

MUTABILITY, ITERATORS, AND GENERATORS Solutions

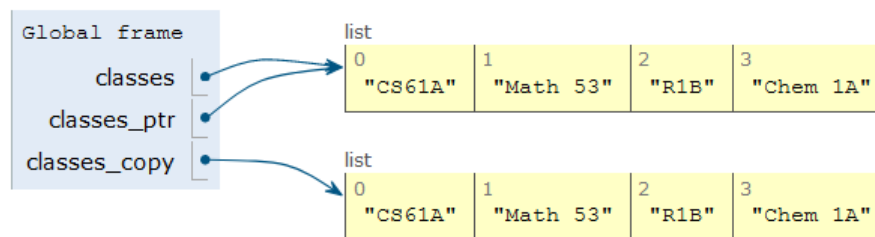
COMPUTER SCIENCE MENTORS 61A

March 6–March 10, 2023

1 Mutability

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

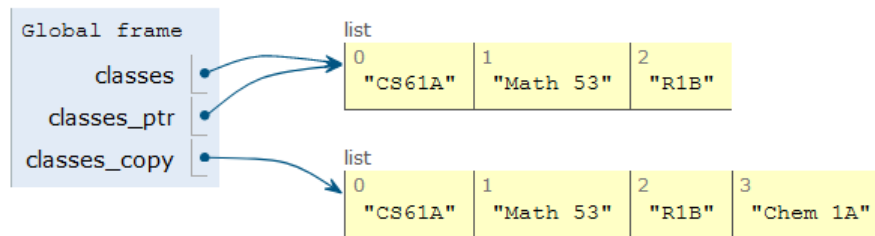
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



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 more list methods that mutate:

Mutability in Lists

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

```
[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
```

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

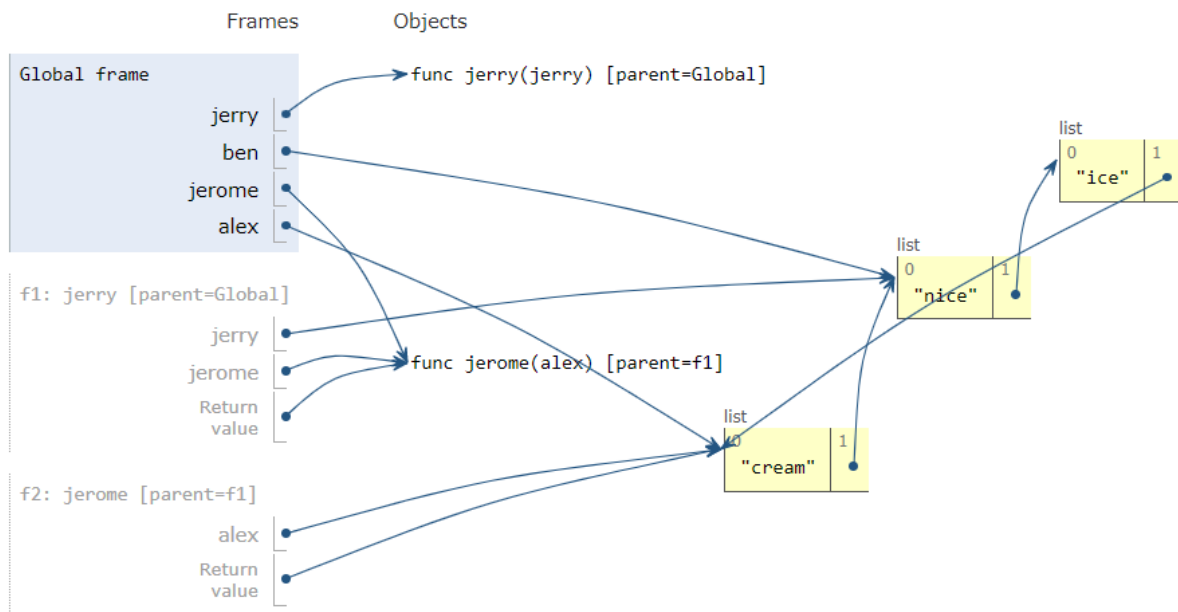
```
def jerry(jerry):  
    def jerome(alex):  
        alex.append(jerry[1:])  
        return alex  
    return jerome
```

```
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
```

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.

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. 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
```


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])
```

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
```