Guerrilla Section 3: Sequences, Data Abstraction, and Trees

Instructions
Form a group of 3-4. Start on Question 0. Check off with a lab assistant when everyone in your group understands how to solve Question 0. Repeat for Question 1, 2, etc. You're not allowed to move on from a question until everyone in your group is comfortable with all exercises in the section. You are allowed to use any and all resources at your disposal, including the interpreter, lecture notes and slides, discussion notes, and labs. You may consult the lab assistants, but only after you have asked everyone else in your group. The purpose of this section is to have all the students working together to learn the material.

Sequences

Question 0
Fill out what python would display at each step if applicable.

Note: (keep in mind list slicing creates a brand new list, does not modify existing list)

i.
>>> lst = [1, 2, 3, 4, 5]
>>> lst[1:3]

>>> lst[0:len(lst)]

>>> lst[-4:]

>>> lst[:3]

>>> lst[3:]

>>> lst[:]


ii. **Hint:** You can also specify the increment step-size for slicing. The notation is lst[start:end:step]. Remember that a negative step size changes the default start and end.

```python
>>> lst[1:4:2]
>>> lst[0:4:3]

>>> lst[:4:2]

>>> lst[1::2]

>>> lst[::2]

>>> lst[::-1]

>>> lst2 = [6, 1, 0, 7]
>>> lst + lst2

>>> lst + 100

>>> lst3 = [[1], [2], [3]]
>>> lst + lst3
```

**Question 1**

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)
```
**EXTRA: Question 2**

Write `combine_skipper`, which takes in a function $f$ and list $lst$ and outputs a list. When this function takes in a list $lst$, it looks at the list in chunks of four and applies $f$ to the first two elements in every set of four elements and replaces the first element with the output of the function $f$ applied to the two elements as the first value and the index of the second item of the original two elements as the second value of the new two elements. $f$ takes in a list and outputs a value. [Assume the length of $lst$ will always be divisible by four]

```python
def combine_skipper(f, lst):
    while n < len(lst) // 4:
        return lst
    return
```

Don’t proceed until everyone in your group has finished and understands all exercises!
Mutability

**Question 0**

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

b. Name two data types that are not mutable.

**Question 1**

a. Will the following code error? Why?

```python
>>> a = 1
>>> b = 2
>>> dt = {a: 1, b: 2}
```

b. Will the following code error? Why?

```python
>>> a = [1]
>>> b = [2]
>>> dt = {a: 1, b: 2}
```
Question 2

a. 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.

```python
>>> a = [1, [2, 3], 4]
>>> c = a[1]
>>> c

_____________________________

>>> a.append(c)
>>> a

_____________________________

>>> c[0] = 0
>>> c

_____________________________

>>> a

_____________________________

>>> a.extend(c)
>>> a

_____________________________
```
b. 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.

```python
>>> lst = [5, 6, 7]
>>> risk = [5, 6, 7]
>>> lst, risk = risk, lst
>>> lst is risk
____________

>>> mist = risk
>>> risk = risk[0:4]
>>> mist.insert(-1, 99)
>>> risk[-1]
_____________

# Hint: Try drawing the result of [y + 1 for y in mist] first.
>>> risk = [x for x in [y + 1 for y in mist] if x % 10 != 0]
>>> risk
_____________

>>> er = [1, 2]
>>> er.extend(risk.pop())
_____________

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!
Data Abstraction

Question 1

a. Why are Abstract Data Types useful?

b. What are the two types of functions necessary to make an Abstract Data Type? Describe what they do.

c. What is a Data Abstraction Violation?

d. Why is it a terrible sin to commit a Data Abstraction Violation?
Question 2

In lecture, we discussed the rational data type, which represents fractions with the following methods:

• `rational(n, d)` - constructs a rational number with numerator `n`, denominator `d`
• `numer(x)` - returns the numerator of rational number `x`
• `denom(x)` - returns the denominator of rational number `x`

We also presented the following methods that perform operations with rational numbers:

• `add_rationals(x, y)`
• `mul_rationals(x, y)`
• `rationals_are_equal(x, y)`

You’ll soon see that we can do a lot with just these simple methods in the exercises below.

a. Write a function that returns the given rational number `x` raised to positive power `e`.

```python
from math import pow

def rational_pow(x, e):
    """
    >>> r = rational_pow(rational(2, 3), 2)
    >>> numer(r)
    4
    >>> denom(r)
    9

    >>> r2 = rational_pow(rational(9, 72), 0)
    >>> numer(r2)
    1
    >>> denom(r2)
    1
    """
```
b. Implement the following rational number methods.

def inverse_rational(x):
    """ Returns the inverse of the given non-zero rational number
    >>> r = rational(2, 3)
    >>> r_inv = inverse_rational(r)
    >>> numer(r_inv)
    3
    >>> denom(r_inv)
    2
    >>> r2 = rational_pow(rational(3, 4), 2)
    >>> r2_inv = inverse_rational(r2)
    >>> numer(r2_inv)
    16
    >>> denom(r2_inv)
    9
    """
def div_rationals(x, y):  
    # Hint: Use functions defined in Question 2
    """ Returns x / y for given rational x and non-zero rational y
    >>> r1 = rational(2, 3)
    >>> r2 = rational(3, 2)
    >>> div_rationals(r1, r2)
    [4, 9]
    >>> div_rationals(r1, r1)
    [6, 6]
    """

c. The irrational number \( e \approx 2.718 \) can be generated from an infinite series. Let’s try calculating it using our rational number data type! The mathematical formula is as follows:

\[
e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots
\]

Write a function approx_e that returns a rational number approximation of \( e \) to \( \text{iter} \) amount of iterations. We’ve provided a factorial function.

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

def approx_e(iter):
Question 3
Assume that rational, numer, and denom, run without error and work like the ADT defined in Question 2. 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):
    # Multiples and simplifies two rational numbers
    r = rational(numer(f1) * numer(f2), denom(f1) * denom(f2))
    return simplify(r)

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

STOP!

Don’t proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!
Trees

Question 0

a. Fill in this implementation of a tree:

```python
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):

def branches(tree):

def is_leaf(tree):
```
b. 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:

![Min-Heap Example](image)

However, the following tree is *not* a min-heap because the node with value 3 has a value greater than one of its children:

![Not-Min-Heap Example](image)

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):
```
c. 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 nonnegative value.

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):
    ""
    >>> largest_product_path(None)
    0
    >>> largest_product_path(tree(3))
    3
    >>> t = tree(3, [tree(7, [tree(2)]), tree(8, [tree(1)]), tree(4)])
    >>> largest_product_path(t)
    42
    ""
    if not ____________:
        return 0
    elif is_leaf(tree):
        return _________________
    else:
        paths = [ ________________________________ ]
        return _________________
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off
**Challenge Question (Optional)**

*Come back after finishing everything!*

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,

$$\begin{array}{ccccc}
& & 3 & & \\
& 7 & & 8 & 4 \\
& & & \end{array}$$

Level-order: 3 7 8 4

**a.** Write a function `print_level_sorted` 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):
    
    >>> t = tree(3, [tree(7, [tree(2, [tree(8), tree(1)]), tree(5)])])
    >>> level_order(t)
    [3 7 5 2 8 1]
    >>> level_order(tree(3))
    [3]
    >>> level_order(None)
    []
    
    if not _______________
        return []
    current_level, next_level = [label(tree)], [tree]
    while _______________
        find_next = []
        for _______________ in _______________
            ________________.extend(_____________________________)
        next_level = find_next
        current_level.extend(______________________________)
    return current_level
```