# Mutability, Iterators, and Generators Meta

Computer Science Mentors 61A

October 10–October 14, 2022

**Recommended Timeline**

- Mutability
  - Mutability Mini Lecture: 6 minutes
  - What Would Python Do: 5 minutes
  - Nice Ice Cream: 7 minutes
  - Insert N: 10 minutes
  - There are a lot of questions and topics, so pick and choose between these problems based on your students comfortability.
- Iterator & Generator:
  - Iterators & Generators Mini Lecture: 7 minutes
  - Foo: 4 minutes
  - Accumulate: 7 minutes
  - In Order: 10 minutes
  - All Sums: 15 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

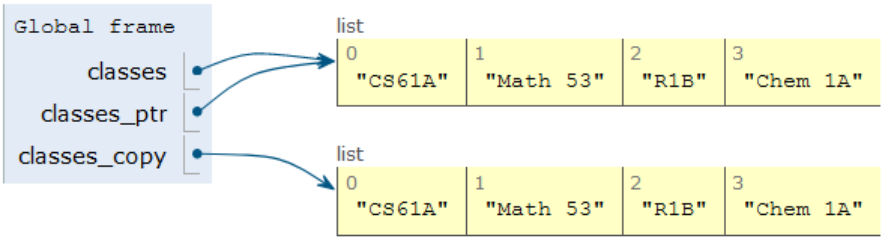
Vibha Tantry and Jordan Schwartz, with
Gabe Classon, Aditya Balasubramanian, Oliver Yu, Hanze Tan, Gene Pan, Carolyn Wang, Raman Varma, Noor Haq, Angela Zhang, Abigail Brooks, Ted Kim, Aiden Joshua Vehemente, Marie Chorpita, Ashley Chiu, Bill Hu, Omar Yu, Angel Alberto Aldaco, Andrew Park, Chase Clements, Grace Yi, Preshtha Garg, Laksith Venkatesh Prabu, Aditya Murali, Irene Yang, Andy Chen, and Hailey Park

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!

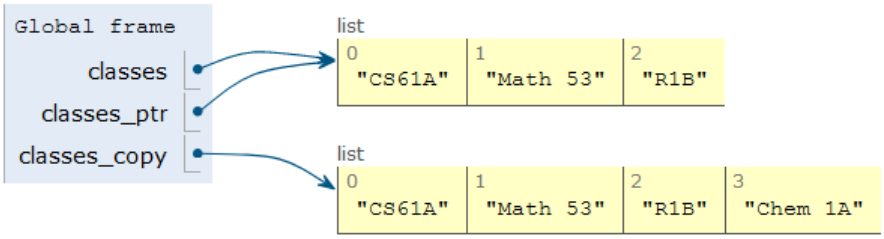
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```




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 a few other list methods that mutate:

- `append(el)`: Adds `el` to the end of the list
- `extend(lst)`: Extends the list by concatenating `lst` onto the end
- `insert(i, el)`: Inserts `el` at index `i` (does not replace element but adds a new one)
- `remove(el)`: Removes the first occurrence of `el` in the list; errors if `el` is not in the list
- `pop(i)`: Removes and returns the element at index `i`; if no index is provided, it removes and returns the last element of the list

In addition to these methods, there are a few other built-in ways to mutate lists:

- `lst += lst` (**This is distinct from** `lst = lst + lst`)
- `lst[i] = x`
- `lst[i:j] = lst`

On the other hand, the following non-mutative (*non-destructive*) operations do not change the original list but create a new list instead:

- `lst + lst`
- `lst * n`
- `lst[i:j]`
- **`list`**`(lst)`

## 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`.

**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<br>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 |

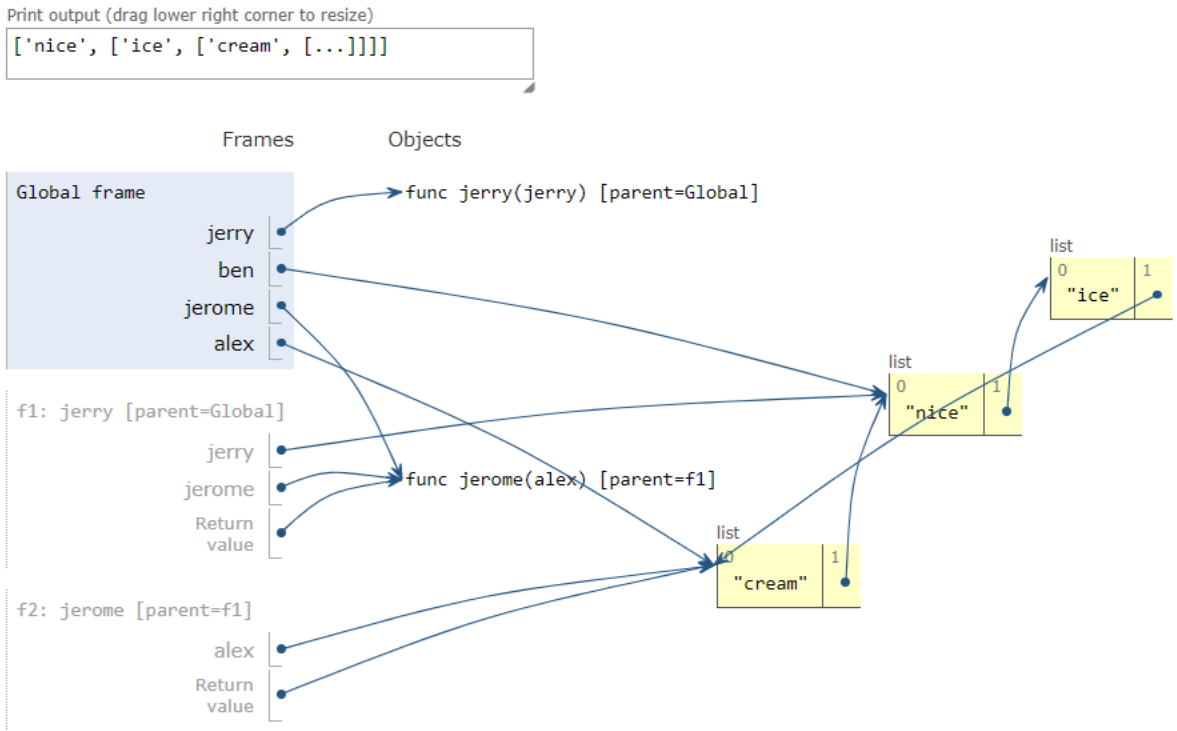
(credits: Mihira Patel)

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/uhSC1r



3. Write a function `insert_n`, which takes in an ascending list of numbers `lst`, a number `x`, and an integer `n`. If `n` is positive, `insert_n` mutatively inserts `n` copies of `x` into `lst` at the correct position so that `lst` is still in ascending order. If `n` is negative, `insert_n` mutatively removes —`n`— copies of `x` from `lst`. (Assume that there are always enough copies to `x` to be removed.)

```
def insert_n(lst, x, n):
    """
    >>> lst = []
    >>> insert_n(lst, 4, 1)
    >>> insert_n(lst, 1, 3)
    >>> insert_n(lst, 2, 2)
    >>> lst
    [1, 1, 1, 2, 2, 4]
    >>> insert_n(lst, 1, -2)
    >>> lst
    [1, 2, 2, 4]
    """
    if n > 0:
        i = 0
        while _____________________________________________:

            ____________________________________________

        ____________________________________________:

            ____________________________________________
    elif n < 0:

        ____________________________________________

            ____________________________________________
```

```
def insert_n(lst, x, n):
    """
    >>> lst = []
    >>> insert_n(lst, 4, 1)
    >>> insert_n(lst, 1, 3)
    >>> insert_n(lst, 2, 2)
    >>> lst
    [1, 1, 1, 2, 2, 4]
    >>> insert_n(lst, 1, -2)
    >>> lst
    [1, 2, 2, 4]
    """
    if n > 0:
        i = 0
        while i < len(lst) and lst[i] < x:
            i += 1
        for _ in range(n):
            lst.insert(i, x)
    elif n < 0:
        for _ in range(-n):
            lst.remove(x)
```

The purpose of this problem is to give students some hands on experience with list mutation methods, and also to give them a refresher on how to handle lists.

The trickiest part of this problem is figuring out where the elements need to be added. It is not sufficient to simply look for the earliest instance of `x` in `lst`, for example, because `lst` may not have any instances of `x` to start with. If students are stuck on this, you can try asking a series of leading questions to help them get there. For example: where do we want to insert our new elements? How can we insert our new elements into the correct place in the list? How can we figure out the correct index to insert our new elements? Why did they tell you that the list is in ascending order?

## 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.

There are a few useful functions that act on iterables that are particularly useful:

- **map**`(f, it)`: Returns an iterable that produces each element of `it` with the function `f` applied to it.
- **filter**`(pred, it)`: Returns an iterable 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], 0)` $\rightarrow 1$.

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. Write a generator function that takes in an iterator `it` and yields the running total of the elements produced by `it`.

```
def accumulate(it):
    """
    >>> def all_ints():
    ...     i = 0
    ...     while True:
    ...         yield i
    ...         i += 1
    >>> a = accumulate(all_ints())
    >>> [next(a) for x in range(6)]
    [0, 1, 3, 6, 10, 15]
    """
```

```
def accumulate(it):
    sum = 0
    while True:
        sum += next(it)
        yield sum
```

This problem is meant to be a relatively simple introduction to the manipulation of iterators, generator function syntax, and how to deal with infinite generators.

The doc test for this question is deliberately a bit complicated. It actually serves as a bit of a hint for how we might construct an infinite generator if one is provided to us (using an infinite loop). If students are concerned about this possibly erroring, remind them that generators only execute the body of the function when we tell them to, so this will not actually cause an infinite loop that hogs all of our computer's resources, causing a crash.

Students have a tendency to want to do everything in generators with lists, which requires the computation of all elements of the generator at the same time; we need to encourage them to do "lazy" iteration, which means that each element is only computed when it's needed. Lazy iteration is a core concept of the topic; not only is it more efficient to do things this way, it's literally the only way to handle infinite iterators (whose elements cannot all be computed at the same time). Whenever I do iteration problems with my students, I ask them to consider what would happen if we provided them an infinite generator; if their implementation breaks, then it's not correct.

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!