CONTAINERS, SEQUENCES, MUTABILITY

CSM 61A

February 21, 2022 to February 25, 2022

1 Lists

Lists Introduction:

Lists are a type of sequence, an ordered collection of values that has both length and the ability to select elements.

```
>>> lst = [1, False, [2, 3], 4] # a list can contain anything
>>> len(lst)
4
>>> lst[0] # Indexing starts at 0
1
>>> lst[4] # Indexing ends at len(lst) - 1
Error: list index out of range
```

We can iterate over lists using their index, or iterate over elements directly

```
for index in range(len(lst)):
        # do things
for item in lst:
        # do things
```

List comprehensions are a useful way to iterate over lists when your desired result is a list.

```
new_list2 = [<expression> for <element> in <sequence> if <
   condition>]
```

We can use **list slicing** to create a copy of a certain portion or all of a list.

```
new_list = lst[<starting index>:<ending index>]
copy = lst[:]
```

1. Draw box-and-pointer diagrams for the following:

```
>>> a = [1, 2, 3]
>>> a

>>> a[2]

>>> b = a
>>> a = a + [4, 5]
>>> a

>>> b

>>> c = a
>>> a = [4, 5]
>>> a


>>> c

>>> d = c[0:2]
>>> c[0] = 9
>>> d
```

2. Write a function `duplicate_list`, which takes in a list of positive integers and returns a new list with each element `x` in the original list duplicated x times.

```
def duplicate_list(lst):
    """
    >>> duplicate_list([1, 2, 3])
    [1, 2, 2, 3, 3, 3]
    >>> duplicate_list([5])
    [5, 5, 5, 5, 5]
    """
    ____________________________________

    for ____________________________________:

        for ____________________________________:

            ____________________________________

    ____________________________________
```

3. Write a list comprehension that accomplishes each of the following tasks.

   (a) Square all the elements of a given list, `lst`.

   (b) Compute the dot product of two lists `lst1` and `lst2`. *Hint*: The dot product is defined as $\text{lst1}[0] \cdot \text{lst2}[0] + \text{lst1}[1] \cdot \text{lst2}[1] + \ldots + \text{lst1}[n] \cdot \text{lst2}[n]$. The Python **`zip`** function may be useful here.

   (c) Return a list of lists such that `a = [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]`.

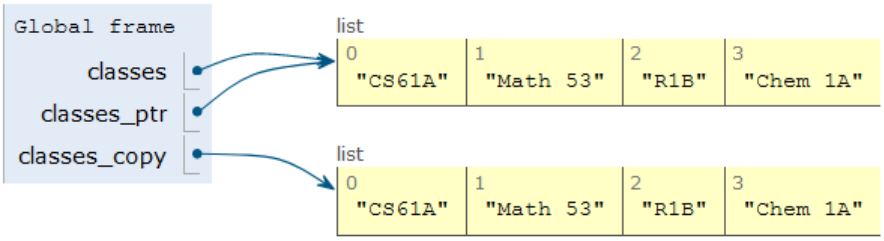
   (d) Return the same list as above, except now excluding every instance of the number 2: `b = [[0], [0, 1], [0, 1], [0, 1, 3], [0, 1, 3, 4]])`.

2 Mutability

Mutation

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

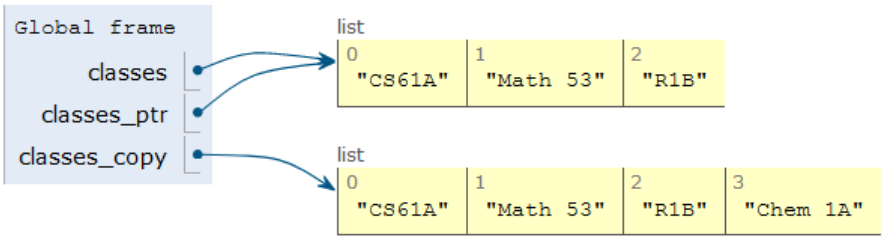
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



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"
```



List methods that mutate:

- `append(el)`: Adds el to the end of the list
- `extend(lst)`: Extends the list by concatenating it with lst
- `insert(i, el)`: Insert el at index i (does not replace element but adds a new one)
- `remove(el)`: Removes the first occurrence of el in list, otherwise errors
- `pop(i)`: Removes and returns the element at index i, if you do not include an index it pops the last element of the list

Ways to copy: list splicing ([start:end:step]), **`list`**`(...)`

Mutable vs immutable

Mutative (*destructive*) operations change the state of a list by adding, removing, or otherwise modifying the list itself.

- `lst.append(element)`
- `lst.extend(lst)`
- `lst.pop(index)`
- `lst += lst` (**Note - this is different than:** `lst = lst + lst`)
- `lst[i] = x`

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. Draw the environment diagram that results from running the following code.

```
ghost = [1, 0,[3], 1]
def boo(spooky):
  ghost.append(spooky.append(ghost))
  spooky = spooky[ghost[2][1][1]]
  ghost[:].extend([spooky])
  spooky = [spooky] + [ghost[spooky - 1].pop()]
  ghost.remove(ghost.remove(1))
  spooky += ["Happy Halloween!"]
  return spooky
pumpkin = boo(ghost[2])
```

3. Given some list `lst`, possibly a deep list, mutate `lst` to have the accumulated sum of all elements so far in the list. If there is a nested list, mutate it to similarly reflect the accumulated sum of all elements so far in the nested list. Return the total sum of the original `lst`.

   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:
            _____________________________________
            _____________________________________
    return ___________________________________
```