

CSM Mock Final

Spring 2018

1. WWPD (10 pts)

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated on the provided line.

The interactive interpreter displays the repr string of the value of a successfully evaluated expression, unless it is None

Write FUNC to indicate a functional value.

Assume that you have started python3 and executed all the code in the left column first.

<pre>class A: a = 5 def __init__(self, lst, n): a = 1 self.lst = lst self.n = n def update(self): for i in range(len(self.lst)): self.lst[i] = self.lst[i] * self.n self.a += 1 class A2(A): a = 3 def update(self): for i in range(len(self.lst)): self.lst[i] = self.lst[i] - self.n self.a -= 1 class B: def __init__(self, a): self.a = a</pre>	<pre>c = [3, 5, 6] a = A(c, 2) b = A2(c, 3) c = b a.a _____ c.a _____ a.update() c.update() A.a _____ A2.a _____ a.lst _____ B(a).a.a _____ B(A).a.update() _____</pre>
--	---

2. Environment Diagram (8 pts)

Create the environment diagram that results from executing the code below until the entire program is finished or an error occurs. Be sure to include the global frame as well as any other frames created.

```
def f(f):
    def h(x, y):
        z = 4
        return lambda z: (x + y) * z

    def g(y):
        nonlocal g, h
        g = lambda y: y[4]
        h = lambda x, y: lambda f: lambda z: f(f(x) + (y + z))

        return y[5] + y[2:4] + y[6]

    return h(g("cosmic!"), g("why!"))

f = f(lambda f: f(f))(2)
```

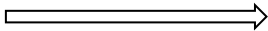
(Extra room if you want to do the diagram on this page)

3. List Diagram (8 pts)

For each part, create the box and pointer diagram representing the lists after the code above is run. You do not need to include indices in your diagram.

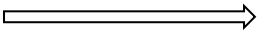
#part a

```
t = [5, [6,7]]  
t.extend(t[1])
```

t 

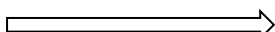
#part b


```
t = [5, [6,7], 8]  
t.append(lambda: t.extend(t[-3:]))  
t[3]()
```

t 

#part c

```
t = [5, [6,7], lambda: t.append([[0]])]  
b = t + [1]  
b[2]()  
t.extend(b.pop(1))
```

t 

b 

5. Clever Pun #5 (10 pts)

Suppose Samo the dog needs to make his way across an $n \times n$ grid to get back to Professor DeNero. Samo is a very loyal dog and wants to reach the Professor in the fewest moves possible, but at the same time, Samo is an opportunist, and notices treats scattered throughout the grid.

Suppose Samo starts at location $(0, 0)$ on the grid G and Professor DeNero is at location (n, n) on the grid; that is, they are on opposite corners. Our input grid G tells us how many treats are at any location - for a location (x, y) , the number of treats in that location can be found with $G[x][y]$. Given that Samo can move up, down, left, or right (no diagonals), and that Samo will eat all the treats in a location **as he leaves it**, fill in the function `trail_of_treats` to return the maximum amount of treats Samo can eat if he takes the minimum moves to get to Professor DeNero. (Samo will also eat all the treats at Professor DeNero's location when he reaches it.)

```
def trail_of_treats(G):
```

```
def trail_helper(G, x, y):
```

```
    if _____:
```

```
    elif _____:
```

```
    else _____:
```

```
        a = _____
```

```
        b = _____
```

6. Infinite Generator (10 pts)

Create a class `Inf_Gen` that generates an iterator that iterates through all the elements of some sequence, and upon reaching the end, loops back to the beginning. It should also be able to go in the reverse order and the first element should loop forward to the last element of the iterable.

```
class Inf_Iter:
```

```
    """Creates an iterator that can iterate in either direction over its elements for any number of calls to next().
```

```
>>> a = Inf_Gen([2,4,6,8,10])
```

```
>>> it = a.gen()
```

```
>>> next(it)
```

```
2
```

```
>>> next(it)
```

```
4
```

```
>>> next(it)
```

```
6
```

```
>>> a.rev()
```

```
>>> next(it)
```

```
4
```

```
>>> next(it)
```

```
2
```

```
>>> next(it)
```

```
10
```

```
>>>a.rev()
```

```
>>> next(it)
```

```
8
```

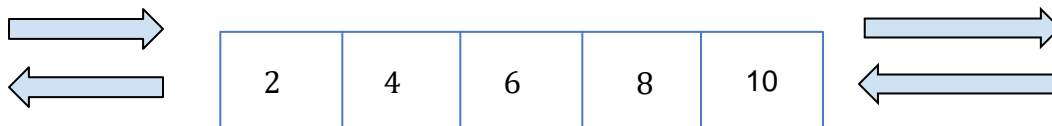
```
>>> next(it)
```

```
10
```

```
>>>next(it)
```

```
2
```

```
"""
```




```
def __init__(self, lst):  
    self.lst = _____  
    self.index = _____  
    self.reverse = _____
```

```
def gen(self):  
    while _____:  
        _____  
        if _____:  
            _____  
            _____  
            _____  
        else _____:  
            _____  
            _____  
            _____  
            _____
```

```
def rev(self):  
    _____  
    _____
```

7. Linked List (10 pts)

Create a function `partition_sll` that takes in a linked list as an argument and non-destructively returns a linked list composed of three smaller linked lists, one less than, one equal to, and one greater than, the first element of the original linked list.

```
def partition_sll(lnk):
```

```
    """
```

```
    >>> lnk = Link(5, Link(2, Link(3, Link(1, Link(4))))
```

```
    >>> partition_sll(lnk)
```

```
    Link(Link(4, Link(1, Link(3, Link(2))))), Link(Link(5), Link(Link.empty)))
```

```
    >>> lnk2 = Link(3, Link(4, Link(3, Link(1, Link(4))))
```

```
    >>> partition_sll(lnk2)
```

```
    Link(Link(1), Link(Link(3, Link(3)), Link(Link(4, Link(4))))
```

```
    """
```

```
    less, equal, greater, pivot = _____
```

```
    while _____:
```

```
        curr = _____
```

```
        _____
```

```
        _____
```

```
        _____
```

```
        _____
```

```
        _____
```

```
    _____
```

8. Branching Out (4/8pts)

Trees are implemented as objects for all parts of this problem.

Part A

Implement **minimize_tree**, which takes in a tree **t** and changes each node to have the smallest value found from that node downwards. That is, for every node **n** in **t**, **n**'s label should be the smallest number in the subtree rooted at **n**.

```
def minimize_tree(t):
    [_____]
    t.label = min(_____)
```

Part B

Define **rotate**, which rotates the labels at each level of tree **t** to the left by one destructively. This rotation should be modular (That is, the leftmost label at a level will become the rightmost label after running rotate. You do NOT need to rotate across different branches.)

```
def rotate(t):
    """
    >>> t1 = Tree(1, [Tree(2), Tree(3, [Tree(4)], Tree(5))]
    >>> rotate(t1)
    >>> t1
    Tree(1, [Tree(3), Tree(5, [Tree(4)], Tree(2))]
    >>> t2 = Tree(1, [Tree(2, [Tree(3), Tree(4)], Tree(5, [Tree(6)])])
    >>> rotate(t2)
    >>> t2
    Tree(1, [Tree(5, [Tree(4), Tree(3)], Tree(2, [Tree(6)])])
    """
    _____
    _____
    for _____:
        _____
        _____ = _____
    _____
```