LISTS, ABSTRACTION, AND MIDTERM 1 REVIEW Solutions

COMPUTER SCIENCE MENTORS

February 15 - February 17, 2020

1 Lists

Lists Introduction:

Lists are a type of sequence, an ordered collection of values that has both length and the ability to select elements.

```
>>> lst = [1, False, [2, 3], 4] # a list can contain anything
>>> len(lst)
4
>>> lst[0] # Indexing starts at 0
1
>>> lst[4] # Indexing ends at len(lst) - 1
Error: list index out of range
```

We can iterate over lists using their index, or iterate over elements directly

```
for index in range(len(lst)):
        # do things
for item in lst:
        # do things
```

List comprehensions are a useful way to iterate over lists when your desired result is a list.

```
new_list2 = [<expression> for <element> in <sequence> if <
   condition>]
```

We can use **list splicing** to create a copy of a certain portion or all of a list.

```
new_list = lst[<starting index>:<ending index>]
copy = lst[:]
```

1. What would Python display? Draw box-and-pointer diagrams for the following:

```
>>> a = [1, 2, 3]
>>> a
```

```
[1, 2, 3]
```

```
>>> a[2]
```

3

```
>>> b = a
>>> a = a + [4, [5, 6]]
>>> a
```

```
[1, 2, 3, 4, [5, 6]]
```

```
>>> b
```

```
[1, 2, 3]
```

```
>>> c = a
>>> a = [4, 5]
>>> a
```

```
[4, 5]
```

```
>>> c
```

```
[1, 2, 3, 4, [5, 6]]
```

```
>>> d = c[3:5]
>>> c[3] = 9
>>> d
```

```
[4, [5, 6]]
```

```
>>> c[4][0] = 7
>>> d
```

```
[4, [7, 6]]
```

```
>>> c[4] = 10
>>> d
```

```
[4, [7, 6]]
```

```
>>> c
```

```
[1, 2, 3, 9, 10]
```

2. Draw the environment diagram that results from running the code.

```
def reverse(lst):
    if len(lst) <= 1:
        return lst
    return reverse(lst[1:]) + [lst[0]]

lst = [1, [2, 3], 4]
rev = reverse(lst)
```

https://goo.gl/6vPeX9

3. Write a function that takes in a list `nums` and returns a new list with only the primes from `nums`. Assume that `is_prime(n)` is defined. You may use a `while` loop, a `for` loop, or a list comprehension.

```
def all_primes(nums):

    result = []
    for i in nums:
        if is_prime(i):
            result = result + [i]
    return result

    List comprehension:
    return [x for x in nums if is_prime(x)]
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

2 Abstraction

Data Abstraction Overview:

Abstraction allows us to create and access different types of data through a controlled, restricted programming interface, hiding implementation details and encouraging programmers to focus on how data is used, rather than how data is organized. The two fundamental components of a programming interface are a constructor and selectors.

1. Constructor: The interface that creates a piece of data; e.g. calling `c = car("Tesla")` creates a new car object and assigns it to the variable `c`.
2. Selectors: The interface by which we access attributes of a piece of data; e.g. calling `get_brand(c)` should return `"Tesla"`.

Through constructors and selectors, a data type can hide its implementation, and a programmer doesn't need to *know* its implementation to *use* it.

1. The following is an **Abstract Data Type (ADT)** for elephants. Each elephant keeps track of its name, age, and whether or not it can fly. Given our provided constructor, fill out the selectors:

```
def elephant(name, age, can_fly):
    """
    Takes in a string name, an int age, and a boolean
       can_fly.
    Constructs an elephant with these attributes.
    >>> dumbo = elephant("Dumbo", 10, True)
    >>> elephant_name(dumbo)
    "Dumbo"
    >>> elephant_age(dumbo)
    10
    >>> elephant_can_fly(dumbo)
    True
    """
    return [name, age, can_fly]
def elephant_name(e):

    return e[0]
def elephant_age(e):

    return e[1]
```

```python
def elephant_can_fly(e):

    return e[2]
```

2. This function returns the correct result, but there's something wrong about its implementation. How do we fix it?

```python
def elephant_roster(elephants):
    """
    Takes in a list of elephants and returns a list of
       their names.
    """
    return [elephant[0] for elephant in elephants]
```

`elephant[0]` is a Data Abstraction Violation (DAV). We should use a selector instead. The corrected function looks like:

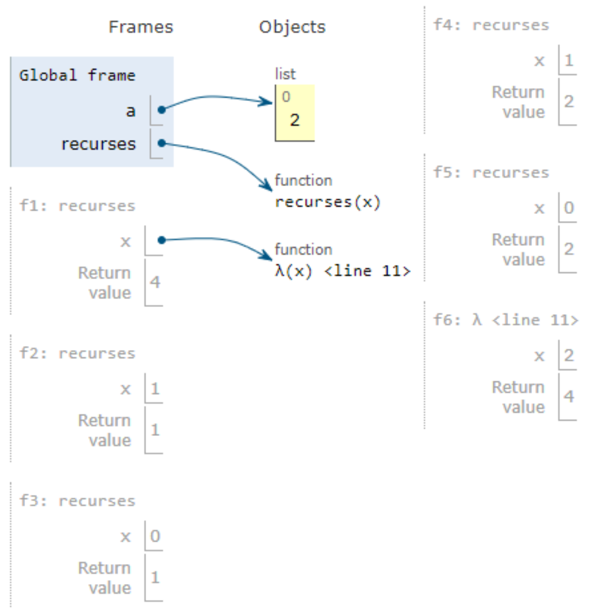
```python
def elephant_roster(elephants):
    return [elephant_name(elephant) for elephant in
       elephants]
```

3 Midterm Review - Environment Diagrams

1. **recurses**

   Fill in each blank in the code below so that its environment diagram is the following. You are not allowed to use operations like +, -, *, /, %, **max**, and **min**.

```
a = [0]
def recurses(x):
    if x == 0:
        return _______________
    elif type(x) == int:
        a[0] += x
        return _______________
    else:
        return _______________\
        ______________________
recurses(lambda x: x * x)
```



Solution

```
a = [0]
def recurses(x):
    if x == 0:
        return a[0]
    elif type(x) == int:
        a[0] += x
        return recurses(0)
    else:
        return x(recurses(recurses(1)))
recurses(lambda x: x * x)
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

4 Midterm Review - Higher Order Functions

1. Make a lambda function, `make_interval()`, that takes in the upper and lower bound of an interval, and returns a function that takes in a value `x` and checks whether `x` is in the interval [`lower`, `upper`], inclusive.

```
>>> make_interval = ______________________________________
>>> in_interval = make_interval(-1, 2)
>>> in_interval(0)
True
>>> in_interval(61)
False
```

```
>>> make_interval = lambda lower, upper: lambda x: x <= upper
    and x >= lower
>>> in_interval = make_interval(-1, 2)
>>> in_interval(0)
True
>>> in_interval(61)
False
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

2. Implement `make_alternator` which takes in two functions and outputs a function. The returned function takes in a number `x` and prints out all the numbers from 1 to `x`, applying `f` to the odd numbers and applying `g` to the even numbers before printing.

```
def make_alternator(f, g):
    """
    >>> a = make_alternator(lambda x: x * x, lambda x: x + 4)
    >>> a(5)
    1
    6
    9
    8
    25
    """

    def alternator(x):
        i = 1
        while i <= x:
            if i % 2 == 1:
                print(f(i))
            else:
                print(g(i))
            i += 1
    return alternator
```

5 Midterm Review - Recursion

1. A game is defined as follows: let `lst` be a list of coins, each coin represented as a positive integer (ex: 1, 5, 25, 10). Two players take turns claiming either the last coin in `lst`, or both the last *and* the second to last coin; after `lst` is exhausted, whichever player has the higher score wins. Fill in the function such that it returns the highest score that the first player (`player = True`) can get in this game if the second player (`player = False`) plays optimally.

   Hint: a player's choice is considered *optimal* if it maximizes their own score and minimizes the opponent's score.

```
def coin_game(lst, player):
    """
    >>> coin_game([1], True) // 1
    1
    >>> coin_game([1, 5, 25], True) // 25 + 5
    30
    >>> coin_game([1, 5, 10, 1, 5, 25], True) // 25 + 1 + 10
    36
    """
    if ______________________ and player:

        return ______________________________
    elif ____________________ and not player:

        return ______________________________
    else:
        if player:
            last = ______________________________________
            second_to_last = ____________________________

            return ________________________________________ \

                ______________________________________________
        else:
            return ________________________________________ \

                ______________________________________________
```

```
def coin_game(lst, player):
    if len(lst) <= 2 and player:
        return sum(lst)
    elif len(lst) <= 2 and not player:
        return 0
    else:
        if player:
            last = lst[len(lst) - 1]
            second_to_last = lst[len(lst) - 2]
            return max(coin_game(lst[:len(lst) - 1], not
               player) + last, coin_game(lst[:len(lst) - 2],
               not player) + last + second_to_last)
        else:
            return min(coin_game(lst[:len(lst) - 1]),
               coin_game(lst[:len(lst) - 2]))
```

6 Midterm Review - Tree Recursion

1. Implement the function `make_change`, which takes in a non-negative integer amount in cents `n` and returns the minimum number of coins needed to make change for `n` using 1-cent, 3-cent, and 4-cent coins.

```
def make_change(n):
    """
    >>> make_change(5) # 5 = 4 + 1 (not 3 + 1 + 1)
    2
    >>> make_change(6) # 6 = 3 + 3 (not 4 + 1 + 1)
    2
    """

    if ______________________:
        return 0

    elif ____________________:

        ____________________________________

    elif ____________________:

        ____________________________________

    else:

        ____________________________________
```

```
def make_change(n):
    if n == 0:
        return 0
    elif n < 3:
        return 1 + make_change(n - 1)
    elif n < 4:
        return 1 + min(make_change(n - 1), make_change(n - 3))
    else:
        return 1 + min(make_change(n - 1), make_change(n - 3),
            make_change(n - 4))
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

7 Midterm Review - Lists

1. Write a list comprehension that accomplishes each of the following tasks.

   (a) Square all the elements of a given list, `lst`.

   ```
   [x ** 2 for x in lst]
   ```

   (b) Compute the dot product of two lists `lst1` and `lst2`. *Hint*: The dot product is defined as $\text{lst1}[0] \cdot \text{lst2}[0] + \text{lst1}[1] \cdot \text{lst2}[1] + \ldots + \text{lst1}[n] \cdot \text{lst2}[n]$. The Python **`zip`** function may be useful here.

   ```
   sum([x * y for x, y in zip(lst1, lst2)])
   ```

   (c) Return a list of lists such that `a = [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]`.

   ```
   a = [[x for x in range(y)] for y in range(1, 6)]
   ```

   (d) Return the same list as above, except now excluding every instance of the number 2: `b = [[0], [0, 1], [0, 1], [0, 1, 3], [0, 1, 3, 4]])`.

   ```
   b = [[x for x in range(y) if x != 2] for y in range(1, 6)]
   ```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni