CS 61A
Summer 2023

Mutability, Data Abstraction, Trees

Discussion 4: July 6, 2023

Mutability

Some objects in Python, such as lists and dictionaries, are **mutable**, meaning that their contents or state can be changed. Other objects, such as numeric types, tuples, and strings, are **immutable**, meaning they cannot be changed once they are created.

Let's imagine you order a mushroom and cheese pizza from La Val's, and they represent your order as a list:

```
>>> pizza = ['cheese', 'mushrooms']
```

With list mutation, they can update your order by mutating `pizza` directly rather than having to create a new list:

```
>>> pizza.append('onions')
>>> pizza
['cheese', 'mushrooms', 'onions']
```

Aside from `append`, there are various other list mutation methods:

- `append(elem)`: Add `elem` to the end of the list. Return `None`.
- `extend(lst)`: Extend the list by concatenating it with `lst`. Return `None`.
- `insert(i, elem)`: Insert `elem` at index `i`. This does not replace any existing elements, but only adds the new element `elem`. Return `None`.
- `remove(elem)`: Remove the first occurrence of `elem` in list. Errors if `elem` is not in the list. Return `None` otherwise.
- `pop(i)`: Remove and return the element at index `i`.

We can also use list indexing with an assignment statement to change an existing element in a list. For example:

```
>>> pizza[1] = 'tomatoes'
>>> pizza
['cheese', 'tomatoes', 'onions']
```

Q1: WWPD: Mutability

What would Python display? In addition to giving the output, draw the box and pointer diagrams for each list to the right.

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4]
>>> x
```

```
>>> y # y is pointing to the same list as x, which got mutated
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4] # creates NEW list, assigns it to x
>>> x
```

```
>>> y   # y still points to OLD list, which was not mutated
```

```
>>> s1 = [1, 2, 3]
>>> s2 = s1
>>> s1 is s2
```

```
>>> s2.extend([5, 6])
>>> s1[4]
```

```
>>> s1.append([-1, 0, 1])
>>> s2[5]
```

```
>>> s3 = s2[:]
>>> s3.insert(3, s2.pop(3))
>>> len(s1)
```

```
>>> s1[4] is s3[6]
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

```
>>> s3[s2[4][1]]
```

```
>>> s1[:3] is s2[:3]
```

```
>>> s1[:3] == s2[:3]
```

```
>>> s1[4].append(2)
>>> s3[6][3]
```

Q2: Insert Items

Write a function which takes in a list `lst`, an argument `entry`, and another argument `elem`. This function will check through each item in `lst` to see if it is equal to `entry`. Upon finding an item equal to `entry`, the function should modify the list by placing `elem` into `lst` right after the item. At the end of the function, the modified list should be returned.

See the doctests for examples on how this function is utilized.

Important: Use list mutation to modify the original list. No new lists should be created or returned.

> **Note:** If the values passed into `entry` and `elem` are equivalent, make sure you're not creating an infinitely long list while iterating through it. If you find that your code is taking more than a few seconds to run, the function may be in an infinite loop of inserting new values.

```
def insert_items(lst, entry, elem):
    """Inserts elem into lst after each occurrence of entry and then returns lst.
    Do not create any new lists.

    >>> test_lst = [1, 5, 8, 5, 2, 3]
    >>> new_lst = insert_items(test_lst, 5, 7)
    >>> new_lst
    [1, 5, 7, 8, 5, 7, 2, 3]
    >>> test_lst
    [1, 5, 7, 8, 5, 7, 2, 3]
    >>> double_lst = [1, 2, 1, 2, 3, 3]
    >>> double_lst = insert_items(double_lst, 3, 4)
    >>> double_lst
    [1, 2, 1, 2, 3, 4, 3, 4]
    >>> large_lst = [1, 4, 8]
    >>> large_lst2 = insert_items(large_lst, 4, 4)
    >>> large_lst2
    [1, 4, 4, 8]
    >>> large_lst3 = insert_items(large_lst2, 4, 6)
    >>> large_lst3
    [1, 4, 6, 4, 6, 8]
    >>> large_lst3 is large_lst
    True
    """
    "*** YOUR CODE HERE ***"













# You can use more space on the back if you want
```

Q3: Add This Many

Write a function that takes in a value `x`, a value `elem`, and a list `s`, and adds `elem` to the end of `s` the same number of times that `x` occurs in `s`. **Make sure to modify the original list using list mutation techniques.**

```
def add_this_many(x, el, s):
    """Adds el to the end of s the number of times x occurs in s.
    >>> s = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, s)
    >>> s
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, s)
    >>> s
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
    "*** YOUR CODE HERE ***"















# You can use more space on the back if you want
```

Data Abstraction

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects. For example, using code to represent cars, chairs, people, and so on. That way, programmers don't have to worry about *how* code is implemented; they just have to know *what* it does.

Data abstraction mimics how we think about the world. If you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of to do so. You just have to know how to use the car for driving itself, such as how to turn the wheel or press the gas pedal.

A data abstraction consists of two types of functions:

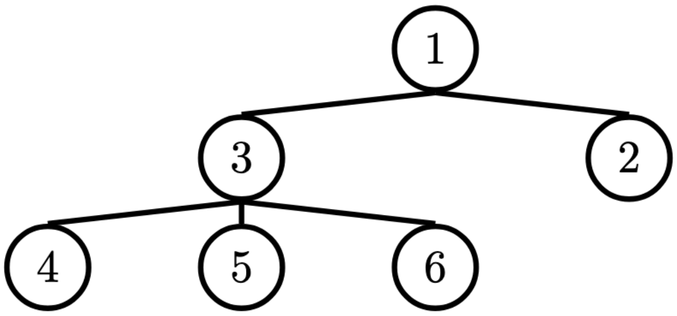
- **Constructors**: functions that build the abstract data type.
- **Selectors**: functions that retrieve information from the data type.

Programmers design data abstractions to abstract away how information is stored and calculated such that the end user does *not* need to know how constructors and selectors are implemented. The nature of *abstraction* allows whoever uses them to assume that the functions have been written correctly and work as described.

Trees

One example of data abstraction is with **trees**.

In computer science, **trees** are recursive data structures that are widely used in various settings and can be implemented in many ways. The diagram below is an example of a tree.



Example Tree

Notice that the tree branches downward. In computer science, the **root** of a tree starts at the top, and the **leaves** are at the bottom.

Some terminology regarding trees:

- **Parent Node**: A node that has at least one branch.
- **Child Node**: A node that has a parent. A child node can only have one parent.
- **Root**: The top node of the tree. In our example, this is the `1` node.
- **Label**: The value at a node. In our example, every node's label is an integer.
- **Leaf**: A node that has no branches. In our example, the `4`, `5`, `6`, `2` nodes are leaves.
- **Branch**: A subtree of the root. Trees have branches, which are trees themselves: this is why trees are *recursive* data structures.
- **Depth**: How far away a node is from the root. We define this as the number of edges between the root to the node. As there are no edges between the root and itself, the root has depth 0. In our example, the `3` node has depth 1 and the `4` node has depth 2.
- **Height**: The depth of the lowest (furthest from the root) leaf. In our example, the `4`, `5`, and `6` nodes are all the lowest leaves with depth 2. Thus, the entire tree has height 2.

In computer science, there are many different types of trees. Some vary in the number of branches each node has; others vary in the structure of the tree.

Working with Trees

In this class, our first encounter with trees will be using the `tree` *abstract data type* (ADT), which is later defined. Through data abstraction, we've essentially designed the behavior for our very own type, just like an integer or string!

A tree has both a value for the root node and a sequence of branches, which are also trees. In our implementation, we represent the branches as a list of trees. Since a tree is a data abstraction, our choice to use lists is just an implementation detail.

- The arguments to the constructor `tree` are the value for the root node and an optional list of branches. *If no branches parameter is provided, the default value* `[]` *is used.*
- The selectors for these are `label` and `branches`.

Remember `branches` returns a list of trees and not a tree directly. It's important to distinguish between working with a tree and working with a **list of** trees.

We have also provided a convenience function, `is_leaf`.

Let's try to create the tree from above:

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

Tree ADT Implementation

For your reference, we have provided our implementation of trees as a data abstraction. However, as with any data abstraction, we should only concern ourselves with what our functions do rather than their specific implementation!

```
def tree(label, branches=[]):
    """Construct a tree with the given label value and a list of branches."""
    return [label] + list(branches)

def label(tree):
    """Return the label value of a tree."""
    return tree[0]

def branches(tree):
    """Return the list of branches of the given tree."""
    return tree[1:]

def is_leaf(tree):
    """Returns True if the given tree's list of branches is empty, and False
    otherwise.
    """
    return not branches(tree)
```

Q4: Tree Abstraction Barrier

Consider a tree `t` constructed by calling `tree(1, [tree(2), tree(4)])`. For each of the following expressions, answer these two questions:

- What does the expression evaluate to?
- Does the expression violate any abstraction barriers? If so, write an equivalent expression that does not violate abstraction barriers.

1. `label(t)`
2. `t[0]`
3. `label(branches(t)[0])`
4. `is_leaf(t[1:][1])`
5. `[label(b) for b in branches(t)]`
6. **Challenge:** `branches(tree(5, [t, tree(3)]))[0][0]`

Q5: Height

Write a function that returns the height of a tree. Recall that the height of a tree is the length of the longest path from the root to a leaf.

```
def height(t):
    """Return the height of a tree.
    >>> t = tree(3, [tree(5, [tree(1)]), tree(2)])
    >>> height(t)
    2
    >>> t = tree(3, [tree(1), tree(2, [tree(5, [tree(6)]), tree(1)])])
    >>> height(t)
    3
    """
    "*** YOUR CODE HERE ***"




# You can use more space on the back if you want
```

Q6: Maximum Path Sum

Write a function that takes in a tree and returns the maximum sum of the values along any path in the tree. Recall that a path is from the tree's root to any leaf.

```
def max_path_sum(t):
    """Return the maximum path sum of the tree.
    >>> t = tree(1, [tree(5, [tree(1), tree(3)]), tree(10)])
    >>> max_path_sum(t)
    11
    """
    "*** YOUR CODE HERE ***"

# You can use more space on the back if you want
```

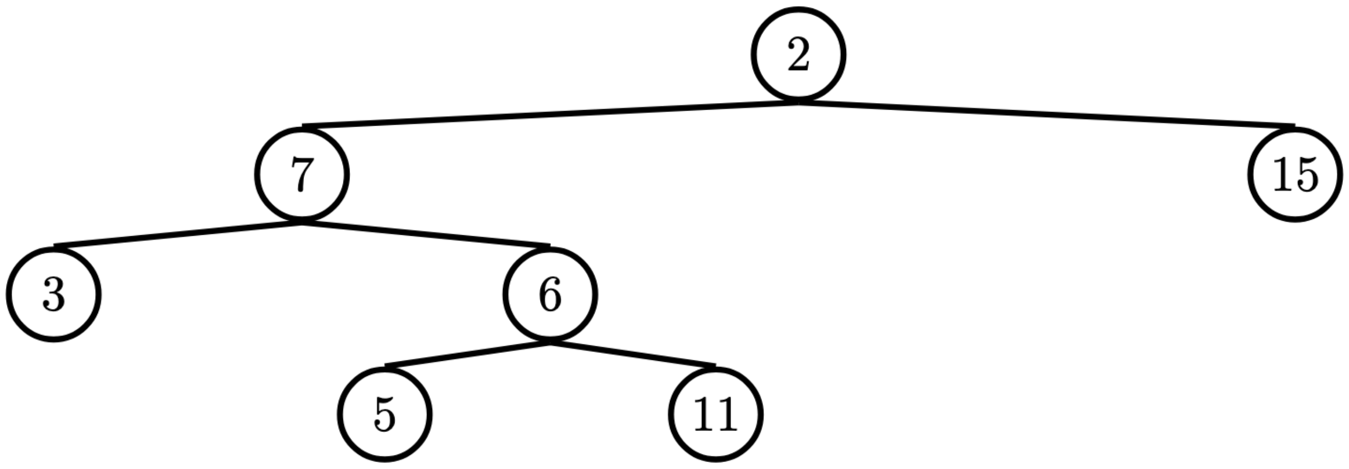
Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Q7: Find Path

Write a function that takes in a tree and a value `x` and returns a list containing the nodes along the path required to get from the root of the tree to a node containing `x`.

If `x` is not present in the tree, return `None`. Assume that the entries of the tree are unique.

For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`.



Example Tree

```
def find_path(t, x):
    """
    >>> t = tree(2, [tree(7, [tree(3), tree(6, [tree(5), tree(11)])] ), tree(15)])
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """
    if ______________________________:
        return ______________________________
    ______________________________:
        path = ______________________
        if ______________________________:
            return ______________________________
```

Q8: Sprout Leaves

Define a function `sprout_leaves` that takes in a tree, `t`, and a list of leaves, `leaves`. It produces a new tree that is identical to `t`, but where each old leaf node has new branches, one for each leaf in `leaves`.

For example, say we have the tree `t = tree(1, [tree(2), tree(3, [tree(4)])])`:

```
  1
 / \
2   3
    |
    4
```

If we call `sprout_leaves(t, [5, 6])`, the result is the following tree:

```
      1
    /   \
   2     3
  / \    |
 5   6   4
        / \
       5   6
```

```
def sprout_leaves(t, leaves):
    """Sprout new leaves containing the data in leaves at each leaf in
    the original tree t and return the resulting tree.
    >>> t1 = tree(1, [tree(2), tree(3)])
    >>> print_tree(t1)
    1
      2
      3
    >>> new1 = sprout_leaves(t1, [4, 5])
    >>> print_tree(new1)
    1
      2
        4
        5
      3
        4
        5

    >>> t2 = tree(1, [tree(2, [tree(3)])])
    >>> print_tree(t2)
    1
      2
        3
    >>> new2 = sprout_leaves(t2, [6, 1, 2])
    >>> print_tree(new2)
    1
      2
        3
          6
          1
          2
    """
    "*** YOUR CODE HERE ***"









# You can use more space on the back if you want
```

Dictionaries

Dictionaries are data structures which map keys to values. Dictionaries in Python are unordered, unlike real-world dictionaries — in other words, key-value pairs are not arranged in the dictionary in any particular order. Let's look at an example:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 151}
```

The *keys* of a dictionary can be any *immutable* value, such as numbers, strings, and tuples.[1] Dictionaries themselves are mutable; we can add, remove, and change entries after creation. There is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed.

To access the value of `dictionary` at `key`, use the syntax `dictionary[key]`.

Element selection and reassignment work similarly to sequences, except the square brackets contain the key, not an index.

[1]To be exact, keys must be *hashable*, which is out of scope for this course. This means that some mutable objects, such as classes, can be used as dictionary keys.

Q9: WWPD: Dictionaries

What would Python display?

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148}
>>> pokemon
```

```
>>> 'mewtwo' in pokemon
```

```
>>> len(pokemon)
```

```
>>> pokemon['mew'] = pokemon['pikachu']
>>> pokemon[25] = 'pikachu'
>>> pokemon
```

```
>>> pokemon['mewtwo'] = pokemon['mew'] * 2
>>> pokemon
```

```
>>> pokemon[['firetype', 'flying']] = 146
```

Note that the last example demonstrates that dictionaries cannot use other mutable data structures as keys. However, dictionaries can be arbitrarily deep, meaning the *values* of a dictionary can be themselves dictionaries.

Additional Practice

Q10: Dictionary Comprehension

Fill in the blank to complete the implementation of the `dict_comp` function, which takes a tuple of tuples and returns a dictionary that maps...*each tuple* to the consecutive pair of digits with maximum absolute pairwise difference.

Assume that sub-tuples have at least 2 elements.

```
>>> dict_comp(((1, 2, 5, -6, 10), (-30, 4, 20, 1)))
{(1, 2, 5, -6, 10): (-6, 10), (-30, 4, 20, 1): (-30, 4)}
```

- In the first sublist `(1, 2, 5, -6, 10)`, we have `(1, 2)`, `(2, 5)`, `(5, -6)`, `(-6, 10)`. Among these pairs, the last one has the maximum absolute pairwise difference of 16.
- In the second sublist, notice that `(-30, 20)` has the maximum pairwise difference. However, they are not consecutive to one another.

  Note: Why are we using tuples? Recall that the keys of a dictionary must be an immutable value! Lists are mutable, so we're going to have to use tuples instead. You'll see more of this in Cats!

  Note: We generally don't use dictionary comprehensions in this class (opting for classic loops and manual insertion instead), so we've set up the dictionary portion of the code for you. Think of this as just a more involved list comprehension problem and a taste of the things you can do with dictionaries!

```
def dict_comp(nested_lst):
    """Given a tuple of tuple of numbers, return a dictionary that maps
    each tuple to the consecutive pair of digits with maximum absolute
    pairwise difference.

    >>> dict_comp(((1, 2, 5, -6, 10), (-30, 4, 20, 1)))
    {(1, 2, 5, -6, 10): (-6, 10), (-30, 4, 20, 1): (-30, 4)}
    >>> dict_comp(((1, 2), (3, 4), (5, 6)))
    {(1, 2): (1, 2), (3, 4): (3, 4), (5, 6): (5, 6)}
    """
    return {lst: ______________________________________________________ for lst in
    nested_lst}
```

Q11: Perfectly Balanced

Part A: Implement `sum_tree`, which returns the sum of all the labels in tree `t`.

Part B: Implement `balanced`, which returns whether every branch of `t` has the same total sum and that the branches themselves are also balanced.

> **Hint:** If we ever need to select a specific branch, we will need to break the abstraction barrier and index into our branches list. While it is not "good practice", you may need to do so in this problem (or on exams)!

Challenge: Solve both of these parts with just 1 line of code each.

```
def sum_tree(t):
    """
    Add all elements in a tree.
    >>> t = tree(4, [tree(2, [tree(3)]), tree(6)])
    >>> sum_tree(t)
    15
    """
    "*** YOUR CODE HERE ***"











# You can use more space on the back if you want
def balanced(t):
    """
    Checks if each branch has same sum of all elements and
    if each branch is balanced.
    >>> t = tree(1, [tree(3), tree(1, [tree(2)]), tree(1, [tree(1), tree(1)])])
    >>> balanced(t)
    True
    >>> t = tree(1, [t, tree(1)])
    >>> balanced(t)
    False
    >>> t = tree(1, [tree(4), tree(1, [tree(2), tree(1)]), tree(1, [tree(3)])])
    >>> balanced(t)
    False
    """
    "*** YOUR CODE HERE ***"











# You can use more space on the back if you want
```