LISTS, TREES, AND ABSTRACTION

COMPUTER SCIENCE MENTORS

September 28, 2020 to October 1, 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. Fill out the following constructor for the given selectors.

```
def elephant(name, age, can_fly):
```

```
def elephant_name(e):
    return e[0][0]
def elephant_age(e):
    return e[0][1]
def elephant_can_fly(e):
    return e[1]
```

4. How can we write the fixed `elephant_roster` function for the constructors and selectors in the previous question?

5. **(Optional)** Fill out the following constructor for the given selectors.

```
def elephant(name, age, can_fly):
    """
    >>> chris = elephant("Chris Martin", 38, False)
    >>> elephant_name(chris)
    "Chris Martin"
    >>> elephant_age(chris)
    38
    >>> elephant_can_fly(chris)
    False
    >> chris("size")
    "Breaking abstraction barrier!"
    """
    def select(command)





    return select
def elephant_name(e):
    return e("name")
def elephant_age(e):
    return e("age")
def elephant_can_fly(e):
    return e("can_fly")
```

3 Trees

What are trees?

A tree has a root label and a sequence of branches. Each branch of a tree is a tree. A tree with no branches is called a leaf. Any tree contained within a tree is called a sub-tree of that tree (such as a branch of a branch). The root of each sub-tree of a tree is called a node in that tree. Trees are a recursive data abstraction, since trees have branches that are trees themselves

Because of this, it often makes sense to solve tree problems using recursion:

1. Base case is often when we reach a leaf node
2. Recursive case is often when we still need to recurse down, e.g. we haven't hit a leaf yet. Recursive calls need to break the problem into smaller parts, which for trees often means passing in each branch as an input.

When trying to understand and solve tree problems, it is helpful to draw out the tree.

Things to remember:

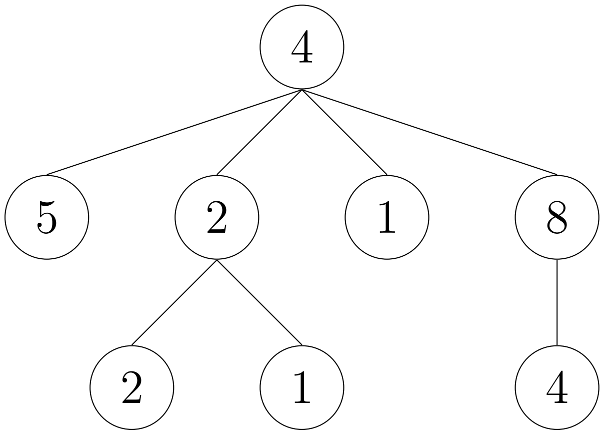
```
def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:] #returns a list of branches
```

Note: You don't have to worry too much about how trees are actually represented as lists–that's the power of abstraction at work!

As shown above, the tree constructor takes in a label and a list of branches (which are themselves trees).

```
tree(4,
     [tree(5),
      tree(2,
          [tree(2),
           tree(1)]),
      tree(1),
      tree(8,
          [tree(4)])])
```

This creates a tree that looks like this (see next page):



1. Let `t` be the tree depicted above. What do the following expressions evaluate to? If the expressions evaluates to a tree, format your answer as `tree(... , ...)`. (Note that the Python interpreter wouldn't display trees like this. This is just so you think about trees as an ADT instead of worrying about their implementation.)

```
>>> label(t)

>>> branches(t)[1]

>>> branches(branches(t)[1])[1]
```

2. Write the function `sum_of_nodes` which takes in a tree and outputs the sum of all the elements in the tree.

```
def sum_of_nodes(t):
    """
    >>> t = tree(...) # Tree from question 1.
    >>> sum_of_nodes(t) # 4 + 5 + 2 + 1 + 8 + 2 + 1 + 4 = 27
    27
    """
```

4 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=
       _________________________)
```