WWPD, Nonlocal / List Mutation, Linked Lists, Trees

Computer Science 61A

October 4, 2016

1 WWPD

1.1 Questions

1. Does Jack Like Jackfruits?

   For each of the statements below, write the output displayed by the interactive Python interpreter when the statement is executed. The output may have multiple lines. **No answer requires more than three lines.** If executing a statement results in an error, write 'Error', but include all lines displayed before the error occurs. The first two have been provided as examples.

   Assume that you have started `python3` and executed the following statements:

```
class Fruit:
    ripe = False
    def __init__(self, taste, size):
        self.taste = taste
        self.size = size
        self.ripe = True
    def eat(self, eater):
        print(eater, `eats the', self.name)
        if not self.ripe:
            print(`But it is not ripe!')
        else:
            print(`What a', self.taste, `and', self.size, `
               fruit!')

class Tomato(Fruit):
    name = `tomato'
    def eat(self, eater):
        print(`Adding some sugar first')
        self.taste = `sweet'
        Fruit.eat(self, eater)

mystery = Fruit(`tart', `small')
tommy = Tomato(`plain', `normal')
```

```
>>> mystery.taste
'tart'
```

```
>>> mystery.name
Error
```

```
>>> mystery.ripe
```

```
>>> tommy.eat('Brian')
```

```
>>> Tomato.ripe
```

```
>>> Tomato.eat(mystery, 'Marvin')
```

```
>>> Fruit.eat(tommy, 'Brian')
```

```
>>> tommy.name = 'sweet tomato'
>>> Fruit.eat = lambda self, own: print(
...                 self.name, 'is too sweet!')
>>> tommy.eat('Marvin')
```

Solution:

`>>> mystery.taste` `'tart'`	`>>> Tomato.ripe` `False`

`>>> mystery.name`
`Error`
	`>>> Tomato.eat(mystery, 'Marvin')`
	`Adding some sugar first`
	`Error`

`>>> mystery.ripe`
`True`
	`>>> Fruit.eat(tommy, 'Brian')`
	`Brian eats the tomato`
	`What a sweet and normal fruit!`

`>>> tommy.eat('Brian')`
`Adding some sugar first`
`Brian eats the tomato`
`What a sweet and normal fruit!`
	`>>> tommy.name = 'sweet tomato'`
	`>>> Fruit.eat = lambda self, own: print(`
	`...                 self.name, 'is too sweet!')`
	`>>> tommy.eat('Marvin')`
	`Adding some sugar first`
	`sweet tomato is too sweet!`

2 Nonlocal and List Mutation

2.1 Questions

```
def x(lst):
    def y(a):
        _______________
    return y

y = x([1, 2, 3])
y(4)
```

1. Which of these options will mutate `lst`?
   1. lst += [a]
   2. lst = lst + [a]
   3. lst.append(a)
   4. lst.extend([a])

Solution: lst.append(a), lst.extend([a])

```
def x(lst):
    def y(a):
```

```
        nonlocal lst
        _______________
    return y

y = x([1, 2, 3])
y(4)
```

2. Which of these options will mutate `lst`?

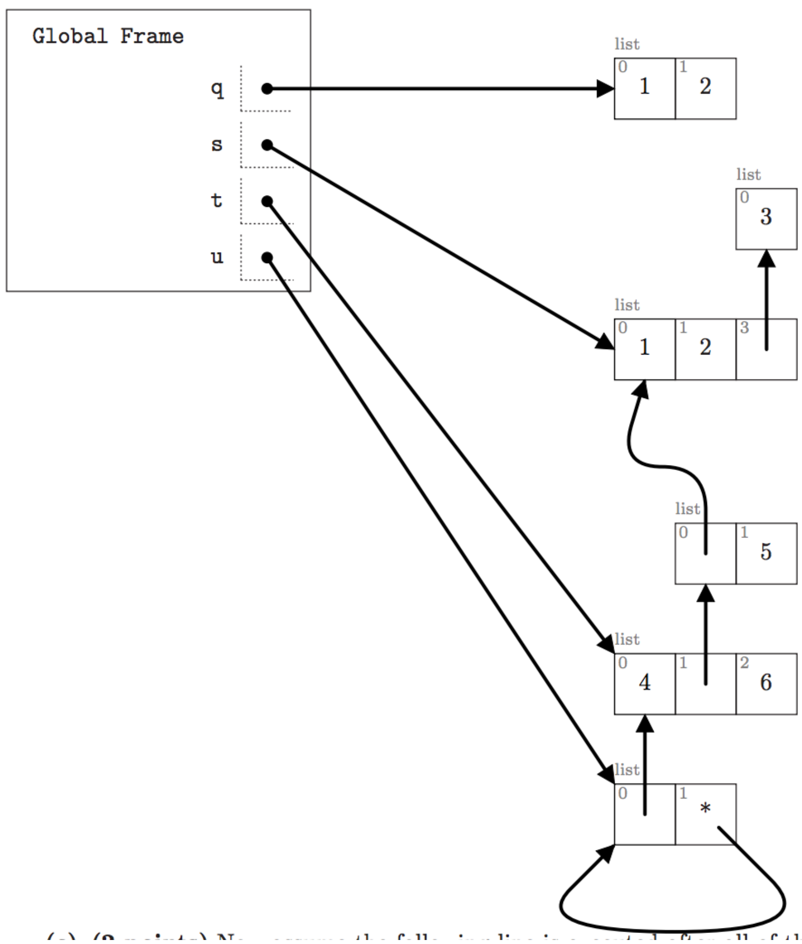
   1. lst += [a]
   2. lst = lst + [a]
   3. lst.append(a)
   4. lst.extend([a])

Solution: lst += [a], lst.append(a), lst.extend([a])

3. Draw the box-and-pointer diagram that results from executing the following code:

```
q = [1 , 2]
s = [1 , 2 , [3]]
t = [4 , [s , 5] , 6]
u = [t ]
```

Solution:



4. Now assume the following line is executed after all of the statements above are executed.

```
u.append(u)
```

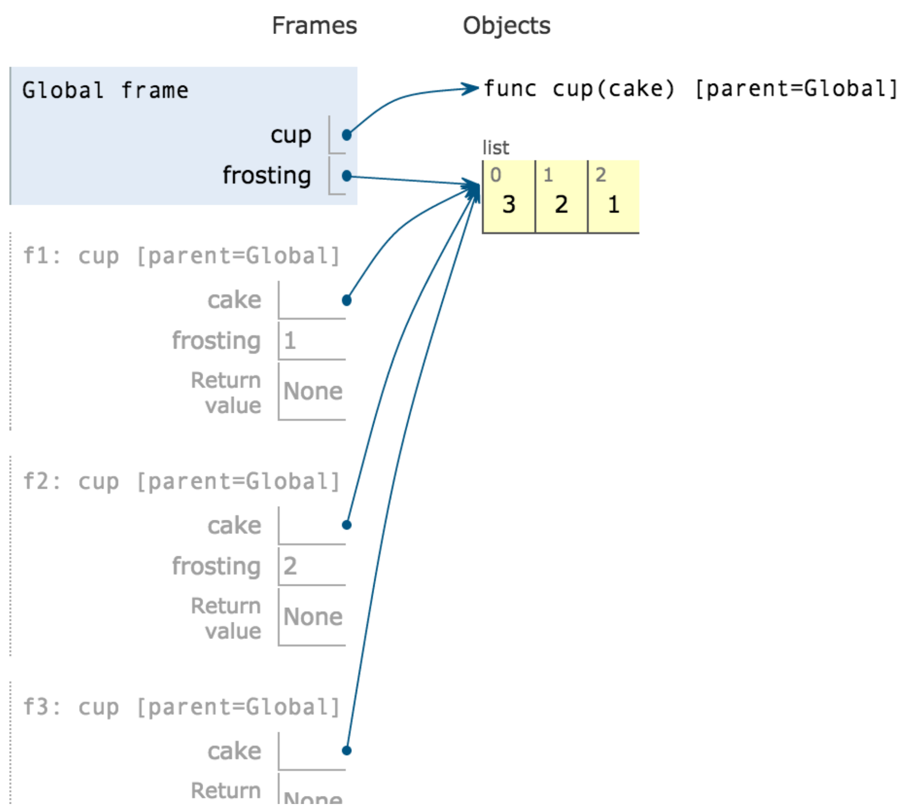
Show the result on your diagram. Mark any added list cells with an asterisk (*) to show what your change.

5. Draw the environment diagram for the code below:

```
def cup(cake):
    if len(cake) != 1:
```

```
        frosting = cake.pop(0)
        cup(cake)
        cake.append(frosting)
frosting = [1, 2, 3]
cup(frosting)
```

Solution:

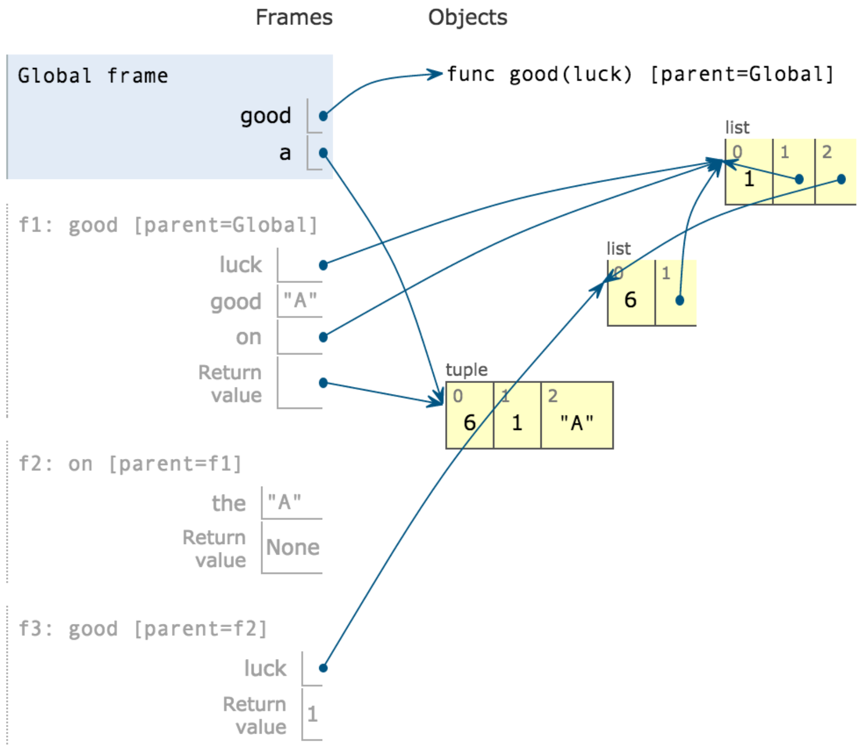


6. Draw the environment diagram for the code below:

```
def good(luck):
    good = 0
    def on(the):
        nonlocal luck, on, good
        on = [luck[0] * luck[1] * luck[2]]
        def good(luck):
            nonlocal good, on
            good = 1
            on, luck[good][good+1] = luck[good], luck
            return good
        on = on + [luck]
        luck[good(on)] = luck
        good = the
    on('A')
    if luck is on:
        return on[2][0], on[0], good
    else:
        return good

a = good([1, 2, 3])
```

Solution:


Frames
Objects
Global frame
good
a
func good(luck) [parent=Global]
list
0
1
1
2
f1: good [parent=Global]
luck
good
"A"
on
Return value
list
0
6
1
tuple
0
6
1
1
2
"A"
f2: on [parent=f1]
the
"A"
Return value
None
f3: good [parent=f2]
luck
Return value
1

3 Linked Lists

3.1 Questions

These questions use the following Linked List implementation

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

1. Implement a `mutating_map` method that takes in a function and applies it to each element in a Linked List. This method should mutate the list in place, replacing each element with the result of applying the function to it. Do not create any new objects. You may assume that the input Linked List contains at least one element.

```
def mutating_map (self, fn):
   """ Mutate this linked list by applying fn to each element
   >>> r = Link(1 , Link (2 , Link (3)))
   >>> r.mutating_map(lambda x: x + 1)
   >>> r
   Link(2 , Link(3 , Link(4)))
   """
   _________________________________

      _________________________________

         _________________________________
```

Solution:

```
self.first = fn ( self.first )
if self.rest != Link.empty:
   self.rest.mutating_map(fn)
```

2. Define the function linked sum that takes in a linked list of positive integers lnk and a non-negative integer total and returns the number of combinations of elements in lnk that sum up to total. You may use each element in lnk zero or more times. See the doctests for details.

```
def linked_sum(lnk, total):
    """Return the number of combinations of elements in lnk
       that
    sum up to total.

    >>> # Four combinations: 1 1 1 1 , 1 1 2 , 1 3 , 2 2
    >>> linked_sum(Link(1, Link(2, Link(3, Link(5)))), 4)
    4
    >>> linked_sum(Link(2, Link(3, Link(5))), 1)
    0
    >>> # One combination: 2 3
    >>> linked_sum(Link(2, Link(4, Link(3))), 5)
    1
    """

    if __________________

        return 1

    elif ___________________

        return 0

    else:

        with_first = ___________________

        without_first = ___________________

        return ___________________
```

Solution:

```
if total == 0:
    return 1
elif lnk == empty or total < 0:
    return 0
else:
    with_first = linked_sum(lnk, total - lnk.first)
    without_first = linked_sum(lnk.rest, total)
    return with_first + without_first
```

4 Trees

4.1 Questions

These questions use the following tree data abstraction.

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

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]
```

1. Given a tree, accumulate the values of the tree. Do not create a new tree.

```
def accumulate_tree(tree):
  if ______________:

    ____________________

  else:

    accumulated = ____________________________________

    ____________________________________

    tree.root += ______________________

    return tree
```

Solution:

```
  if tree.is_leaf():
    return tree
  else:
      accumulated = [accumulate_tree(branch) for branch in
         tree.branches]
```

```
    total = sum([t.root for t in t.branches])
    tree.root += total  # include myself!
    return tree
```

2. Define the function track lineage that takes in a tree of strings family tree and a string name. Assume that there is a unique node with entry name. track lineage returns a list with the entries of the parent and grandparent of that node.1 If the node with entry name does not have a parent or grandparent, return None for that element in the list. See the doctests for details. Do not violate abstraction barriers. You may only use the lines provided. You may not need to fill all the lines.

```
def track_lineage(family_tree, name):
    """Return the entries of the parent and grandparent of
    the node with entry name in family_tree.

    >>> t = tree(`Tytos', [
    ...     tree(`Tywin', [
    ...     tree(`Cersei'), tree(`Jaime'), tree(`Tyrion')
    ...     ]),
    ...     tree(`Kevan', [
    ...     tree(`Lancel'), tree(`Martyn'), tree(`Willem')
    ...     ])])
    >>> track_lineage(t, `Cersei')
    [`Tywin', `Tytos']
    >>> track_lineage(t, `Tywin')
    [`Tytos', None]
    >>> track_lineage(t, `Tytos')
    [None, None]
    """
    def tracker(t, p, gp):

        if ____________________________________________

        _______________________________________________

        for c in children(t):

        _______________________________________________

        _______________________________________________

        _______________________________________________

        _______________________________________________

    return tracker(_____________, _____________, _____________)
```

Solution:
```
def track_lineage(family_tree, name):
    def tracker(t, p, gp):

        if name == entry(t):

            return [p, gp]

        for c in children(t):

            res = tracker(c, entry(t), p)

            if res:

                return res

    return tracker(family_tree, None, None)
```

3. Assuming that track lineage works correctly, define the function are cousins that takes in a tree of strings family tree and two strings name1 and name2 and returns True if the node with entry name1 and the node with entry name2 are cousins in family tree. Assume that there are unique nodes with entries name1 and name2 in family tree. See the doctests for details.

```
def are_cousins(family_tree, name1, name2):
    """Return True if a node with entry name1 is a cousin of a
       node with
    entry name2 in family_tree.

    >>> are_cousins(t, `Kevan', `Tytos')  # same tree as
       before
    False
    >>> are_cousins(t, `Cersei', `Lancel')
    True
    >>> are_cousins(t, `Jaime', `Lancel')
    True
    >>> are_cousins(t, `Jaime', `Tyrion')
    False
    """
```

Solution:

```
p1, gp1 = track_lineage(family_tree, name1)

p2, gp2 = track_lineage(family_tree, name2)

return p1 != p2 and gp1 is not None and gp1 == gp2
```