Abstraction

Imagine you wanted to build a game that required the use of a board of *size* x *size* dimensions. Consider the following constructor and selector functions:

```
def gameboard(size):
    # Return a gameboard of dimensions size x size.
    # YOUR CODE HERE

def get_piece(board, x, y, piece):
    # Return the piece of a gameboard at position x, y. x refers to the row number
    # and y refers to the column number. Both x and y begin at 0.
    # YOUR CODE HERE

def place_piece(board, x, y, piece):
   # Place a piece at position x, y of the gameboard and return the new gameboard. x
   # refers to the row number and y refers to the column number. Both x and y begin
   # at 0.
   # YOUR CODE HERE
```

1a.) Implement the above using only *one* list. So for example, if you had a 2 x 2 board where all of the items are 0, the board should be a list of [0, 0, 0, 0].
Visually this is equivalent the following:

```
0 0
0 0
```

```
def gameboard(size):
    # Return a gameboard of dimensions size x size.
    board = []
    for i in range(0, size * size):
        board.append(0)
    return board

def get_piece(board, x, y):
    # Return the piece of a gameboard at position x, y.
    return board[x * (int(len(board) ** (1/2))) + y]

def place_piece(board, x, y, piece):
   # Place a piece at position x, y of the gameboard and return the new gameboard.
   board[x * (int(len(board) ** (1/2))) + y] = piece
```

```
  return board
```

1b.) Implement the above abstraction utilizing a dictionary. You may utilize lists as the items of the dictionary if you would like.

```
def gameboard(size):
    # Return a gameboard of dimensions size x size.
    board = {}
    row_list = []
    for col in range(0, size):
        row.append(0)
    for row_num in range(0, size):
        board[row_num + 1] = row_list[:]
    return board

def get_piece(board, x, y):
    # Return the piece of a gameboard at position x, y.
    return board[x][y - 1]

def place_piece(board, x, y, piece):
   # Place a piece at position x, y of the gameboard and return the new gameboard.
   board[x][y - 1] = piece
   return board
```

1c.) Imagine that we didn't know what type of data structure (i.e. lists, dictionaries, tuples, etc.) that we used to implement our gameboard. Why would accessing the board like this (board[0][0]) to get the top-most left piece be an abstraction violation?

The point of abstraction is to make it so that we don't need to know what the data structure is that we're using to implement the board. Because of that, we don't want to access the data structure directly because it might cause an error if we access a data structure incorrectly.

Lambdas

1.) What would this return:

```
bar = lambda y: lambda x: pow(x, y)
foo = lambda: 32
foobar = lambda x, y: x // y
a = lambda x: foobar(foo(), bar(4)(x))
>>>a(2)
```

Answer: 2

2.) writing a function count_cond, which takes in a two-argument predicate function condition(n, i). count_cond returns a one-argument n function that counts all the numbers from 1 to n that satisfy condition. i refers to every number from 1 to n.

Answer: def count_cond(condition):

```
    """Returns a function with one parameter N that counts all the numbers from
    1 to N that satisfy the two-argument predicate function CONDITION.
   def counter(n):
        i, count = 1, 0
        while i <= n:
            if condition(n, i):
                count += 1
            i += 1
        return count
    return counter
```

Dictionaries

1.) Write a dictionary that converts a list into a nested dictionary of keys.

Def list_to_keyDict(lst):

“””list_to_keyDict([1,2,3,4]) would return {1: {2: {3: {4: {}}}}}”””

Answer:

```
new_dict = current = {}
for name in lst:
    current[name] = {}
    current = current[name]
Return (new_dict)
```

2.) Write a Python program to count the values associated with key in a list of dictionary.

def countVal(dl, key):

return _____(sum(d[key] for d in dl)_____

Example:

student = [{'id': 1, 'success': True, 'name': 'Lary'},
 {'id': 2, 'success': False, 'name': 'Rabi'},
 {'id': 3, 'success': True, 'name': 'Alex'}]

countVal(student, ‘success’) would return 2

Tuples

```
>>> t = (1, 2, 3, 4)
>>> s = (1, 2, 4, 3)
```

What would the following questions output?

a) `>>> t[3] = 45` Error
b) `t * 2` (1, 2, 3, 4, 1, 2, 3, 4)
c) `t[1:-1]` (2, 3)
d) `s < t` True
e) `t.append(s)` Error

Trees

```
# Constructor
def tree(label, branches=[]):
        for branch in branches:
                assert is_tree(branch)
        return [label] + list(branches)

# Selectors
def label(tree):
        return tree[0]

def branches(tree):
        return tree[1:]

# For convenience
def is_leaf(tree):
        return not branches(tree)
```

Use the above functions to write the functions below:

1.
```
def tree_max(t):
        """Returns the max of a tree."""
        return max([label(t)] + [tree_max(branch) for branch in branches(t)])
```

2.
```
def height(t):
        """Returns the height of a tree."""
        if is_leaf(t):
                return 0
        return 1 + max([height(branch) for branch in branches(t)])
```

3.
```
def square_tree(t):
        """Returns a tree with the square of every element in t."""
        sq_branches = [square_tree(branch) for branch in branches(t)]
        return tree(label(t)**2, sq_branches)
```

Mutability

Notes on mutating lists:

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 (does not replace element but adds a new one)
3. remove(el): removes the first occurrence of el in list, otherwise errors
4. pop(i): removes and returns the element at index i

1. What would Python display? It may be helpful to draw the box and pointers diagrams to the right in order to keep track of the state.

(a)
```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst1 == lst2 # compares each value
True
```

(b)
```
>>> lst1 is lst2 # compares references
False
```

(c)
```
>>> lst2 = lst1
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
```

(d)
```
>>> lst2
[1, 2, 3, 4]
```

(e)
```
>>> lst1 = lst1 + [5]
>>> lst1 == lst2
False
```

(f)
```
>>> lst1
[1, 2, 3, 4, 5]
```

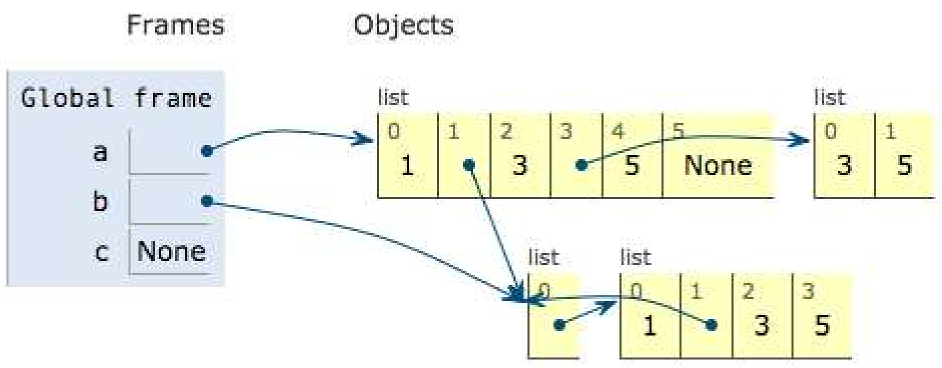
(g)
```
>>> lst2
[1, 2, 3, 4]
```

(h)
```
>>> lst2 is lst1
False
```

2. Draw the environment diagram that results from executing the following code.

```
a = [1, 2, 3, 4, 5]
a.pop(3)
b = a[:]
a[1] = b
b[0] = a[:]
b.pop()
b.remove(2)
c = [ ].append(b[1])
a.insert(b.pop(1), a[2:])
a.append(c)
```

http://goo.gl/YKETVv



3. Write a function that takes in a list and reverses it 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]
        """
        # YOUR CODE HERE

        for i in range(len(lst) // 2):
                lst[i], lst[-i - 1] = lst[-i - 1], lst[i]
```

4. In Lab 4, you wrote a version of deep_copy for a list of lists. However, lists contain an arbitrary number of lists at arbitrary depths. Write a deep_copy function that works for lists with any number of lists nested inside it.

Hint 1: The **isinstance** function returns True for **isinstance**(l, list) if l is a list and False otherwise.

Hint 2: You'll need to use recursion.

```
def real_deep_copy(lst):
    """
    >>> a = [[1, 2], [[3, 4, [5]], 6], 7, [8]]
    >>> b = real_deep_copy(a)
    >>> a[0][1], a[1][0][2], a[3] = 20, 50, 80
    >>> a
    [[1, 20], [[3, 4, 50], 6], 7, 80]
    >>> b
    [[1, 2], [[3, 4, [5]], 6], 7, [8]]
    """
    # YOUR CODE HERE
    ret = []
    for x in lst:
        if type(x) == list:
            ret.append(deep_copy(x))
        else:
            ret.append(x)
    return ret
```

Nonlocal

Until now, you've been able to access variables in parent frames, but you have not been able to modify them. The nonlocal keyword can be used to modify a variable in the parent frame outside the current frame. For example, consider stepper, which uses nonlocal to modify num:

```
def stepper(num):
    def step():
        nonlocal num # declares num as a nonlocal variable
        num = num + 1 # modifies num in the stepper frame
        return num
    return step
```

However, there are two important caveats with nonlocal variables:

- Global variables cannot be modified using the nonlocal keyword.
- Variables in the current frame cannot be overridden using the nonlocal keyword. This means we cannot have both a local and nonlocal variable with the same names in a single frame.

1. Fill in the blank:

Nonlocal variables cannot modify _____global______ variables.

A variable declared nonlocal can't already exist in ________the current frame________.

A nonlocal variable must be declared in a ______parent_______ frame

2. Write a function that takes in a value x and updates and prints the result based on input functions.

```
def memory(n):
        """
        >>> f = memory(10)
        >>> f = f(lambda x: x * 2)
        20
        >>> f = f(lambda x: x - 7)
        13
        >>> f = f(lambda x: x > 5)
        True
        """
        # YOUR CODE HERE

        def f(g):
                nonlocal n
                n = g(n)
                print(n)
                return f
        return f
```

Exceptions - Jobel

1. Why might you want to have an error message specific to exception you catch rather than having a generic error message?

   Having a generic error message isn't very useful as you will not know precisely what is wrong with your code as opposed to having a specific error message.