Guerrilla Section 5: Object Oriented Programming, Nonlocal & Mutable Trees

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?

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):
          Foo.__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()
```

STOP!

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

Mutable Functions/Nonlocal

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

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

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)
```

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
      """
```

STOP!

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

Question 5: What Would Scheme Print?

```
(define pie 1)
(define (apple pie)
      (define macaron (+ pie 1))
      (define (lemon bar)
            (set! pie bar)
            (set! macaron (* bar 2)))
      (define tart (lambda () (cons pie (cons macaron nil))))
      (cons tart (cons lemon nil)))
```

```
> (define cheese (apple 5))

> (define choco (car cheese))

> (define cake (car (cdr cheese)))

> pie

> choco

> (define taffy (choco))

> (car taffy)

> (cdr taffy)

> (print (cake 9))

> pie

> (define taffy (choco))

> (car taffy)

> (cdr taffy)
```

Question 6

The CS61A TAs are worried that students find calling functions to be too boring. To make things more interesting, they write a higher order function excite that converts boring functions into more exciting functions.

Help them complete the definition of `excite`. It takes in three arguments:

1. `boring-fn`: a function that takes one argument
2. `fun-fn`: a function that takes two arguments, the second of which is always an integer
3. `n`: an integer

Once called with these three arguments, `excite` should return an exciting function, which takes one argument and does the following:

- Every `n`-th time the `excite` function is called, it calls `fun_fn` with the provided argument and the number of times the `excite` function has ever been called. It returns the result of this call.
- Otherwise, calling the `excite` function should return the value of calling `boring_fn` with the provided argument.

Complete the definition of `excite` in the provided space below.

```
; Doctests
scm> (define boring (lambda (name) (begin (print name) (print 'walked-the-dog))))
boring
scm> (define fun (lambda (name i) (begin (print name) (print 'won) (print i) (print
'new-cars!))))
fun
scm> (define great-deal (excite boring fun 2))
great-deal
scm> (great-deal 'mitas)
mitas
walked-the-dog
scm> (great-deal 'mitas)
mitas
won
2
new-cars
scm> (great-deal 'mitas)
mitas
walked-the-dog
scm> (great-deal 'mitas)
mitas
won
4
new-cars
```

```
(define (excite boring-fn fun-fn n)
      (define count _______________)
      (define (exciting arg)
            ____________________________________________________________
            (if (= (_____________________________________________) 0)
                ____________________________________________________
                ____________________________________________________
      ____________________________________________________________)
```

Question 7

Implement the `func-set` function, which returns two functions in a list that together represent a set. Both the `add` and `has` functions return whether a value is already in the set. The `add` function also adds its argument value to the set. You may assign to only one name in the assignment statement.

```
; Doctests
scm> (define result (func-set))
scm> (define add (car result))
scm> (define has (car (cdr result))
scm> (add 1)
#f
scm> (add 3)
#f
scm> (list (has 1) (has 2) (has 3) (has 4) (has 5))
(#t #f #t #f #f)
scm> (add 3)
#t
scm> (add 2)
#f
scm> (list (has 1) (has 2) (has 3) (has 4) (has 5))
(#t #t #t #f #f)

(define (func-set)
      (define items (lambda (x) false))
      (define (add y)
            (define f items)
            (set! _________________________________)
            __________)
      (cons add (cons _________________________ nil)))
```

STOP!

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

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.

```
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.

```
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)])])
   """
```

STOP!

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

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###



     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"""
        ###Write your code here###

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.

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

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|'
```