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, row, col):
    # Return the piece of a gameboard at position row, col. Both row and col start at 0.
    # YOUR CODE HERE

def place_piece(board, row, col, piece):
   # Place a piece at position x, y of the gameboard and return the new gameboard.
   # YOUR CODE HERE
```

1a.) Implement the above constructor and selectors using only *one* list to represent your gameboard. So for example, if you had a 2 x 2 board where all four items are 0, then the corresponding board should be represented by [0, 0, 0, 0]. Visually this is equivalent to the following:

```
0  0
0  0
```

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

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?

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)
```

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

Dictionaries

1.) Write a function 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: {}}}}}”””
```

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

```
   def countVal(dl, key):
        return ___________________
```

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
b) t * 2
c) t[1:-1]
d) s < t
e) t.append(s)

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."""

2. def height(t):
   """Returns the height of a tree."""

3. def square_tree(t):
   """Returns a tree with the square of every element in t."""

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
```

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

(c)
```
>>> lst2 = lst1
>>> lst1.append(4)
>>> lst1
```

(d)
```
>>> lst2
```

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

(f)
```
>>> lst1
```

(g) `>>> lst2`

(h) `>>> lst2 is lst1`

2. Draw the box and pointer diagram that results from executing the following code.
[Note: This question is to help you understand this concept but will not be tested on the exam]

```
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)
```

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
```

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
```

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 ______________ variables.

A variable declared nonlocal can't already exist in the ________________ frame.

A nonlocal variable must be declared in a _______________ 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 ____________: # higher order function

                ________________________ # nonlocal variables

                n = ____________________

                print(___________)
        return __________________

    return _________________________
```

Exceptions

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