LISTS, ABSTRACTION, AND MIDTERM 1 REVIEW

COMPUTER SCIENCE MENTORS

February 15 - February 17, 2020

1 Lists

Lists Introduction:

Lists are type of sequence, which means they are 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]
1
>>> lst[4]      # Indices of a list only go up to the len(lst)
Error: list index out of range
```

We can iterate over lists using their index, or iterate of 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>]
```

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

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

>>> a[2]

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

>>> b

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

>>> c


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

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

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

>>> c
```

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)
```

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):
```

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 `t = tree(3)` creates a new tree object and assigns it to `t`. `tree()` is a constructor.
2. Selectors: The interface by which we access attributes of a piece of data; e.g. calling `branches(t)` and `is_leaf(t)` return different attributes of a tree (a list of branches and whether the tree is a leaf, respectively). `branches()` and `is_leaf()` are both selectors.

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):
```

```
def elephant_age(e):
```

```
def elephant_can_fly(e):
```

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

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

3 Midterm Review - WWPD

1.
```
def square(x):
    return x * x

def argentina(n):
    print(n)
    if n > 0:
        return lambda k: k(n+1)
    else:
        return 1 / n
```

After executing the above code, what will Python display when evaluating each of the following expressions?

```
>>> print(1, print(2))
```

```
>>> argentina(0)
```

```
>>> argentina(1)(square)
```

2.
```
def quick_maths(fast, maths):
    return fast ** maths
def maths(fast, maths):
    return fast(maths, 2)
quick_maths, maths = maths, quick_maths
```

After executing the above code, what will Python display when evaluating each of the following expressions?

```
>>> maths(2, 3)
```

```
>>> quick_maths(print, 10)
```

```
>>> quick_maths(maths, 10)
```

4 Midterm Review - Environment Diagrams

1. Draw the environment diagram that results from running the following code.

```
def snow(snow, x):
    if snow(x, x) == x:
        def x(x):
            return 32
        return x(x)
    else:
       return snow(snow, x)
def flake(x, y):
    return y + x - 1
griffin = snow(flake, 1)
```

5 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
```

2. Implement `make_alternator` which takes in two functions and outputs a function. When the function takes in a number `x`, it prints out all the numbers between 1 and `x`, applying `f` to every odd-indexed number and `g` to every even-indexed number 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
    """
```

3. Make a function `print_n` which returns a repeatable print function that can print the next `n` parameters. When there are no parameters left, it prints "done" each time the function is called.

```
def print_n(n):
    """
    >>> f = print_n(2)
    >>> f = f("hi")
    hi
    >>> f = f("hello")
    hello
    >>> f = f("bye")
    done
    >>> f = f("no, not done")
    done
    """
    def inner_print(print_str):
        if ____________________:
            ____________________
        else:
            _____________________
        return _________________
    return _____________________
```

4. Write a function, `print_sum`, that takes in a positive integer, `a`, and returns a function that does the following:
(1) takes in a positive integer, `b`
(2) prints the sum of all natural numbers from 1 to `a*b`
(3) returns a higher-order function that, when called, prints the sum of all natural numbers from 1 to `(a+b)*c`, where `c` is another positive integer.

```
def print_sum(a):
    """
    >>> f = print_sum(1)
    >>> g = f(2) # 1*2 => 1 + 2
    3
    >>> h = g(4) # (1+2)*4 => 1 + 2 + ... + 11 + 12
    78
    >>> i = h(5) # (3+4)*5 => 1 + 2 + ... + 34 + 35
    630
    """
    def helper(b):

        i, total = ____________________________

        while ____________________________:

            ____________________________

            ____________________________

        print(____________________________)

        return ____________________________

    return ____________________________
```

6 Midterm Review -Recursion

1. Suppose a game is defined as follows: let `lst` be a list of coins, each coin represented as a positive integer (ex: 1, 5, 10, 25). 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 ____________________________________ \

                ____________________________________________
```

7 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:

        _______________________________________
```

8 Midterm Review -Challenge Problems

Note: These problems are meant to be challenging and may take a long time. Please attempt the previous questions on the worksheet first.

1. Fill in the methods below according to the doctests.

```
def gen_list(n):
    """
    Returns a nested list structure of n elements where the
    ith element is a list from 0 (inclusive) to i (exclusive).
    >>> gen_list(3)
    [[0], [0, 1], [0, 1, 2]]
    >>> gen_list(5)
    [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]
    """
    return ______________________________________________
```

For an additional challenge, try out the following:

```
def gen_increasing(n):
    """
    Returns a nested list structure of n elements where the
    ith element of each list is one more than the previous
    element (even if the previous is in a prior sublist).
    >>> gen_increasing(3)
    [[0], [1, 2], [3, 4, 5]]
    >>> gen_increasing(5)
    [[0], [1, 2], [3, 4, 5], [6, 7, 8, 9], [10, 11, 12, 13,
    14]]
    """
    return ______________________________________________
```

Hint: You can sum ranges. E.g. `sum(range(3))` gives us $0 + 1 + 2 = 3$.

2. A character tree is a tree where the characters along a path of the tree form a word (as defined in the English dictionary). A path through a tree is a list of adjacent node values that starts from any node and ends with a leaf value.

   Imagine you're playing a version of Scrabble and you really want to win. Implement `scrabble_tree` which takes in a character tree. The function will then find all words in the character tree and return the word with the highest value. You may use the pre-defined functions `is_word(word)` and `score(word)`. You can assume that all characters in the character tree are lower cased.

   The function `is_word(word)` returns True if word is a valid dictionary word and False otherwise. Additionally, you are given a function `score(word)`, which returns the score of word in a game of Scrabble. You do not need to worry about how these functions are implemented.

   Note: If all characters have a weight of 1, then this problem is the same as finding the longest string of the character tree.

   (a) First, implement the function `word_exists`, which takes in a word `word` and a character tree `t`. The function will return `True` if characters along a path from the root of `t` to a leaf spells `word`. Otherwise, it returns `False`.

   ```
   def word_exists(word, t):
       if len(word) == 1:
           return ______________________________________
       elif ____________________:
           return False
       return _____(
              ______________________________________________)
   ```

   (b) Now, implement the function `scrabble_tree`. You may use the function you defined in part a, as well as the provided functions `is_word(word)` and `score`. You may also want to use the built-in Python function `filter`.

   The function `filter` takes in a single argument function as its first parameter and a sequence as its second parameter. The function will then test which elements of the sequence is `True` using the provided function.

   ```
   >>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
   >>> evens = list(filter(lambda x: x % 2 == 0, lst))
   >>> evens
   [2, 4, 6, 8, 10]
   ```

   Note: We have to call list on the output of filter because filter returns an object (which will be covered in a later part of this course).

```
def scrabble_tree(t):
    """
    We assume that all characters have a score of 1.

    >>> t1 = tree('h', [tree('j', [tree('i')])])
    >>> scrabble_tree(t1)
    'hi'
    >>> t2 = tree('i', [tree('l', [tree('l')])])
    >>> t3 = tree('h', [tree('i'), t2])
    >>> scrabble_tree(t3)
    'hill'
    """
    def find_all_words(t):
        if ________________:
            return __________________
        all_words = []
        for b in branches(t):
            words_in_branch = ______________________
            words_from_t = [
               _____________________________________]
            filter_from_t =
               _____________________________________
            all_words =
               _________________________________________
        return __________________
    clean_words = [
       _______________________________________________]
    return max(________________, key=
       ___________________________)
```