MUTABILITY, ITERATORS, AND GENERATORS

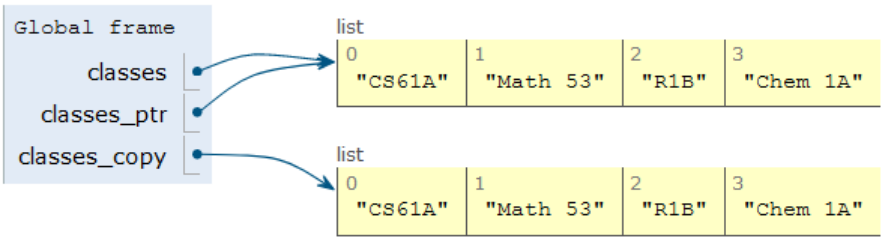
COMPUTER SCIENCE MENTORS 61A

October 10–October 14, 2022

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[:]
```

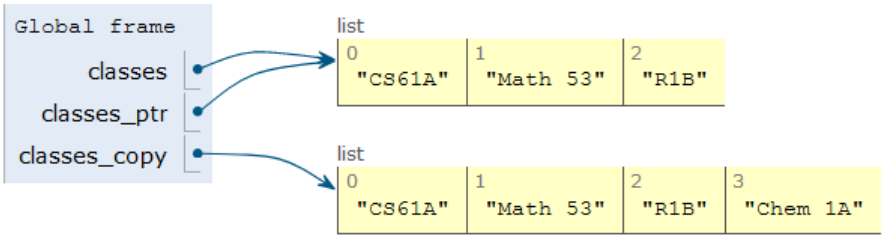


After a few weeks, you realize that you cannot keep up with the workload and you need to drop a class. You've chosen to drop Chem 1A. Based on what we know so far, to change our classes list, we would have to create a new list with all the same elements as the original list except for Chem 1A. But that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

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

```
>>> 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)`

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. 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)
```

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:

        ___________________________________________

            ___________________________________________
```

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`.

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. 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]
    """
```

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]
    """
```

.

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]
    """
```