CS 61A Fall 2020 — Linked Lists, Trees, Representation

Discussion 8: October 21, 2020 Solutions

1 Representation - A Note on Str and Repr

There are two main ways to produce the "string" of an object in Python: `str()` and `repr()`. While the two are similar, they are used for different purposes. `str()` is used to describe the object to the end user in a "Human-readable" form, while `repr()` can be thought of as a "Computer-readable" form mainly used for debugging and development.

When we define a class in Python, `str()` and `repr()` are both built-in functions for the class. We can call an object's `str()` and `repr()` by using their respective functions. These functions can be invoked by calling `repr(obj)` or `str(obj)` rather than the dot notation format `obj.__repr__()` or `obj.__str__()`. In addition, the `print()` function calls the `str()` function of the object, while simply calling the object in interactive mode calls the `repr()` function.

Here's an example:

```
class Test:
    def __str__(self):
        return "str"
    def __repr__(self):
        return "repr"

>>> a = Test()
>>> str(a)
'str'
>>> repr(a)
'repr'
>>> print(a)
str
>>> a
repr
```

Questions

1.1 What would Python display? Feel free to use the environment diagram template below to help with visualization.

```
class A():
    def __init__(self, x):
        self.x = x
    def __repr__(self):
         return self.x
    def __str__(self):
         return self.x * 2

class B():
    def __init__(self):
         print("boo!")
         self.a = []
    def add_a(self, a):
         self.a.append(a)
    def __repr__(self):
         print(len(self.a))
         ret = ""
         for a in self.a:
             ret += str(a)
         return ret
```

```
>>> A("one")
```

one

```
>>> print(A("one"))
```

oneone

```
>>> repr(A("two"))
```

'two'

```
>>> b = B()
```

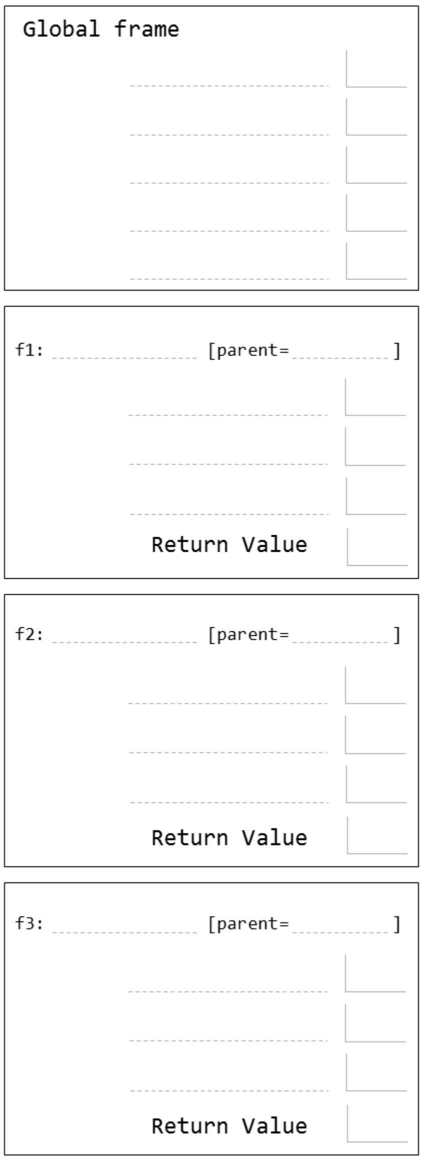
boo!

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

```
>>> b.add_a(A("a"))
>>> b.add_a(A("b"))
>>> b
```

2

aabb



2 Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a `first` value and the `rest` of the linked list.

To check if a linked list is an empty linked list, compare it against the class attribute `Link.empty`:

```
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

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

    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Questions

2.1 Write a function that takes in a a linked list and returns the sum of all its elements. You may assume all elements in `lnk` are integers.

```
def sum_nums(lnk):
    """
    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """


    if lnk == Link.empty:
        return 0
    return lnk.first + sum_nums(lnk.rest)
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

2.2 Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
    # Note: you might not need all lines in this skeleton code
    ___________________ = ___________
    for _______________________________________:
        if __________________________________________:
                _________________________________
        _________________________________
    ________________________________________________________
    ________________________________________________________
```

Recursive solution:

```
    product = 1
    for lnk in lst_of_lnks:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lnks_rests = [lnk.rest for lnk in lst_of_lnks]
    return Link(product, multiply_lnks(lst_of_lnks_rests))
```

For our base case, if we detect that any of the lists in the list of `Links` is empty, we can return the empty linked list as we're not going to multiply anything.

Otherwise, we compute the product of all the firsts in our list of `Links`. Then, the subproblem we use here is the rest of all the linked lists in our list of Links. Remember that the result of calling `multiply_lnks` will be a linked list! We'll use the product we've built so far as the first item in the returned `Link`, and then the result of the recursive call as the rest of that `Link`.

Iterative solution:

```
import operator
from functools import reduce
def prod(factors):
     return reduce(operator.mul, factors, 1)

head = Link.empty
tail = head
while Link.empty not in lst_of_lnks:
    all_prod = prod([l.first for l in lst_of_lnks])
    if head is Link.empty:
        head = Link(all_prod)
        tail = head
    else:
        tail.rest = Link(all_prod)
        tail = tail.rest
    lst_of_lnks = [l.rest for l in lst_of_lnks]
return head
```

The iterative solution is a bit more involved than the recursive solution. Instead of building the list "backwards" as in the recursive solution (because of the order that the recursive calls result in, the last item in our list will be finished first), we'll build the resulting linked list as we go along.

We use `head` and `tail` to track the front and end of the new linked list we're creating. Our stopping condition for the loop is if any of the `Links` in our list of `Links` runs out of items.

Finally, there's some special handling for the first item. We need to update both head and tail in that case. Otherwise, we just append to the end of our list using tail, and update tail.

2.3 **Tutorial:** Write a recursive function `flip_two` that takes as input a linked list `lnk` and mutates `lnk` so that every pair is flipped.

```
def flip_two(lnk):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
```

Recursive solution:

```
    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
    flip_two(lnk.rest.rest)
```

If there's only a single item (or no item) to flip, then we're done.

Otherwise, we swap the contents of the first and second items in the list. Since we've handled the first two items, we then need to recurse on

Although the question explicitly asks for a recursive solution, there is also a fairly similar iterative solution:

```
    while lnk is not Link.empty and lnk.rest is not Link.empty:
        lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
        lnk = lnk.rest.rest
```

We will advance `lnk` until we see there are no more items or there is only one more Link object to process. Processing each `Link` involves swapping the contents of the first and second items in the list (same as the recursive solution).

Notice that the code is remarkably similar to the recursive implementation of `flip_two`.
Video walkthrough

2.4 **Tutorial:** Implement `filter_link`, which takes in a linked list `link` and a function `f` and returns a generator which yields the values of `link` for which `f` returns `True`.

Try to implement this both using a while loop and without using any form of iteration.

```
def filter_link(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> g = filter_link(link, lambda x: x % 2 == 0)
    >>> next(g)
    2
    >>> next(g)
    StopIteration
    >>> list(filter_link(link, lambda x: x % 2 != 0))
    [1, 3]
    """

    while _________________________:

        if ________________________:

            _______________________

        ___________________________
```

```
def filter_link(link, f):
    while link is not Link.empty:
        if f(link.first):
            yield link.first
        link = link.rest
```

3 Trees

Recall the tree abstract data type: a tree is defined as having a label and some branches. Previously, we implemented the tree abstraction using Python lists. Let's look at another implementation using objects instead:

```
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

Notice that with this implementation we can mutate a tree using attribute assignment, which wasn't possible in the previous implementation using lists.

```
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.label = 5
>>> t.label
5
```

Questions

3.1 Define a function `make_even` which takes in a tree `t` whose values are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.label
    2
    >>> t.branches[0].branches[0].label
    4
    """

    if t.label % 2 != 0:
        t.label += 1
    for branch in t.branches:
        make_even(branch)
```

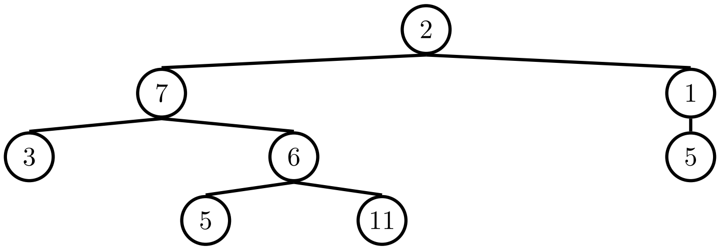
3.2 Define a function `square_tree(t)` that squares every value in the non-empty tree `t`. You can assume that every value is a number.

```
def square_tree(t):
    """Mutates a Tree t by squaring all its elements."""

    t.label = t.label ** 2
    for branch in t.branches:
        square_tree(branch)
```

3.3 **Tutorial:** Define the procedure `find_paths` that, given a Tree `t` and an `entry`, returns a list of lists containing the nodes along each path from the root of `t` to `entry`. You may return the paths in any order.

For instance, for the following tree `tree_ex`, `find_paths` should return:




```
def find_paths(t, entry):
    >>> tree_ex = Tree(2, [Tree(7, [Tree(3), Tree(6, [Tree(5), Tree(11)])]), Tree(1, [Tree(5)])])
    >>> find_paths(tree_ex, 5)
    [[2, 7, 6, 5], [2, 1, 5]]
    >>> find_paths(tree_ex, 12)
    []

    paths = []
    if ____________________________:
        ______________________________________________
    for __________________________________:
        _________________________________________:
            ___________________________________________
    return _______________________
```

```
    paths = []
    if t.label == entry:
        paths.append([t.label])
    for b in t.branches:
        for path in find_paths(b, entry):
            paths.append([t.label] + path)
    return paths
```

Here is an alternate solution that uses a list comprehension instead:

```
    paths = []
    if t.label == entry:
        paths.append([t.label])
    for b in t.branches:
        branch_paths = [[t.label] + path for path in find_paths(b, entry)]
        paths.extend(branch_paths)
    return paths
```

3.4 Write a function that combines the values of two trees `t1` and `t2` together with the `combiner` function. Assume that `t1` and `t2` have identical structure. This function should return a new tree.

Hint: consider using the `zip()` function, which returns an iterator of tuples where the first items of each iterable object passed in form the first tuple, the second items are paired together and form the second tuple, and so on and so forth.

```
def combine_tree(t1, t2, combiner):
    """
    >>> a = Tree(1, [Tree(2, [Tree(3)])])
    >>> b = Tree(4, [Tree(5, [Tree(6)])])
    >>> combined = combine_tree(a, b, mul)
    >>> combined.label
    4
    >>> combined.branches[0].label
    10
    """


    combined = [combine_tree(b1, b2, combiner) for b1, b2
                in zip(t1.branches, t2.branches)]
    return Tree(combiner(t1.label, t2.label), combined)
```

3.5 Implement the `alt_tree_map` function that, given a function and a `Tree`, applies the function to all of the data at every other level of the tree, starting at the root.

```
def alt_tree_map(t, map_fn):
   """
   >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
   >>> negate = lambda x: -x
   >>> alt_tree_map(t, negate)
   Tree(-1, [Tree(2, [Tree(-3)]), Tree(4)])
   """


   def helper(t, depth):
       if depth % 2 == 0:
           label = map_fn(t.label)
       else:
           label = t.label
       branches = [helper(b, depth + 1) for b in t.branches]
       return Tree(label, branches)
   return helper(t, 0)
```

Alternate solution without a helper function:

```
def alt_tree_map(t, map_fn):
    label = map_fn(t.label)
    branches = []
    for b in t.branches:
        next_branches = [alt_tree_map(bb, map_fn) for bb in b.branches]
        branches.append(Tree(b.label, next_branches))
    return Tree(label, branches)
```