Lecture 14: Midterm Review

July 12th, 2022

Tim Tu

Announcements

- Midterm **tomorrow Thursday 7-9 PM**
  - Seat assignments will come out later today / tomorrow morning via email
  - Remote Exam logistics posted on Ed
- Midterm review OH today from 1-5:30 PM
  - Please ask any past midterm questions here!
  - Also MT review OH on Thursday 12-1:30 PM

ADT Trees Review

General tips

- Think about the input, and the output
- Understand
  - that is_leaf() takes a tree
  - a tree with no branches is a leaf
  - branches(t) returns a ***list*** of branches, we can index into branches(t), iterate over branches(t)
    - If a function takes in a tree, passing into it branches(t) will be wrong
- Think it terms of base case and recursive calls
- Assume the recursive calls work
- To do work on whole tree, need to loop over branches for recursive call!
- When creating a tree, we need to have new label and new branches ready before we use the tree constructor

Aggregation

1. Basic examples: max_tree, sum_tree, num_nodes, num_leaves
2. Advanced: odd_row_sum, even_row_sum
    a. Implement `odd_row_sum(t)` and `even_row_sum(t)`, which both take in a tree `t`, and return the sum of the labels on the odd rows of the tree, and the even row of the tree, respectively

```
t = tree(1, [tree(2, [tree(3), tree(4)]), tree(5, [tree(6,
[tree(7, [tree(8)])])])])

print_tree(t)
1
  2
    3
    4
  5
    6
      7
        8
odd_row_sum(t)  == 22 # 1 + 3+4+6 + 8
even_row_sum(t) == 14 # 2+5 + 7
```

Booleans

1. Basic: is_tree, is_even_tree # checks if tree has at least one even label
2. Advanced: is_binary_tree
    a. Implement is_binary_tree(t), which takes in a tree t and returns if each node has exactly 2 branches

Fa20 Midterm 2 Q4 Fork It

https://cs61a.org/exam/fa20/mt2/61a-fa20-mt2.pdf

Creating trees

1. Basic: factorial_tree
2. Advanced: factor_tree
    a. Implement a function `factor_tree(n)` which creates a tree factors for a given positive number input `n`

Su19 Final Q4 Combo Nation

https://cs61a.org/exam/su19/final/61a-su19-final.pdfYou may assume the two trees have the same shape (that is, each node has the same number of children).

```
def apply_tree(fn_tree, val_tree):
    """ Creates a new tree by applying each function stored in fn_tree
    to the corresponding labels in val_tree
    >>> double = lambda x: x*2
    >>> square = lambda x: x**2
    >>> identity = lambda x: x
    >>> t1 = tree(double, [tree(square), tree(identity)])
    >>> t2 = tree(6, [tree(2), tree(10)])
    >>> t3 = apply_tree(t1, t2)
    >>> print_tree(t3)
    12
      4
      10
    """

    __________
    __________
    for __________:
        __________
    return __________
```

Su19 Final Q4 Combo Nation

Definition. A combo of a non-negative integer n is the result of adding or multiplying the digits of n from left to right, starting with 0. For n = 357, combos include 15 = (((0 + 3) + 5) + 7), 35 = (((0 * 3) + 5) * 7), and 0 = (((0 * 3) * 5) * 7), as well as 0, 7, 12, 22, 56, and 105. But 36 = ((0 + 3) * (5 + 7)) is not a combo of 357.

```
def is_combo(n, k):
    """ Is k a combo of n? A combo of a non-negative integer n
    is the result of adding or multiplying the digits of n
    from left to right, starting with 0
    >>> [k for k in range(1000) if is_combo(357, k)]
    [0, 7, 12, 15, 22, 35, 56, 105]
    """
    assert n >= 0 and k >= 0
    if __________:
        return True
    if __________:
        return False
    rest, last = n // 10, n % 10
    added = __________ and is_combo(__________, __________)
    multiplied = __________ and is_combo(__________,__________)
    return added or multiplied
```

Su19 Final Q4 Combo Nation

Implement `make_checker_tree` which takes in a tree, t containing digits as its labels and returns a tree with functions as labels (a function tree). When applied to another tree, the function tree should return a new tree with label as True if the label is a combo of the number formed by concatenating the labels from the root to the corresponding node of t. You may use is_combo in your solution.

```
def make_checker_tree(t, so_far=0):
    """ Returns a function tree that, when applied to another tree,
    will create a new tree where labels are True if the label is a
combination
    of the path in t from the root to its corresponding node.
    >>> t1 = tree(5, [tree(2), tree(1)])
    >>> fn_tree = make_checker_tree(t1)
    >>> t2 = tree(5, [tree(10), tree(7)])
    >>> t3 = apply_tree(fn_tree, t2) #5 is a combo of 5, 10 is a combo of 52,
7 isn't a combo of 51
    >>> print_tree(t3)
    True
      True
      False
    """
    new_path = ____________________________________________________
    branches = ____________________________________________________
    fn = ___________________________________________________________
    return tree(fn, branches)
```

Tree Recursion Review

The Essence of Tree Recursion

- **base case(s)**: usually one or more – when have I found a valid path? an invalid one (i.e. there's no possible way I can end up on a valid path)?
  - *count_stair_ways*: at the top of the staircase / stepped past the top
    - return 1 represents valid path, return 0 represents invalid path
  - *count_partitions*: successfully partitioned n fully / exceeded n with my parts OR run out of parts to use
  - *insect_combinatorics*: hit the top-right corner / gone out-of-bounds
- **recursive calls**: multiple, often each represents a choice
  - *c_s_w*: take 1 step or take 2 steps
  - *count_part*: use a part of size k or don't use any parts of size k
  - *insect_comb*: move right or move up
- **recombination**: some function or operation to construct the answer of your original problem from the answer of your subproblems
  - *c_s_w*, *count_part*, *insect_comb*: total num of ways → sum recursive calls
  - *largest_drop*: biggest difference between two digits → call max

Tips for Tree Recursion

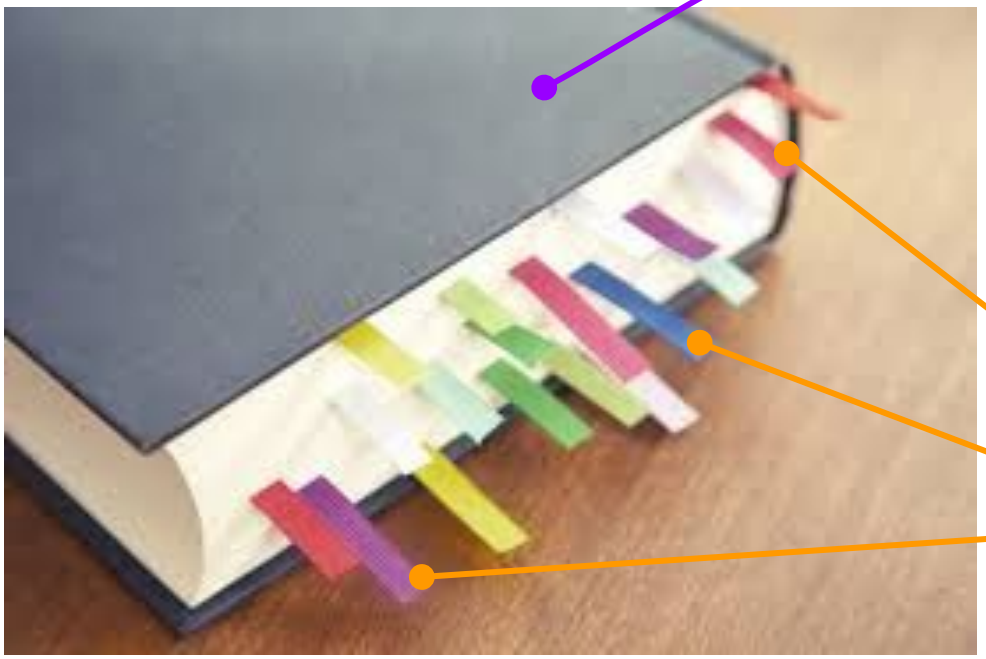
- Structure
  - **base case**: what is the simplest input that we know the answer to? when should we stop recursing?
  - **recursive case**: what smaller inputs am I allowed to compute given my original input (*decision-making*)? how can i use their solution to solve my input (*recursive leap of faith*)?
- Consider **domain**, **range**, and **intended behavior** of your function
  - **domain**: what type(s) and set of values can you take in? the edge(s) of your domain could be your base case(s)
  - **range**: most recursive functions return the same type of value in all cases
    - verify that all return statements have matching return types
  - **intended behavior**: if I were given this problem, how would I solve it? → how can I formalize my steps into general instructions a computer could follow?
- Parse the skeleton
  - What is the high-level purpose of each blank line? Of each variable?
- Problem-solving
  - partial solutions can lead you to the full solution (jigsaw puzzle approach)
  - Write down what you know first → what else can you fill out given that the other blanks have been filled?

Fall 2019 Final Q6a-c

Iterators and Generators Review

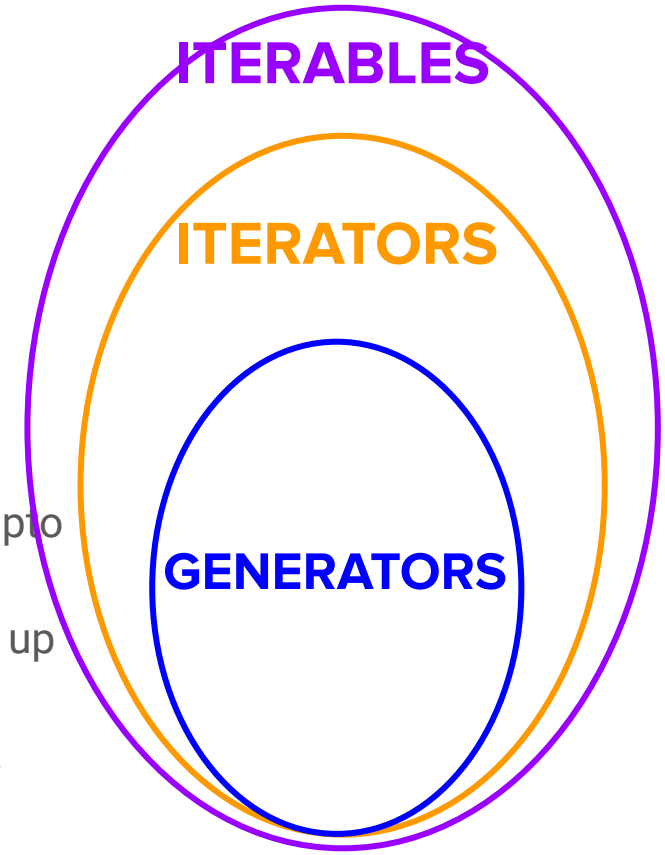
Iterator Tips

- **Iterable (book)**: data type which contains a collection of values which can be processed one by one sequentially. **Ex**. lists, dictionaries, tuples, strings
  - Things you can call for loop on!
- **Iterator (bookmark)**: an object that retrieve values contained in an iterable
- Key functions:
  - **iter(iterable)** : (in most cases) return a new iterator at the top of iterable
    - Exception: **iter(iterator) returns the same iterator**
  - **next(iterator)** : returns the current value in the iterable and move the iterator's position to the next value



- Bookmarks (iterators) remember where they left off in the book (iterable)
- There can be multiple bookmarks (iterators) in one book (iterable)

Generator Tips



Review

- **Generators**: Iterators defined using functions
- **Generator functions** return generator object (iterator)
- **Calling next method on generator object** runs the code upto **yield** statement and returns (yields) value
- **Calling next method on the same generator object** picks up the code from where it left off and runs until the next encounter to **yield** statement (or the end of the function).
- Generators are **Lazily Evaluated.** Only evaluated when needed

Tips

- Think about how to solve the problem **one element at a time.**
- Iterators and generators can use a current value to figure out how to access later values
- Use **yield from** for recursive generators!
- Can have **many iterators to one iterable**

Su18 Final Q7a-b

Sp18 Final Q4a

Mutability Review

Quick Review

What's important to know about lists?

You should know:

- How to construct a new list
- How to index elements out of a list
- How to mutate a list by indexing
- How to take a slice of a list
- How to write a list comprehension, and what they do
- You should be familiar with all the list mutation operations—you don't have to memorize them all, because they'll be on the study guide, but you should feel like you've seen each of them and know what they do
  - `append`, `extend`, `pop`, `remove`, `insert`
- What operations create a new list, and which ones mutate an existing list
- How to represent lists in environment diagrams

Making a copy vs mutating

These will create an entirely new list:

- Taking any slice of a list
  - `a[1:3]`
- Writing a list comprehension
- Concatenating lists
  - `a = a + [3, 2]`

These will mutate a list that already exists

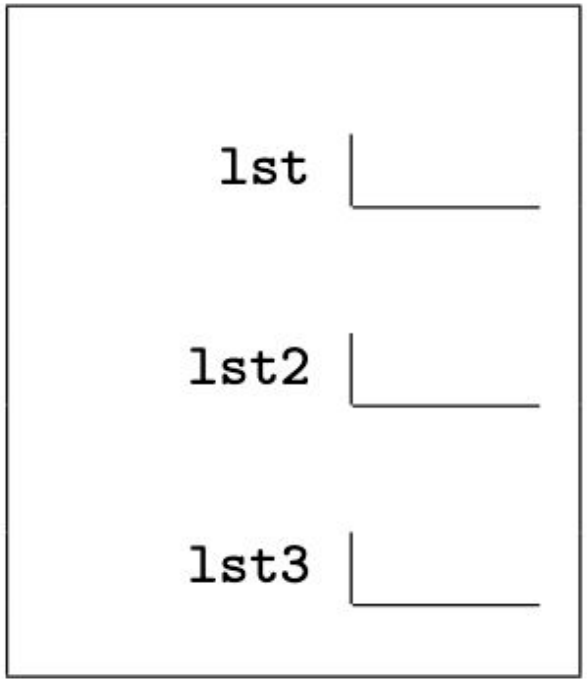
- Any of the mutation functions (see previous slide)
- Bracketing on the right side of an assignment statement
  - `a[0] = 3`
  - `a[1:3] = [3, 4, 5]`
  - **not** `a = [3, 4, 5]`
  - `a += [3, 4, 5]` is a special case in which mutation actually does occur

Lists and Mutation in Environment Diagrams

Su19 MT Q3a

```
lst = [2, 4, lambda: lst]
lst2 = lst
lst = lst[2:]
lst3 = lst[0]()
```

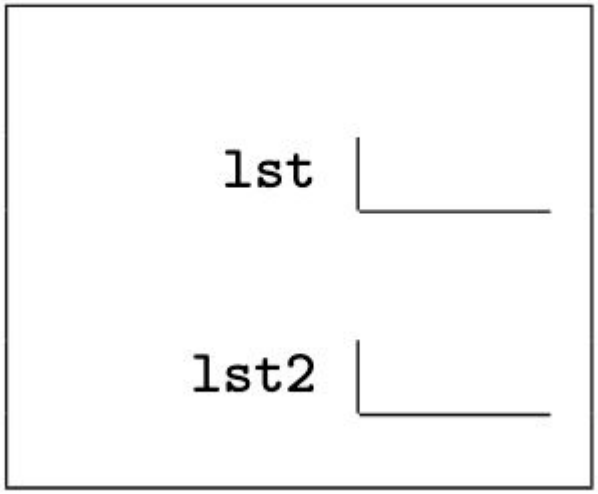
PythonTutor



Su19 MT Q3b

```
lst = [[5], 2, 4, 10]
lst2 = lst[1:3] + lst[:3]
for n in lst2[:2]:
    lst.append(lst2[n])
```

PythonTutor



Su19 MT Q3c

```
lst = ['goodbye', 0, None, 8, 'hello', 1]
while lst.pop():
    x = lst.pop()
    if x:
        lst.pop()
    else:
        lst.append('three')
```

PythonTutor

General tips

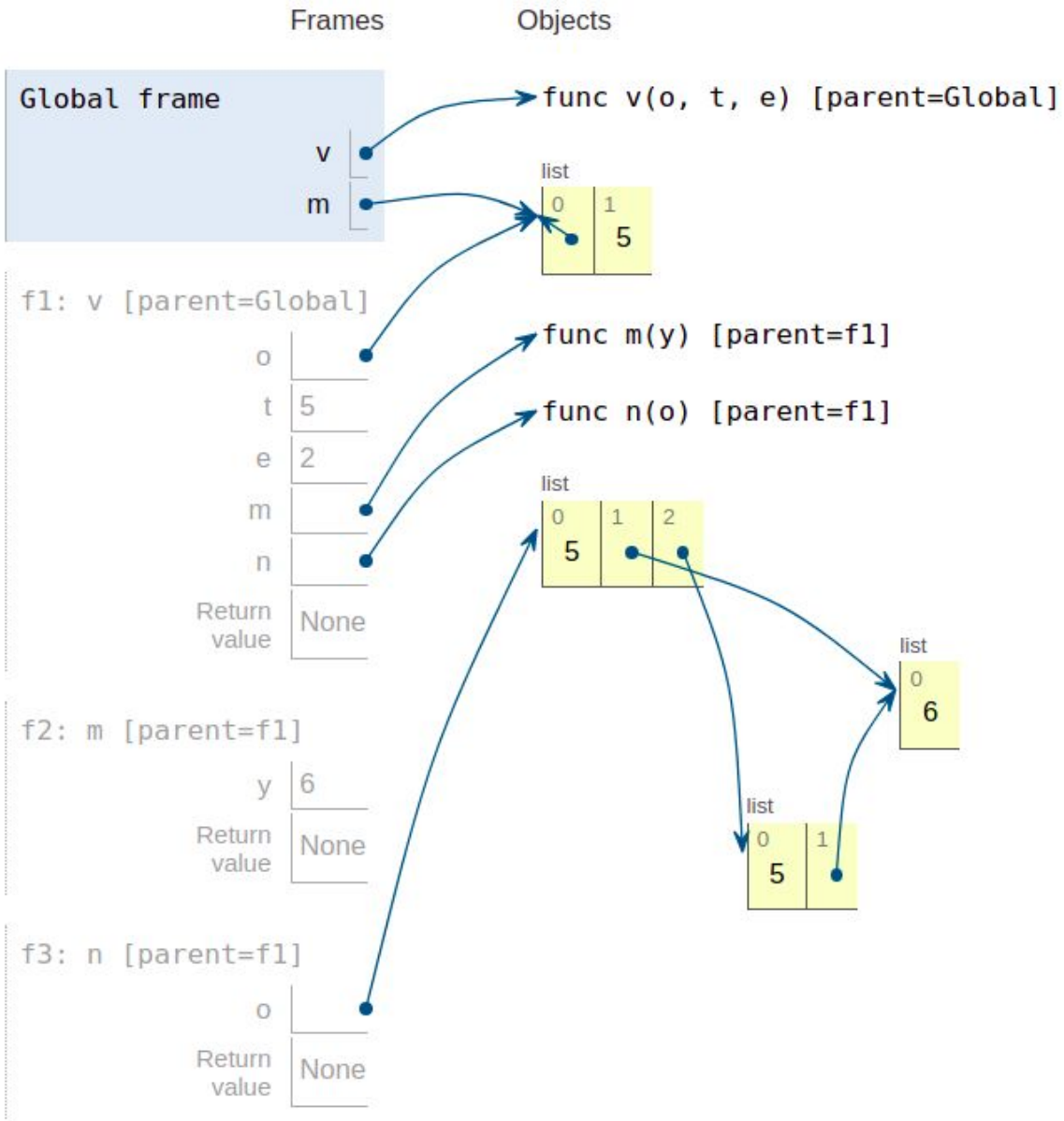
- **Use python tutor!!!!**
  - Fill in the blanks with random/default/generic values.
  - Run through the code line by line to construct your own environment diagram.
  - If your environment diagram doesn't match the image, go back and try to fix each blank one by one.
- Understand which methods mutate a list and which create a copy
- Each line of the environment diagram is a clue!
  - Names show what variables you should have
  - Values show what the expressions should evaluate to eventually
  - Frame names show which function is called
  - Frame numbers show order of program flow

You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.

```
def v(o, t, e):
    def m(y):
        _________ #(a)
    def n(o):
        o.append(_____)#(b)
        o.append(_____)#(c)
    m(e)
    n([t])
    e = 2
m = [3, 4]
v(m, 5, 6)
```

Blank (c) choose all that apply

```
o
[o]
list(o)
list([o])
o + []
[o[0], o[1]]
o[:]
```

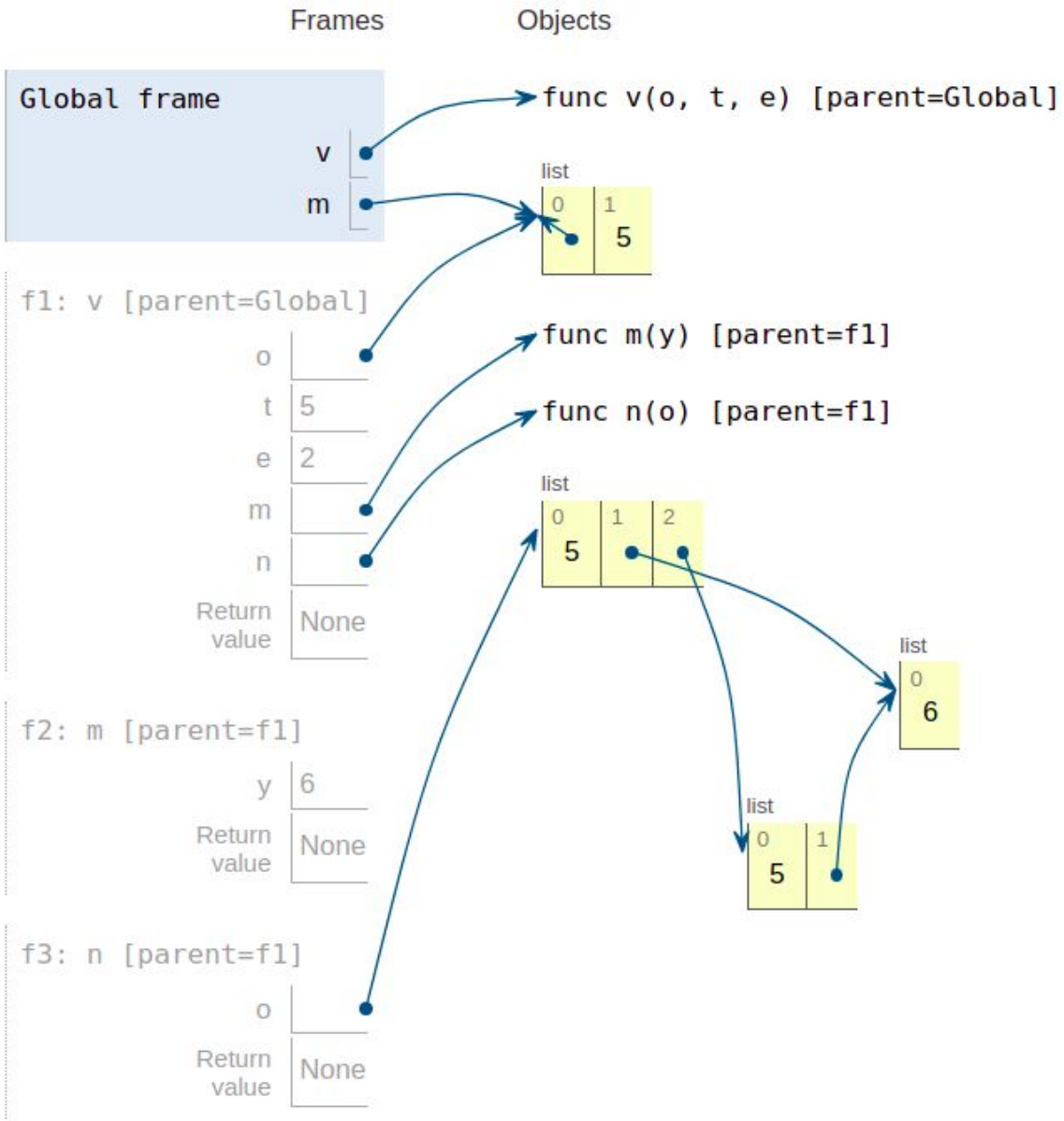


You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.

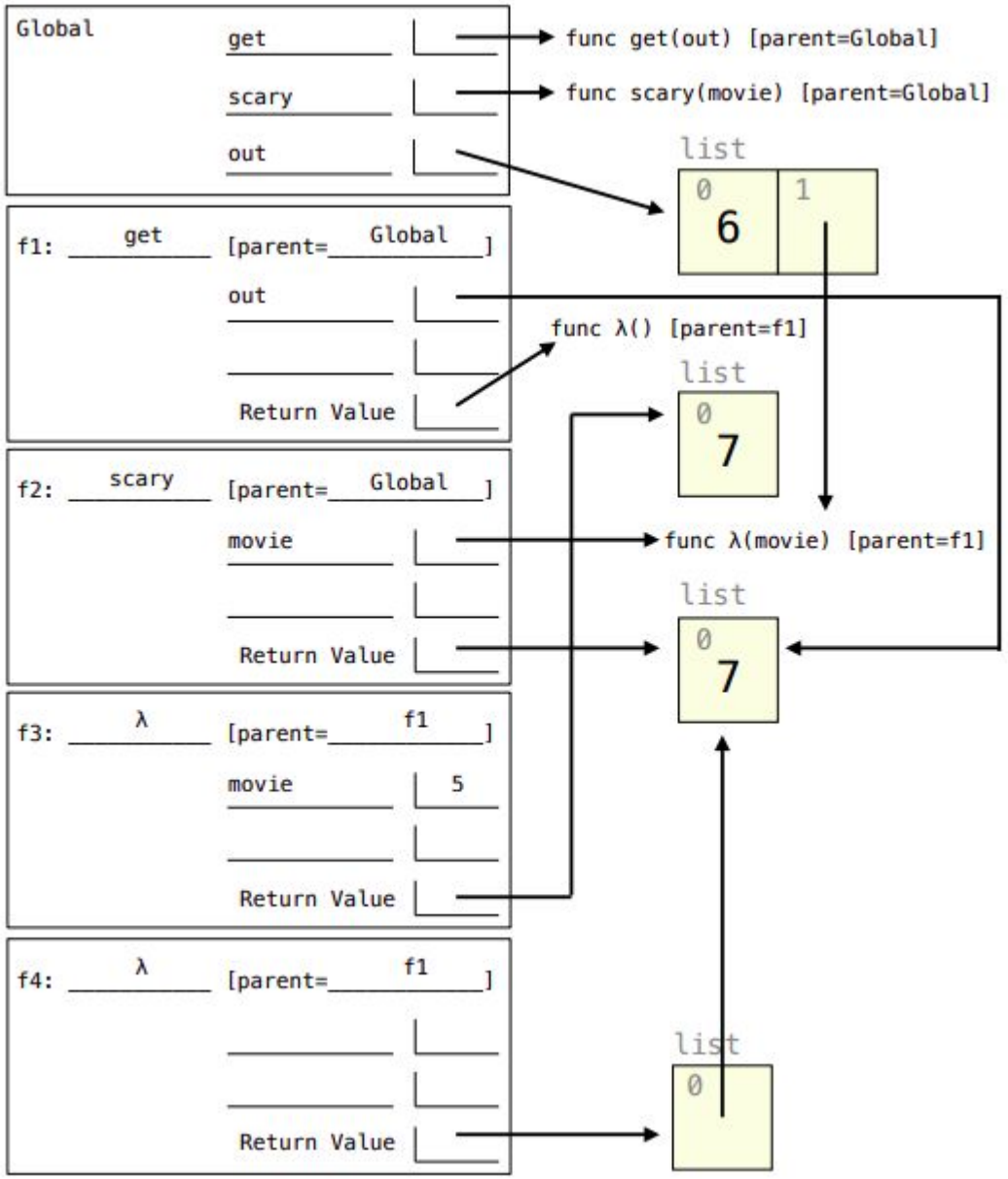
```
def v(o, t, e):
    def m(y):
        o[:] = [o, t] #(a)
    def n(o):
        o.append([e])#(b)
        o.append(o[:])#(c)
    m(e)
    n([t])
    e = 2
m = [3, 4]
v(m, 5, 6)
```

Blank (c) choose all that apply

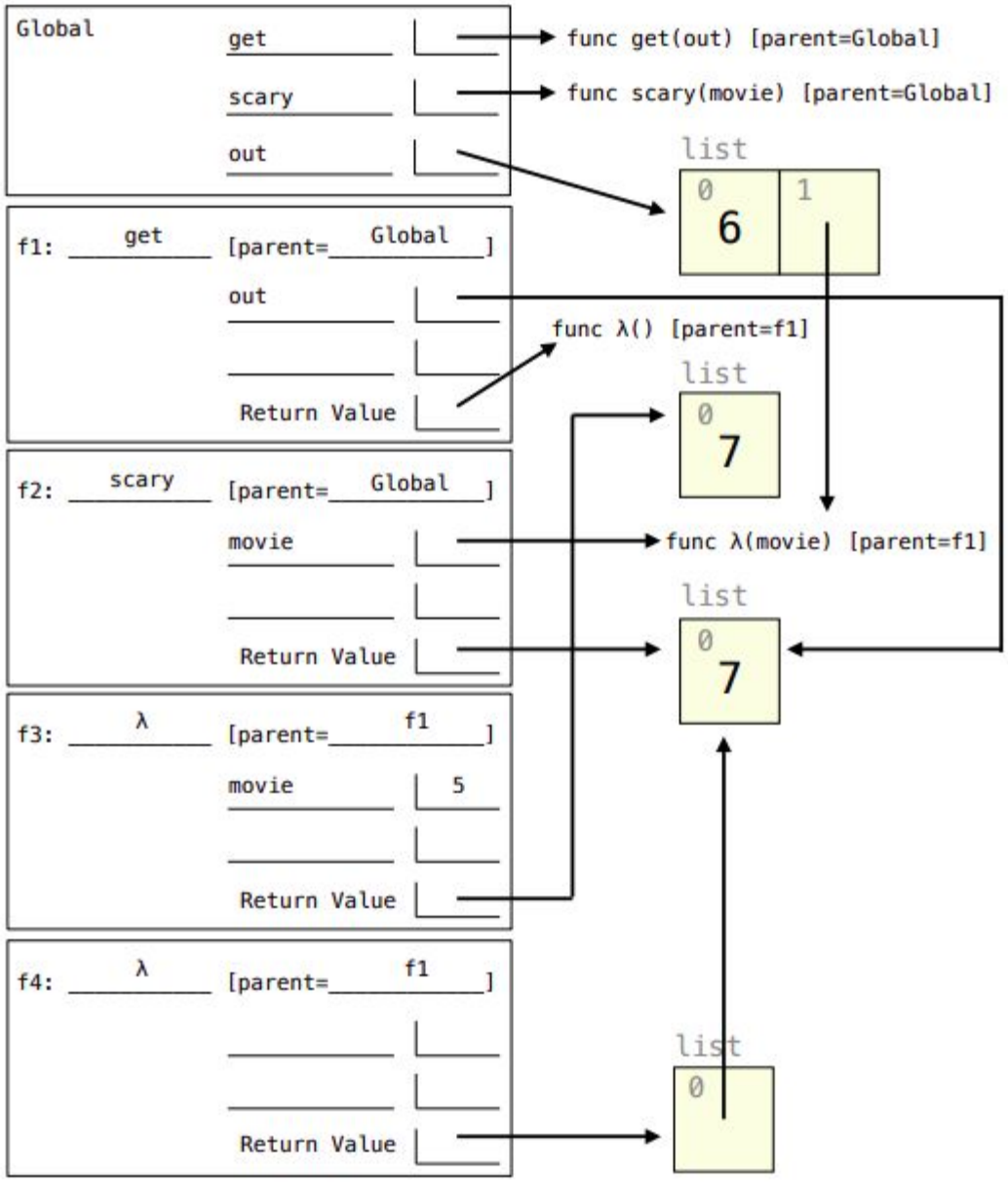
```
o
[o]
list(o)
list([o])
o + []
[o[0], o[1]]
o[:]
```



```
def get(out):
    out.pop()
    out = _____
    return lambda: [out]
def scary(movie):
    out.append(movie)
    return _____

out = [6]

_____
```



```
def get(out):
    out.pop()
    out = scary(lambda movie:
    out)
    return lambda: [out]
def scary(movie):
    out.append(movie)
    return movie(5)[:1]

out = [6]
get([7, 8])()
```



More Review