LISTS AND LIST COMPREHENSION 2

DATA C88C

January 29, 2024

1 Lists

Let's imagine you order a mushroom and cheese pizza from Domino's, and that they represent your order as a list:

```
>>> pizza1 = ['cheese', 'mushrooms']
```

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, Domino's would have to build an entirely new list to add onions:

```
>>> pizza2 = pizza1 + ['onions'] # creates a new python list
>>> pizza2
['cheese', mushrooms', 'onions']
>>> pizza1 # the original list is unmodified
['cheese', 'mushrooms']
```

But this is silly, considering that all Domino's had to do was add onions on top of `pizza1` instead of making an entirely new `pizza2`.

Python actually allows you to *mutate* some objects, includings lists and dictionaries. Mutability means that the object's contents can be changed. So instead of building a new `pizza2`, we can use `pizza1.append('onions')` to mutate `pizza1`.

```
>>> pizza1.append('onions')
>>> pizza1
['cheese', 'mushrooms', 'onions']
```

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created. We can use the familiar indexing operator to mutate a single element in a list. For instance `lst[4]='hello'` would change the fifth element in `lst` to be the string `'hello'`. In

addition to the indexing operator, lists have many mutating methods. List *methods* are functions that are bound to a specific list. Some useful list methods are listed here:

1. `append(el)` adds `el` to the end of the list
2. `insert(i, el)` insert `el` at index `i`
3. `remove(el)` removes the first occurrence of `el` in list, otherwise errors
4. `sort()` sorts elements of list *in place*

List methods are called via *dot notation*, as in:

```
>>> colts = ['andrew luck', 'reggie wayne']
>>> colts.append('trent richardson')
```

None of the mutating list methods *return* a new list — they simply modify the original list and return `None`.

2 For Loops and List Comprehensions

There are two common methods of looping through lists. If you don't need indices, looping over elements is usually more clear.

- `for el in lst` loops through the elements in `lst`
- `for i in range(len(lst))` loops through the valid, positive indices of `lst`

A **list comprehension** is a compact way to create a list whose elements are the results of applying a fixed expression to elements in another sequence.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

Let's break down an example:

```
[x * x - 3 for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

In this list comprehension, we are creating a new list after performing a series of operations to our initial sequence `[1, 2, 3, 4, 5]`. We only keep the elements that satisfy the filter expression `x % 2 == 1` (`1`, `3`, and `5`). For each retained element, we apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in the output `[-2, 6, 22]`.

Note: The `if` clause in a list comprehension is optional.

3 Questions

1. What would Python print?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
```

```
>>> [i * i - i for i in [5, -1, 3, -1, 3] if i > 2]
```

```
>>> [[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
```

2. Define a function `foo` that takes in a list `lst` and returns a new list that keeps only the even-indexed elements of `lst` and multiplies each of those elements by the corresponding index.

```
def foo(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> foo(x)
    [0, 6, 20]
    """

    return [_____]
```

3. Write a function `square_elements` which takes a `lst` and replaces each element with the square of that element. *Mutate* `lst` *rather than returning a new list*.

```
def square_elements(lst):
    """
    >>> lst = [1, 2, 3]
    >>> square_elements(lst)
    >>> lst
    [1, 4, 9]
    """
```

4. Write a function that takes in two values `x` and `el`, and a list, and adds as many `el`'s to the end of the list as there are `x`'s.

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    """
```

5. Reverse a list *in place*, i.e. mutate the given list itself, instead of returning a new list.

```
def reverse(lst):
    """ Reverses lst in place.
    >>> x = [3, 2, 4, 5, 1]
    >>> reverse(x)
    >>> x
    [1, 5, 4, 2, 3]
    """
```