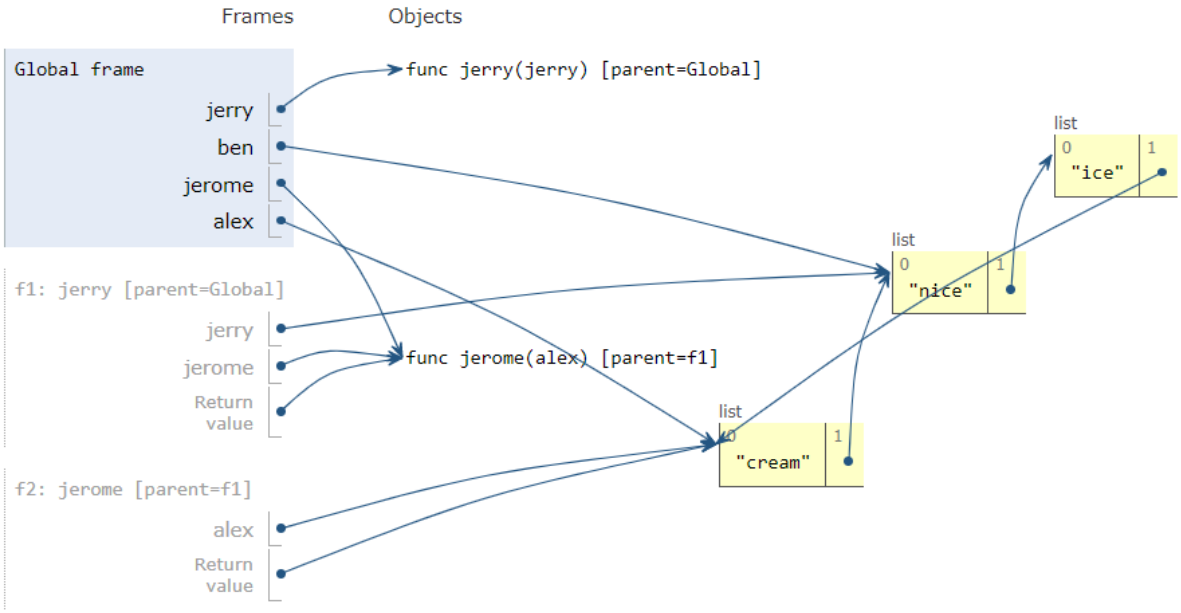
```
def jerry(jerry):
    def jerome(alex):
        alex.append(jerry[1:])
        return alex
    return jerome

ben = ['nice', ['ice']]
jerome = jerry(ben)
alex = jerome(['cream'])
ben[1].append(alex)
ben[1][1][1] = ben
print(ben)
```

https://goo.gl/uhSClr

Print output (drag lower right corner to resize)

```
['nice', ['ice', ['cream', [...]]]]
```



3. 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.

2 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. What Would Python Display?

```
class SkipMachine:
    skip = 1
    def __init__(self, n=2):
        self.skip = n + SkipMachine.skip

    def generate(self):
        current = SkipMachine.skip
        while True:
            yield current
            current += self.skip
            SkipMachine.skip += 1

p = SkipMachine()
twos = p.generate()
SkipMachine.skip += 1
twos2 = p.generate()
threes = SkipMachine(3).generate()
```

(a) **next**(twos)

2

(b) **next**(threes)

2

(c) **next**(twos)

5

(d) `next(twos)`

8

(e) `next(threes)`

7

(f) `next(twos2)`

5

2. 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.

3. Define a generator function `in_order`, which takes in a tree `t`; assume that `t` and each of its subtrees have either 0 or 2 branches only. Fill in `in_order` to yield the labels of `t` "in order"; that is, for each node, the labels of the left branch should precede the parent label, which should precede the labels of the right branch.

```
def in_order(t):
    """
    >>> t = tree(0, [tree(1), tree(2, [tree(3), tree(4)])])
    >>> list(in_order(t))
    [1, 0, 3, 2, 4]
    """
```

```
def in_order(t):
    if is_leaf(t):
        yield label(t)
    else:
        yield from in_order(branches(t)[0])
        yield label(t)
        yield from in_order(branches(t)[1])
```

Teaching Tips

- Trees are meant to be implemented recursively, and this should be emphasized to students.
- What is the base case of the problem? With trees it is typically the leaf, and it works out in this case, where there is only one item to yield.
- Draw out an example of a tree (maybe the doctest). What do we expect the recursive call on each of the branches to return (note that trees either have 0 or 2 branches)?
- After seeing what the recursive calls do, figure out how you combine the label, the left tree recursive call, and the right tree recursive call to get the desired result. Yielding the left recursive call's values, then the label, and then the right recursive call will give the in-order traversal.

4. 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!