1 Sequences

Questions

1.1 What would Python display?

```python
lst = [1, 2, 3, 4, 5]
lst[1:3]  # \begin{solution} [2, 3] \end{solution}
lst[0:len(lst)]  # \begin{solution} [1, 2, 3, 4, 5] \end{solution}
lst[-4:]  # \begin{solution} [2, 3, 4, 5] \end{solution}
lst[3:]  # \begin{solution} [4, 5] \end{solution}
lst[1:4:2]  # \begin{solution} [2, 4] \end{solution}
lst[2:4:2]  # \begin{solution} [2, 4] \end{solution}
lst[3:4:2]  # \begin{solution} [1, 3] \end{solution}
lst[1::2]  # \begin{solution} [2, 4] \end{solution}
lst[::2]  # \begin{solution} [2, 4] \end{solution}
lst[::1]  # \begin{solution} [5, 4, 3, 2, 1] \end{solution}
lst + 100  # Error (These aren't numpy arrays)
```
\begin{solution}
1st3 = [[1], [2], [3]]
1st + 1st3
\begin{solution}
[1, 2, 3, 4, 5, [1], [2], [3]]
\end{solution}
\end{solution}
1.2 Draw the environment diagram that results from running the code below

```python
def reverse(lst):
    if len(lst) <= 1:
        return lst
    return reverse(lst[1:]) + [lst[0]]

lst = [1, [2, 3], 4]
rev = reverse(lst)
```

1.3 Implement a function `map_mut` that takes a list as an argument and maps a function `f` onto each element of the list. You should mutate the original lists, without creating any new lists. Do NOT return anything.

```python
def map_mut(f, L):
    for i in range(len(L)):
        L[i] = f(L[i])
```

```
def map_mut(f, L):
    >>> L = [1, 2, 3, 4]
    >>> map_mut(lambda x: x**2, L)
    >>> L
    [1, 4, 9, 16]
```

1.4 Check your understanding

1. When copying the list, when are you copying a pointer of the list vs. copying the actual value inside of a list?
We copy pointers when we refer to a list within a list or another object and we copy the actual values of the list when the item inside that box is a primitive. We also copy pointers of the entire list when we assign variables to that list because we copy pointers for objects.

2 How would you make a deep copy of a list?

Recurse through the list and for every element in the list that is also a list object, copy, the elements of the list object into your copy. In other words, we create a function deep_copy(lst) which takes in a list. We go through each element of that lst, and if the element is type list, we call deep_copy on that sublist. We eventually go through the entire lst that we pass in, and return our deep copy list back. OR: Import copy, copy.deepcopy()
2 Mutability

Questions

2.1 Name two data types that are mutable. What does it mean to be mutable?

Dictionaries, Lists. Being mutable means we can modify them after they’ve been created.

2.2 Name at least two data types at are not mutable.

Tuples, functions, int, float

2.3 Will the following code error? If so, why?

\begin{verbatim}
a = 1
b = 2
dt = {a: 1, b: 2}
\end{verbatim}

\begin{solution}
No -- a and b are both immutable, so we can use them as Dictionary keys.
\end{solution}

\begin{verbatim}
a = [1]
b = [2]
dt = {a: 1, b: 2}
\end{verbatim}

\begin{solution}
Yes -- a and b are mutable, so we can’t use them as Dictionary keys.
\end{solution}

2.4 Fill in the output and draw a box-and-pointer diagram for the following code. If an error occurs, write Error, but include all output displayed before the error.

\begin{verbatim}
a = [1, [2, 3], 4]
c = a[1]
c
\end{verbatim}

\begin{solution}
[2, 3]
\end{solution}

\begin{verbatim}
a.append(c)
a
\end{verbatim}

\begin{solution}
[1, [2, 3], 4, [2, 3]]
\end{solution}

\begin{verbatim}
c[0] = 0
c
\end{verbatim}

\begin{solution}
[0, 3]
\end{solution}

\begin{verbatim}
a
\end{verbatim}

\begin{solution}
[1, [0, 3], 4, [0, 3]]
\end{solution}
2.5 Check your understanding:

1. What is the difference between the append function, extend function, and the `+` operator?

The append and extend functions both return a value of none. They just mutate the list that we are currently working on. For example, if we did `a = [1,2,3].append(4)`, `a` would evaluate to None because the return value of the append function is None. However, when we are using the `+` operator, we return the value of the two lists added together. If we did `a = [1,2,3] + [4,5,6]`, we would get that `a` is equal to `[1,2,3,4,5,6]`. The difference between appends and extends is that appends opens up one single space in the list to place the parameter of appends. This allows for appends to take in both numbers and lists. The extends function takes in a list (that we will refer to as `a`) as its parameter and will open `len(a)` number of boxes in the original list that we are extending.

2. Given the below code, answer the following questions: `a = [1, 2, [3, 4], 5]`  
   `b = a[:]`  
   `b[1] = 6`  
   `b[2][0] = 7`

What does `b` evaluate to?
b = [1,6, [7, 4], 5]

What does a evaluate to? Are a and b the same? Please explain your reasoning.

A = [1,2, [7, 4], 5].

A, B are not the same. Because lists are mutable, when you assign b to a shallow copy of a, you are also copying the pointers to lists within a. Thus, that is why nested elements inside a list changed in both arrays, but all the other elements were unaffected by changes to the shallow copy.
3 Data Abstraction

Questions

3.1 What are the two types of functions necessary to make an Abstract Data Type? What do they do?

Constructors make the ADT.
Selectors take instances of the ADT and output relevant information stored in it.

3.2 Assume that rational, numer, denom, and gcd run without error and behave as described below. Can you identify where the abstraction barrier is broken? Come up with a scenario where this code runs without error and a scenario where this code would stop working.

```python
def rational(num, den): # Returns a rational number ADT
    # implementation not shown

def numer(x): # Returns the numerator of the given rational
    # implementation not shown

def denom(x): # Returns the denominator of the given rational
    # implementation not shown

def gcd(a, b): # Returns the GCD of two numbers
    # implementation not shown

def simplify(f1): # Simplifies a rational number
    g = gcd(f1[0], f1[1])
    return rational(numer(f1) // g, denom(f1) // g)

def multiply(f1, f2): # Multiplies and simplifies two rational numbers
    r = rational(numer(f1) * numer(f2), denom(f1) * denom(f2))
    return simplify(r)
```

```python
x = rational(1, 2)
y = rational(2, 3)
multiply(x, y)
```

The abstraction barrier is broken inside simplify(f1) when calling gcd(f1[0], f1[1]). This assumes rational returns a type that can be indexed through. i.e. This would work if rational returned a list. However, this would not work if rational returned a dictionary.

The correct implementation of simplify would be

```python
def simplify(f1):
    g = gcd(numer(x), denom(x))
    return rational(numer(f1) // g, denom(f1) // g)
```

3.3 Check your understanding
1 How do we know what we are breaking an abstraction barrier?

Put simply, a Data Abstraction Violation is when you bypass the constructors and selectors for an ADT, and directly use how its implemented in the rest of your code, thus assuming that its implementation will not change. We cannot assume we know how the ADT is constructed except by using constructors and likewise, we cannot assume we know how to access details of our ADT except through selectors. The details are supposed to be abstracted away by the constructors and selectors. If we bypass the constructors and selectors and access the details directly, any small change to the implementation of our ADT could break our entire program.

2 What are the benefits to Data Abstraction?

With a correct implementation of these data types, it makes for more readable code because:
- You can make constructors and selectors have more informative names.
- Makes collaboration easier.
- Other programmers don’t have to worry about implementation details.
- Prevents error propagation.
- Fixes errors in a single function rather than all over your program.
4 Trees

Questions

4.1 Fill in this implementation of the Tree ADT.

def tree(label, branches = []):
    for b in branches:
        assert is_tree(b), 'branches must be trees'
    return [label] + list(branches)

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for b in branches(tree):
        if not is_tree(b):
            return False
    return True

def label(tree):
    \begin{solution}
    \begin{verbatim}
    return tree[0]
    \end{verbatim}
    \end{solution}

def branches(tree):
    \begin{solution}
    \begin{verbatim}
    return tree[1:]
    \end{verbatim}
    \end{solution}

def is_leaf(tree):
    \begin{solution}
    \begin{verbatim}
    return not branches(tree)
    \end{verbatim}
    \end{solution}

4.2 A min-heap is a tree with the special property that every node's value is less than or equal to the values of all of its children. For example, the following tree is a min-heap:

```
    1
   / | \ 
  5 3 6
 | / \ 
7 9 4
```
However, the following tree is not a min-heap because the node with value 3 has a value greater than one of its children:

```
        1
       / | \
      5 3 6
     / | \
    7 9 2
```
Write a function `is_min_heap` that takes a tree and returns True if the tree is a min-heap and False otherwise.

```python
def is_min_heap(t):
    for b in branches(t):
        if label(t) > label(b) or not is_min_heap(b):
            return False
    return True
```

4.3 Write a function `largest_product_path` that finds the largest product path possible. A product path is defined as the product of all nodes between the root and a leaf. The function takes a tree as its parameter. Assume all nodes have a non-negative value.

```
3
/ | \
7 8 4
| |
2 1
```

For example, calling `largest_product_path` on the above tree would return 42, since 3 * 7 * 2 is the largest product path.

```python
def largest_product_path(tree):
    if not tree:
        return 0
    elif is_leaf(tree):
        return label(tree)
    else:
        paths = [largest_product_path(t) for t in branches(tree)]
        return label(tree) * max(paths)
```

4.4 Check your understanding:
1. Given the first tree in 4.2, write the corresponding python call to create the tree:

tree(1, [tree(5, [tree(7)]), tree(3, [tree(9), tree(4)]), tree(6)])

2. What is the benefit of using a tree as a data structure, rather than a list or linked list?

Trees can be organized more effectively (in the case of a binary search tree), and in the general case you can tell whether or not an element is in that tree faster than if it were in a linked list or a list. You can also search faster in a BST. Trees can also be used to express hierarchy.

3. Below is the function contains, which takes in an input of a tree, t and a value, e. The function returns true if e exists as a label inside t. However, the function does not work properly, debug this code and find the error(s).

```python
def contains(t, e):
    if is_leaf(t):
        return False
    elif e == label(t):
        return True
    else:
        for b in branches(t):
            return contains(b, e)
    return True
```

Errors:
- One should check the label before checking if the tree is a leaf, in case the leaf contains the value of the tree.
- In the for loop it should not return contains(b,e), but rather return True, if contains(b,e) evaluates to true.
- At the end, if we have searched through all the trees, we should return False if the value, e is not found.

4. Implement a function max_tree, which takes a tree t. It returns a new tree with the exact same structure as t; at each node in the new tree, the entry is the largest number that is contained in that node’s subtrees or the corresponding node in t.

```python
def max_tree(t):
    >>> max_tree(tree(1, [tree(5, [tree(7)]), tree(3, [tree(9), tree(4)]), tree(6)]))
    tree(9, [tree(7, [tree(7)]), tree(9, [tree(9), tree(4)]), tree(6)])
    if __________:
        return ______________
    else:
        new_branches= ______________________________
        new_label = ______________________________
        return __________________
```
def max_tree(t):
    if is_leaf(t):
        return tree(root(t))
    else:
        new_branches = [max_tree(b) for b in branches(t)]
        new_label = max([root(t)] + [root(s) for s in new_branches])
        return tree(new_label, new_branches)
4.5 Challenge Question: The level-order traversal of a tree is defined as visiting the nodes in each level of a tree before moving onto the nodes in the next level. For example, the level order of the following tree is: 3 7 8 4

```
3
/|
7 8 4
```

Write a function `level_order` that takes in a tree as the parameter and returns a list of the values of the nodes in level order.

```python
def level_order(tree):
    # iterative solution
    if not tree:
        return []
    current_level, next_level = [label(tree)], [tree]
    while next_level:
        find_next = []
        for b in next_level:
            find_next.extend(branches(b))
        next_level = find_next
        current_level.extend([label(t) for t in next_level])
    return current_level
```

4.6 Challenge Question: Write a function `all_paths` which will return a list of lists of all the possible paths of an input tree, t. When the function is called on the same tree as the problem above, the function would return: [[3, 7], [3, 8], [3, 4]]

```python
def all_paths(t):
    if ____________:
        _______________
    else:
        _______________
        _______________
        _______________
```

```python
def all_paths(t):
    if is_leaf(t):
        return [[label(t)]]
    else:
        total = []
        for b in branches(t):
            for path in all_paths(b):
                total.append([label(t)] + path)
        return total
```