Data Abstraction, Function-based Trees, and Mutability Solutions

Computer Science Mentors 61A

February 27–March 03, 2022

1 Abstraction

Data abstraction allows us to create and access data through a controlled, restricted programming interface—hiding implementation details for sake of brevity and reusability of code and encouraging programmers to focus on how data is used rather than worrying about how data is internally organized. The two fundamental components of an **abstract data type** are a constructor and selectors:

1. A **constructor** creates a piece of data, and includes all the attributes that make the data unique; e.g. executing `c = car("Nissan", "Leaf")` creates a new instance of a car abstraction and assigns it to the variable `c`.
2. **Selectors** access attributes of a piece of data; e.g. calling `get_make(c)` returns `"Nissan"`.

In the example above, you don't know specifically how the model name "Nissan" and the make name "Leaf" are internally bundled into a car, and you don't care, either. The creator of the abstract data type dealt with those details, so that you, the user of the ADT, would only have to know how to store and retrieve the data you need. This separation of concerns between designing and using an interface is called the **abstraction barrier**. While your program won't necessarily break if you break the abstraction barrier, heeding the barrier is best practice and can prevent errors down the road.

Using abstraction to hide unnecessary details can be seen everywhere, not just in code—keyboards, printers, cars, stovetops, and typewriters all employ abstractive interfaces. What are some examples of abstraction in your everyday life?

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang
Provided by CSM 61A contributors under CC BY 4.0/3-clause BSD.

1. The following is an abstract data type that represents Pokemon. Each Pokemon keeps track of its name, type, and friends. Given our provided constructor, fill out the selectors:

```
def pokemon(name, p_type, friends):
    """
    Constructs a Pokemon with the given attributes.
    >>> cyndaquil = pokemon('Cyndaquil', 'Fire', ['Chikorita', 'Totodile'])
    >>> p_name(cyndaquil)
    'Cyndaquil'
    >>> p_type(cyndaquil)
    'Fire'
    >>> p_friends(cyndaquil)
    ['Chikorita', 'Totodile']
    """
    return [name, p_type, friends]

def p_name(p):


    return p[0]

def p_type(p):


    return p[1]

def p_friends(p):


    return p[2]
```

2. This function returns the correct result, but there's something wrong with its implementation. What's the issue, and how can we fix it?

```
def are_friends(p1, p2):
    """
    Returns True iff the Pokemon p1 and p2 are each other's friends.
    """
    return p1[0] in p2[2] and p2[0] in p1[2]
```

Treating the `p1` and `p2` are lists is a Data Abstraction Violation (DAV). We should use a selector instead. The corrected function looks like:

```
def are_friends(p1, p2):
    return p_name(p1) in p_friends(p2) and p_name(p2) in p_friends(p1)
```

3. Write the function `cross_type_friends`, which takes in a Pokemon `p` and a list of Pokemon `pokemon_list` and returns a list of the names of `p`'s cross-type friends in `pokemon_list`. (A cross-type friend is a friend of a different type.) You may assume that the `are_friends` function has been correctly implemented.

```
def cross_type_friends(p, pokemon_list):
    """
    >>> c = pokemon('Charmander', 'Fire', ['Torchic', 'Squirtle',
        'Bulbasaur'])
    >>> t = pokemon('Torchic', 'Fire', ['Charmander', 'Squirtle'])
    >>> s = pokemon('Squirtle', 'Water', ['Torchic', 'Bulbasaur'])
    >>> b = pokemon('Bulbasaur', 'Grass', ['Charmander', 'Squirtle'])
    >>> cross_type_friends(c, [t, s, b])
    ['Bulbasaur']
    >>> cross_type_friends(b, [c, s, b])
    ['Charmander', 'Squirtle']
    """


    friend_list = []
    for other in pokemon_list:
        if are_friends(p, other) and p_type(p) != p_type(other):
            friend_list += [p_name(other)]
    return friend_list

    # Alternative solution

    return [p_name(o) for o in pokemon_list if are_friends(p, o) and
        p_type(p) != p_type(o)]
```

4. In this problem, you'll change the implementation of the Pokemon ADT while keeping the interface the same.

   (a) Complete the constructor for the given selectors.

```
def pokemon(name, p_type, friends):
    """
    >>> lil_guy = pokemon('Pikachu', 'Electric', ['Mewtwo', 'Lucario'])
    >>> p_name(lil_guy)
    'Pikachu'
    >>> p_type(lil_guy)
    'Electric'
    >>> p_friends(lil_guy)
    ['Mewtwo', 'Lucario']
    """


    def select(command):
        if command == 'name':
            return name
        elif command == 'type':
            return p_type
        elif command == 'friends':
            return friends
    return select
```

   Alternate solution:

```
    return lambda sel: {'name': name, 'type': p_type, 'friends':
       friends}[sel]
```

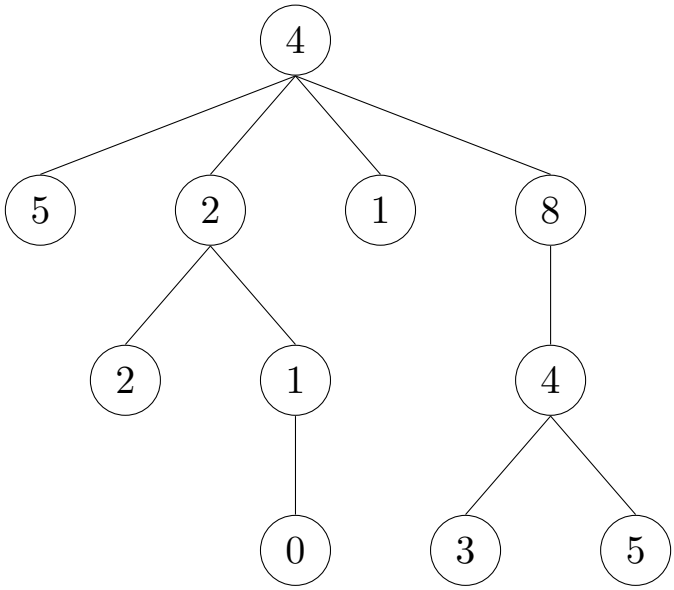
```
def p_name(p):
    return p('name')

def p_type(p):
    return p('type')

def p_friends(p):
    return p('friends')
```

   (b) What do we need to change about the implementations of `are_friends` (as revised) and `cross_type_friends` now that we've changed the implementation of the Pokemon ADT? Why?

   Nothing. Because we relied on the implementation-independent interface of the Pokemon ADT, changing the underlying implementation does not affect the correctness of these functions.

2 Tree ADT

Trees are a kind of recursive data structure. Each tree has a **root label** (which is some value) and a sequence of **branches**. Trees are "recursive" because the branches of a tree are trees themselves! A typical tree might look something like this:



This tree's root label is 4, and it has 4 branches, each of which is a smaller tree. The 6 of the tree's **subtrees** are also **leaves**, which are trees that have no branches.

Trees may also be viewed **relationally**, as a network of nodes with parent-child relationships. Under this scheme, each circle in the tree diagram above is a node. Every non-root node has one parent above it and every non-leaf node has at least one child below it.

Trees are represented by an abstract data type with a `tree` constructor and `label` and `branches` selectors. The `tree` constructor takes in a label and a list of branches and returns a tree. Here's how one would construct the tree shown above with `tree`:

```
tree(4,
    [tree(5),
     tree(2,
        [tree(2),
         tree(1,
            [tree(0)])]),
     tree(1),
     tree(8,
        [tree(4,
            [tree(3), tree(5)])])])
```

The implementation of the ADT is provided here, but you shouldn't have to worry about this too much. (Remember the abstraction barrier!)

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

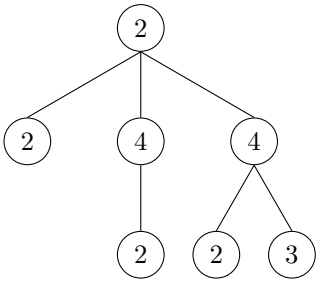
```
def label(tree):
        return tree[0]

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

Because trees are recursive data structures, recursion tends to a be a very natural way of solving problems that involve trees.

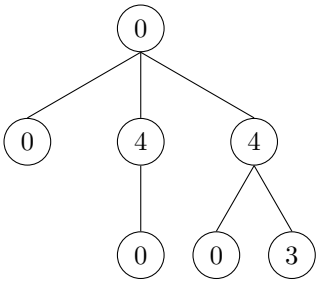
- The **recursive case** for tree problems often involves recursive calls on the branches of a tree.
- The **base case** is often reached when we hit a leaf because there are no more branches to recurse on.

1. Write a function, `replace_x` that takes in a tree, `t`, and returns a new tree with all labels `x` replaced with 0.

   For example, if we called `replace_x(t, 2)` on the following tree:



   We would expect it to return




```
def replace_x(t, x):
    """
    >>> t = tree(2, [tree(1), tree(2)])
    >>> replace_x(t, 2)
    tree(0, [tree(1), tree(0)])
    """
    ____________________________________________

    if ____________________________________________:

        return ____________________________________

    return ________________________________________
```

```
def replace_x(t, x):
    new_branches = [replace_x(b, x) for b in branches(t)]
    if label(t) == x:
        return tree(0, new_branches)
    return tree(label(t), new_branches)
```

Here, we construct and return a new tree. First, we make a new list of branches where each branch is the same as the previous branch but all occurrences of x have been replaced with 0 (this is what the output of `replace_x` is defined to be.)

If the label of our tree is equal to x, we will additionally need to "replace" our current label with 0 in the tree we return. Otherwise, we can keep our previous label.

These two steps guarantee that each occurrence of x is replaced.

We do not need a base case here, as if we are at a leaf, the list comprehension we use to create the new branches will evaluate to an empty list. Then we will either return `tree(0, [])` or `tree(label(t), [])` as appropriate.

2. Write a function that returns `True` if and only if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

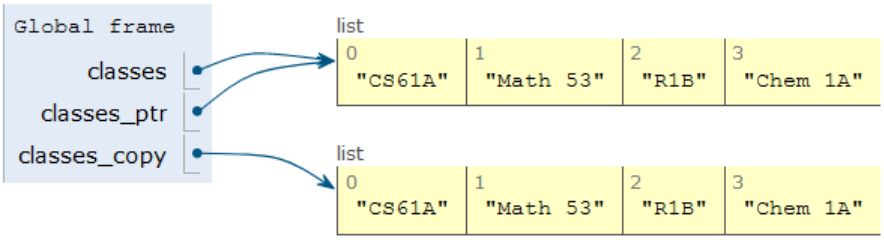
```
def contains_n(elem, n, t):
    """
    >>> t1 = tree(1, [tree(1, [tree(2)])])
    >>> contains_n(1, 2, t1)
    True
    >>> contains_n(2, 2, t1)
    False
    >>> contains_n(2, 1, t1)
    True
    >>> t2 = tree(1, [tree(2), tree(1, [tree(1), tree(2)])])
    >>> contains_n(1, 3, t2)
    True
    >>> contains_n(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif ____________________________________________:

        return ____________________________________

    elif label(t) == elem:

        return ____________________________________

    else:

        return ____________________________________
```

```
def contains_n(elem, n, t):
    if n == 0:
        return True
    elif is_leaf(t):
        return n == 1 and label(t) == elem
    elif label(t) == elem:
        return True in [contains_n(elem, n - 1, b) for b in
          branches(t)]
    else:
        return True in [contains_n(elem, n, b) for b in
          branches(t)]
```

3 Mutability

Let's imagine it's your first year at Cal, and you have signed up for your first classes!

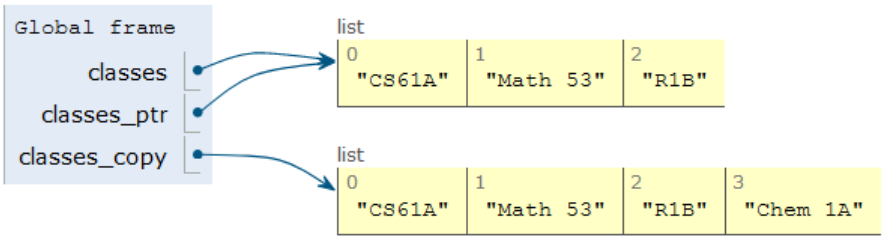
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



After a few weeks, you realize that you cannot keep up with the workload and you need to drop a class. You've chosen to drop Chem 1A. Based on what we know so far, to change our classes list, we would have to create a new list with all the same elements as the original list except for Chem 1A. But that is silly, since all we really need to do is remove the Chem 1A element from our list.

We can fix this issue with list mutation. In Python, some objects, such as lists and dictionaries, are mutable, meaning that their contents or state can be changed over the course of program execution. Other objects, such as numeric types, tuples, and strings are immutable, meaning they cannot be changed once they are created. Therefore, instead of creating a new list, we can just call `classes.pop()`, which removes the last element from the list.

```
>>> classes.pop() # pop returns whatever item it removed
"Chem 1A"
```



(credits: Mihira Patel)

Here are a few other list methods that mutate:

- `append(el)`: Adds `el` to the end of the list
- `extend(lst)`: Extends the list by concatenating `lst` onto the end
- `insert(i, el)`: Inserts `el` at index `i` (does not replace element but adds a new one)
- `remove(el)`: Removes the first occurrence of `el` in the list; errors if `el` is not in the list
- `pop(i)`: Removes and returns the element at index `i`; if no index is provided, it removes and returns the last element of the list

In addition to these methods, there are a few other built-in ways to mutate lists:

- `lst += lst` (**This is distinct from** `lst = lst + lst`)
- `lst[i] = x`
- `lst[i:j] = lst`

On the other hand, the following non-mutative (*non-destructive*) operations do not change the original list but create a new list instead:

- `lst + lst`
- `lst * n`
- `lst[i:j]`
- **`list`**`(lst)`

1. What would Python display? If an error occurs, write "Error". If a function is displayed, write "Function". If nothing is returned, write "Nothing". Also, draw box-and-pointer diagram for each list created.

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

```
2
```

```
>>> a.append([3, 4])
>>> a
```

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

```
>>> b = a[1:]
>>> a[1] = 5
>>> a[2][0] = 6
>>> b
```

```
[2, [6, 4]]
```

```
>>> a.extend([7])
>>> a += [8]
>>> a
```

```
[1, 5, [6, 4], 7, 8]
```

```
>>> a += 9
```

```
TypeError: 'int' object is not iterable
```

Box-and-pointer diagram: https://goo.gl/YJfNgf

2. Given some list `lst` of numbers, mutate `lst` to have the accumulated sum of all elements so far in the list. If `lst` is a deep list, mutate it to similarly reflect the accumulated sum of all elements so far in the nested list.

   Hint: The **`isinstance`** function returns True for **`isinstance(l, list)`** if `l` is a list and False otherwise.

```
def accumulate(lst):
    """
    >>> l = [1, 5, 13, 4]
    >>> accumulate(l)
    23
    >>> l
    [1, 6, 19, 23]
    >>> deep_l = [3, 7, [2, 5, 6], 9]
    >>> accumulate(deep_l)
    32
    >>> deep_l
    [3, 10, [2, 7, 13], 32]
    """
    sum_so_far = 0
    for _____________________________________________:

        ____________________________________________

        if isinstance(_____________________, list):

            inside = _____________________________

            ____________________________________

        else:
            ____________________________________

            ____________________________________
```

```
def accumulate(lst):
    sum_so_far = 0
    for i in range(len(lst)):
        item = lst[i]
        if isinstance(item, list):
            inside = accumulate(item)
            sum_so_far += inside
        else:
            sum_so_far += item
            lst[i] = sum_so_far
    return sum_so_far
```