Object Oriented Programming

Question 0
0a) What is the relationship between a class and an ADT?

0b) Define the following:

- Instance

- Class

- Class Attribute

- Instance Attribute

- Bound Method
Question 1: What would Python Print?

class Foo():
    x = 'bam'
    def __init__(self, x):
        self.x = x

    def baz(self):
        return self.x

class Bar(Foo):
    x = 'boom'
    def __init__(self, x):
        super().__init__(self, 'er' + x)
    def baz(self):
        return Bar.x + Foo.baz(self)

foo = Foo('boo')

>>> foo.x

>>> foo.x

>>> foo.baz()

>>> Foo.baz()

>>> Foo.baz(foo)

>>> bar = Bar('ang')

>>> Bar.x

>>> bar.x

>>> bar.baz()
Question 2: Attend Class

class Student:
    def __init__(self, subjects):
        self.current_units = 16
        self.subjects_to_take = subjects
        self.subjects_learned = {}
        self.partner = None

    def learn(self, subject, units):
        print("I just learned about " + subject)
        self.subjects_learned[subject] = units
        self.current_units -= units

    def make_friends(self):
        if len(self.subjects_to_take) > 3:
            print("Whoa! I need more help!")
            self.partner = Student(self.subjects_to_take[1:]
        else:
            print("I’m on my own now!")
            self.partner = None

    def take_course(self):
        course = self.subjects_to_take.pop()
        self.learn(course, 4)
        if self.partner:
            print("I need to switch this up!")
            self.partner = self.partner.partner
        if not self.partner:
            print("I have failed to make a friend :("

What Would Python Print?
It may be helpful to draw an object diagram (You can draw this however you’d like) representing
Tim, and all his attributes (be sure to keep track of all partners and their respective attributes).
The diagram is not required.

>>> tim = Student(["Chem1A", "Bio1B", "CS61A", "CS70", "CogSci1"])
>>> tim.make_friends()

>>> print(tim.subjects_to_take)

>>> tim.partner.make_friends()
>>> tim.take_course()

>>> tim.partner.take_course()

>>> tim.take_course()

>>> tim.make_friends()
Nonlocal

Question 3: Draw an environment diagram each of the following:

3a) 

```python
ore = "settlers"
def sheep(wood):
    def ore(wheat):
        nonlocal ore
        ore = wheat
        ore(wood)
    return ore
sheep(lambda wood: ore)("wheat")
```

3b) 

```python
aang = 120
def airbend(zuko):
aang = 2
def katara(aang):
    nonlocal zuko
    zuko = lambda sokka : aang + 4
    return aang
if zuko(10) == 1:
katara(aang + 9)
return zuko(airbend)
airbend(lambda x: aang + 1)
```
**Question 4**

Write `make_max_finder`, which takes in no arguments but returns a function which takes in a list. The function it returns should return the maximum value it's been called on so far, including the current list and any previous list. You can assume that any list this function takes in will be nonempty and contain only non-negative values.

```python
def make_max_finder():
    
    >>> m = make_max_finder()
    >>> m([5, 6, 7])
    7
    >>> m([1, 2, 3])
    7
    >>> m([9])
    9
    >>> m2 = make_max_finder()
    >>> m2([1])
    1
    
    """
```
Mutable Trees

Question 8
Use following definition of a tree to answer the questions below:
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
            self.branches = list(branches)

    def __repr__(self):
        if self.branches:
            branches_str = ', ' + repr(self.branches)
        else:
            branches_str = ''
        return 'Tree({0}{1})'.format(self.entry, branches_str)

    def is_leaf(self): # a leaf has no branches
        return len(self.branches) == 0
8a) Define `filter_tree`, which takes in a tree `t` and one argument predicate function `fn`. It should mutate the tree by removing all branches of any node where calling `fn` on its label returns `False`. In addition, if this node is not the root of the tree, it should remove that node from the tree as well.

```python
def filter_tree(t, fn):
    """
    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(6, [Tree(7)])])
    >>> filter_tree(t, lambda x: x % 2 != 0)
    >>> t
    tree(1, [Tree(3)])
    >>> t2 = Tree(2, [Tree(3), Tree(4), Tree(5)])
    >>> filter_tree(t2, lambda x: x != 2)
    >>> t2
    Tree(2)
    """
    if not fn(t.label):
        ______________________
    else:
        for ___________________
            if __________________:
                ______________________
            else:
                ______________________
```

8b) Fill in the definition for `nth_level_tree_map`, which also takes in a function and a tree, but mutates the tree by applying the function to every `n`th level in the tree, where the root is the 0th level.

```python
def nth_level_tree_map(fn, tree, n):
    """Mutates a tree by mapping a function all the elements of a tree.
    >>> tree = Tree(1, [Tree(7, [Tree(3), Tree(4), Tree(5)]),
                       Tree(2, [Tree(6), Tree(4)])])
    >>> nth_level_tree_map(lambda x: x + 1, tree, 2)
    >>> tree
    Tree(2, [Tree(7, [Tree(4), Tree(5), Tree(6)]),
             Tree(2, [Tree(7), Tree(5)])])
    """
```
Extra Challenge Question 9: Photosynthesis
9a) Fill in the methods below, so that the classes interact correctly according to the documentation (make sure to keep track of all the counters!).

```python
>>> p = Plant()
>>> p.height
1
>>> p.materials
[]
>>> p.absorb()
>>> p.materials
[|Sugar|]
>>> Sugar.sugars_created
1
>>> p.leaf.sugars_used
0
>>> p.grow()
>>> p.materials
[]
>>> p.height
2
>>> p.leaf.sugars_used
1

class Plant:
    def __init__(self):
        """A Plant has a Leaf, a list of sugars created so far, and an initial height of 1""
        ###Write your code here###

    def absorb(self):
        """Calls the leaf to create sugar""
        ###Write your code here###

    def grow(self):
        """A Plant uses all of its sugars, each of which increases its height by 1""
```
class Leaf:
    def __init__(self, plant):  # Source is a Plant instance
        """A Leaf is initially alive, and keeps track of how many sugars it has created"""
        ###Write your code here###

    def absorb(self):
        """If this Leaf is alive, a Sugar is added to the plant’s list of sugars"""
        if self.alive:
            ###Write your code here###

class Sugar:
    sugars_created = 0

def __init__(self, leaf, plant):
    ###Write your code here###

def activate(self):
    """A sugar is used, then removed from the Plant which contains it"""
    ###Write your code here###

def __repr__(self):
    return '|Sugar|'
9b) (Optional -- only do if time at the end!) Let's make this a little more realistic by giving these objects ages. Modify the code above to satisfy the following conditions. See the doctest for further guidance.

1) Every plant and leaf should have an age, but sugar does not age. Plants have a lifetime of 20 time units, and leaves have a lifetime of 2 time units.

2) Time advances by one unit at the end of a call to a plant's absorb or grow method.
3) Every time a leaf dies, it spawns a new leaf on the plant. When a plant dies, its leaf dies, and the plant becomes a zombie plant--no longer subject to time. Zombie plants do not age or die.

4) Every time a generation of leaves dies for a zombie plant, twice as many leaves rise from the organic matter of its ancestors--defying scientific explanation.

```python
>>> p = Plant()
>>> p.age
0
>>> p.leaves
[Leaf]
>>> p.leaves[0].age
0
>>> p.age = 18
>>> p.age
18
>>> p.height
1
>>> p.absorb()
>>> p.materials
[Sugar]
>>> p.age
19
>>> p.leaves[0].age
1
>>> p.grow()
>>> p.age
20
>>> p.is_zombie
True
>>> p.leaves
[Leaf, Leaf]
>>> p.leaves[0].age
```
>>> p.absorb()
>>> p.age
20

""
You will only need to make changes to the Plant and Leaf classes.

class Plant:
    def __init__(self):
        """A Plant has a Leaf, a list of sugars created so far,
        and an initial height of 1""
        self.materials = []
        self.height = 1
        ###Write your code here###

    def absorb(self):
        """Calls the leaf to create sugar""
        ###Write your code here###

    def grow(self):
        """A Plant uses all of its sugars, each of which increases
        its height by 1""
        for sugar in self.materials:
            sugar.activate()
            self.height += 1
        ###Write your code here###

    def death(self):
        ###Write your code here###
class Leaf:
    def __init__(self, plant): # plant is a Plant instance
        """A Leaf is initially alive, and keeps track of how many sugars it has created""
        self.alive = True
        self.sugars_used = 0
        self.plant = plant
        ###Write your code here###
    
    def absorb(self):
        """If this Leaf is alive, a Sugar is added to the plant’s list of sugars""
        if self.alive:
            self.plant.materials.append(Sugar(self, self.plant))
        ###Write your code here###
    
    def death(self):
        ###Write your code here###
    
    def __repr__(self):
        return '|Leaf|'
**Linked Lists**

0a) What is a linked list? Why do we consider it a naturally recursive structure?

0b) Draw a box and pointer diagram for the following:

\[ \text{Link('c', Link(6, Link(1, Link('a'))), Link('s'))} \]

Question 1: The `Link` class can represent lists with cycles. That is, a list may contain itself as a sublist.

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.rest.rest.rest = s
>>> s.rest.rest.rest.rest.rest.first
3
```

Implement `has_cycle` that returns whether its argument, a `Link` instance, contains a cycle. There are two ways to do this, both iteratively, either with two pointers or keeping track of `Link` objects we've seen already. Try to come up with both!

```python
def has_cycle(link):
    ""
    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle(s)
    True
    """
```
def seq_in_link(link, sub_link):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> lnk2 = Link(1, Link(3))
    >>> lnk3 = Link(4, Link(3, Link(2, Link(1))))
    >>> seq_in_link(lnk1, lnk2)
    True
    >>> seq_in_link(lnk1, lnk3)
    False
    """
    if sub_link is Link.empty:
        __________________________

    if link is Link.empty:
        __________________________

    if link.first == ________________
        __________________________
    else:
        __________________________
1. Generator WWPD

```python
>>> def g(n):
    while n > 0:
        if n % 2 == 0:
            yield n
        else:
            print('odd')
        n -= 1

>>> t = g(4)
>>> t

>>> next(t)

>>> n

>>> t = g(next(t) + 5)

>>> next(t)
```

2. Write a generator function `gen_inf` that returns a generator which yields all the numbers in the provided list one by one in an infinite loop. Write your solution to the right.

```python
>>> t = gen_inf([3, 4, 5])
>>> def gen_inf(lst):

>>> next(t)
3
>>> next(t)
4
>>> next(t)
5
>>> next(t)
3
>>> next(t)
4
```
3. Write a function `nested_gen` which, when given a nested list of iterables (including generators) `lst`, will return a generator that yields all elements nested within `lst` in order. Assume you have already implemented `is_iter`, which takes in one argument and returns True if the passed in value is an iterable and False if it is not.

```python
def nested_gen(lst):
    '''
    >>> a = [1, 2, 3]
    >>> def g(lst):
    >>>     for i in lst:
    >>>         yield i
    >>> b = g([10, 11, 12])
    >>> c = g([b])
    >>> lst = [a, c, [[2]]]
    >>> list(nested_gen(lst))
    [1, 2, 3, 10, 11, 12, 2]
    '''

    if _________________________________:
        ________________
    else:
        ________________
```
4. Write a function that, when given an iterable lst, returns a generator object. This generator should iterate over every element of lst, checking each element to see if it has been changed to a different value from when lst was originally passed into the generator function. If an element has been changed, the generator should yield it. If the length of lst is changed to a different value from when it was passed into the function, and next is called on the generator, the generator should stop iteration.

```python
def mutated_gen(lst):
    '''
    >>> lst = [1, 2, 3, 4, 5]
    >>> gen = mutated_gen(lst)
    >>> lst[1] = 7
    >>> next(gen)
    7
    >>> lst[0] = 5
    >>> lst[2] = 3
    >>> lst[3] = 9
    >>> lst[4] = 2
    >>> next(gen)
    9
    >>> lst.append(6)
    >>> next(gen)
    StopIteration Exception
    '''

    curr = __________
    while __________:
        if __________:
            break
        else:
            __________
            yield __________

    return __________
```
Growth

Question 0

What are the runtimes of the following?

def one(n):
    if 1 == 1:
        return None
    else:
        return n

a) \Theta(1)  
   b) \Theta(n)  
   c) \Theta(2^n)

b) \Theta(n)  
   c) \Theta(n^2)


def two(n):
    for i in range(n):
        print(n)

a) \Theta(1)  
   b) \Theta(n)  
   c) \Theta(2^n)

b) \Theta(n)  
   c) \Theta(n^2)


def three(n):
    while n > 0:
        n = n // 2

a) \Theta(1)  
   b) \Theta(n)  
   c) \Theta(2^n)

b) \Theta(n)  
   c) \Theta(n^2)


def four(n):
    for i in range(n):
        for j in range(i):
            print(str(i), str(j))

a) \Theta(1)  
   b) \Theta(n)  
   c) \Theta(2^n)

b) \Theta(n)  
   c) \Theta(n^2)