Guerrilla Section 4: Object Oriented Programming, Nonlocal, Linked Lists, Iterators & Generators, Mutable Trees, and Growth

Instructions

Form a group of 3-4. Start on Question 1. Check off with a lab assistant when everyone in your group understands how to solve Question 1. Repeat for Question 2, 3, etc. **You are not allowed to move on from a question until you check off with a lab assistant.** 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.**

Object Oriented Programming

Question 0

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

In general, we can think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

There are two different layers to the abstract data type:

1) The program layer, which uses the data, and
2) The concrete data representation that is independent of the programs that use the data. The only communication between the two layers is through selectors and constructors that implement the abstract data in terms of the concrete representation.

Classes are a way to implement an Abstract Data Type. But, ADTs can also be created using a collection of functions, as shown by the rational number example. (See Composing Programs 2.2)

0b) Define the following:

> Instance - A specific object created from a class. Each instance shares class attributes and stores the same methods and attributes. But the values of the attributes are independent of other instances of the class. For example, all humans have two eyes but the color of their eyes may vary from person to person.

Class - Template for all objects whose type is that class that defines attributes and methods that an object of this type has.

Class Attribute - A static value that can be accessed by any instance of the class and is shared among all instances of the class.

Instance Attribute - A field or property value associated with that specific instance of the object.

Bound Method - A function is coupled with the object on which that method will be invoked.This means that when we invoke the bound method, the instance is automatically passed in as the first argument.

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

foo = Foo('boo')
>>> Foo.x
'bam'
>>> foo.x
'boo'
>>> foo.baz()
'boo'
>>> Foo.baz()
Error
>>> Foo.baz(foo)
'boo'
>>> bar = Bar('ang')
>>> Bar.x
'boom'
```

```
>>> bar.x
'erang'
>>> bar.baz()
'boomerang'
```

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 will 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.

The pythontutor can be found via this link: https://goo.gl/y22hNU

Another diagram

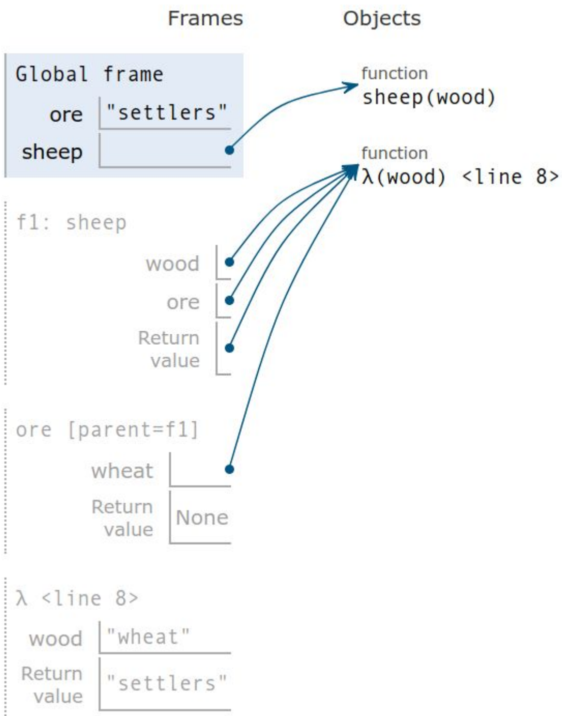
```
>>> tim = Student([“Chem1A”, “Bio1B”, “CS61A”, “CS70”, “CogSci1”])

>>> tim.make_friends()
Whoa! I need more help!

>>> print(tim.subjects_to_take)
[“Chem1A”, “Bio1B”, “CS61A”, “CS70”, “CogSci1”]

>>> tim.partner.make_friends()
Whoa! I need more help!

>>> tim.take_course()
I just learned about CogSci1
I need to switch this up!

>>> tim.partner.take_course()
I just learned about CogSci1

>>> tim.take_course()
I just learned about CS70
I need to switch this up!
I have failed to make a friend :(

>>> tim.make_friends()
I’m on my own now!
```

Mutable Functions/Nonlocal

Question 3

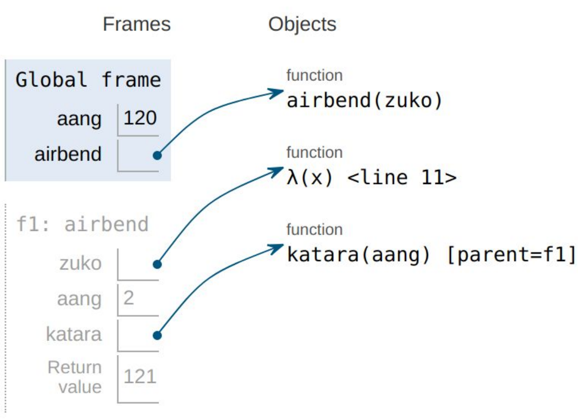
3a)
```
ore = "settlers"
def sheep(wood):
    def ore(wheat):
        nonlocal ore
        ore = wheat
    ore(wood)
    return ore
sheep(lambda wood: ore)("wheat")
```

Solution:



3b)
```
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)
```

Solution:



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.

```
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
    """
    max_so_far = 0
    def find_max_overall(lst):
        nonlocal max_so_far
        if max(lst) > max_so_far:
            max_so_far = max(lst)
        return max_so_far
    return find_max_overall
```

Mutable Trees

Question 8

Given the following definition of a tree, fill in the implementation of tree_map, which takes in a function and a tree, and maps that function across every element in the tree

```
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

# ASSUME THIS IS DEFINED FOR ALL TESTS BELOW
```

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.

```
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):
        t.branches = []
    else:
        for b in t.branches[:]:
            if not fn(b.label):
                t.branches.remove(b)
            else:
                filter_tree(b, fn)
```

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 nth level in the tree, where the root is the 0th level.

```
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)])])
   """
     def helper(tree, level):
          if level % n == 0:
               tree.label = fn(tree.label)
          for b in tree.branches:
               helper(b, level + 1)
     helper(tree, 0)
```

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!).

```
"""
>>> 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###
        self.leaf = Leaf(self)
        self.materials = []        #list of Sugar instances
        self.height = 1

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

    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

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"""
        self.alive = True
        self.sugars_used = 0
        self.plant = plant

    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))

class Sugar:
    sugars_created = 0

    def __init__(self, leaf, plant):
        self.leaf = leaf
        self.plant = plant
        Sugar.sugars_created += 1

    def activate(self):
        """A sugar is used, then removed from the Plant which
        contains it"""
        self.leaf.sugars_used += 1
        self.plant.materials.remove(self)

    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.

```
"""
>>> 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
```

```
0
>>> p.absorb()
>>> p.age
20
"""
```

The changed and added portions are in red. There are no changes made to the Sugar class, so I didn't include it below.

```
class Plant:
      def __init__(self):
           """A Plant has a list of leaves, a list of sugars created
           so far, and an initial height of 1.(Keep in mind, Plant
           class may need some other necessary attributes to achieve
           the requirement.)"""
            self.leaves = [Leaf(self)]
            self.materials = []
            self.height = 1
            self.age = 0
            self.is_zombie = False

      def absorb(self):
            """Calls each leaf the Plant has to create sugars"""
            for leaf in self.leaves:
                  leaf.absorb()
            if not self.is_zombie:
                  self.age += 1
             if self.age >= 20:
                        self.death()

      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
                  if not self.is_zombie:
                        self.age += 1
                        if self.age >= 20:
                              self.death()

      def death(self):
            self.is_zombie = True
```

```
            old_leaves = self.leaves[:]
            for leaf in old_leaves:
                leaf.death()


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
        self.age = 0

    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))
            self.age += 1
        if self.age >= 2:
            self.death()

    def death(self):
        self.alive = False
        self.plant.leaves.remove(self)
        self.plant.leaves.append(Leaf(self.plant))
        if self.plant.is_zombie:
            self.plant.leaves.append(Leaf(self.plant))

    def __repr__(self):
        return '|Leaf|'
```

Linked Lists

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

A linked list is a data structure with a first and a rest, where the first is some arbitrary element and the rest MUST be another linked list

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

```
Link('c', Link(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.

```
>>> 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!

```
def has_cycle(link):
    """
    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle(s)
    True
    """
    Solution 1:
    tortoise = link
    hare = link.rest
    while tortoise.rest and hare.rest and hare.rest.rest:
        if tortoise is hare:
            return True
        tortoise = tortoise.rest
        hare = hare.rest.rest
    return False
```

```
Solution 2:
    seen = []
    while link.rest:
        if link in seen:
            return True
        seen.append(link)
        link = link.rest
    return False
```

Question 2: Fill in the following function, which checks to see if a particular sequence of items in wone linked list, `sub_link` can be found in another linked list `link` (the items have to be in order, but not necessarily consecutive).

```
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:
        return True
    if link is Link.empty:
        return False
    if link.first == sub_link.first:
        return seq_in_link(link.rest, sub_link.rest)
    else:
        return seq_in_link(link.rest, sub_link)
```

Iterators & Generators

1. Generator WWPD

```
>>> def g(n):
        while n > 0:
            if n % 2 == 0:
                yield n
            else:
                print('odd')
            n -= 1
>>> t = g(4)
>>> t
Generator Object

>>> next(t)
4

>>> n
NameError: name 'n' is not defined

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

>>> next(t)
odd
6
```

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.

```
>>> t = gen_inf([3, 4, 5])
>>> next(t)
3
>>> next(t)
4
>>> next(t)
5
>>> next(t)
3
>>> next(t)
4
```

```
def gen_inf(lst):
  while True:
    for elem in lst:
      yield elem

def gen_inf(lst):
  while True:
     yield from iter(lst)
```

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.

```
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]
     '''

     for elem in lst:
          if is_iter(elem):
               yield from nested_gen(elem)
          else:
               yield elem
     (solution using try / except instead of is_iter:)
     def nested_gen(lst):
          for elem in lst:
               try:
                    iter(elem)
                    yield from nested_gen(elem)
          except TypeError:
               yield elem
```

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.

Alternative wording: Write the mutated_gen function which given a list lst, returns a generator that only yields values that have been changed from their original value.
(or yields elements from the list that have been changed from their original value since lst was first passed in as an argument for mutated_gen)
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.

```
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

    >>> lst2 = [1, 2, 3, 4, 5]
    >>> gen2 = mutated_gen(lst2)
    >>> lst2 = [2, 3, 4, 5, 6]
    >>> next(gen)
    StopIteration Exception #the list that the operand was
evaluated
                              to has not been changed.

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

```
>>> gen3 = mutated_gen(lst3)
>>> lst3.pop()
>>> next(gen)
StopIteration Exception #the length of the list that was passed
                                in was changed
>>> lst4 = [[1], 2 , 3]
>>> gen4 = mutated_gen(lst4)
>>> lst4[0] = [1]
>>> next(gen)
StopIteration Exception
'''

original = list(lst)
def gen_maker(original, lst):
     curr = 0
     while curr < len(original):
          if len(original) != len(lst):
               break
          else:
               if original[curr] != lst[curr]:
                    yield lst[curr]
          curr += 1
return gen_maker(original, lst)
```

Growth

Question 0

What are the runtimes of the following?

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

a. **θ(1)** b) θ(log n) c) θ(n) d) $\theta(n^2)$ e) $\theta(2^n)$

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

a. θ(1) b) θ(log n) **c) θ(n)** d) $\theta(n^2)$ e) $\theta(2^n)$

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

a. θ(1) **b) θ(log n)** c) θ(n) d) $\theta(n^2)$ e) $\theta(2^n)$

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

a. θ(1) b) θ(log n) c) θ(n) **d) $\theta(n^2)$** e) $\theta(2^n)$

STOP!

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