CS 61A Summer 2026

Sequences, Trees

Discussion 4: July 7, 2026

Lists

Some of you already know list operations that we haven't covered yet, such as `append`. Don't use those today. All you need are list literals (e.g., `[1, 2, 3]`), item selection (e.g., `s[0]`), list addition (e.g., `[1] + [2, 3]`), `len` (e.g., `len(s)`), and slicing (e.g., `s[1:]`). Use those! There will be plenty of time for other list operations when we introduce them later this week.

The most important thing to remember about lists is that a non-empty list `s` can be split into its first element `s[0]` and the rest of the list `s[1:]`.

```
>>> s = [2, 3, 6, 4]
>>> s[0]
2
>>> s[1:]
[3, 6, 4]
```

A list comprehension describes the elements in a list and evaluates to a new list containing those elements.

There are two forms:

```
[<expression> for <element> in <sequence>]
[<expression> for <element> in <sequence> if <conditional>]
```

Here's an example that starts with `[1, 2, 3, 4]`, picks out the even elements `2` and `4` using `if i % 2 == 0`, then squares each of these using `i*i`. The purpose of `for i` is to give a name to each element in `[1, 2, 3, 4]`.

```
>>> [i*i for i in [1, 2, 3, 4] if i % 2 == 0]
[4, 16]
```

This list comprehension evaluates to a list of:

- The value of `i*i`
- For each element `i` in the sequence `[1, 2, 3, 4]`
- For which `i % 2 == 0`

In other words, this list comprehension will create a new list that contains the square of every even element of the original list `[1, 2, 3, 4]`.

We can also rewrite a list comprehension as an equivalent `for` statement, such as for the example above:

```
>>> result = []
>>> for i in [1, 2, 3, 4]:
...     if i % 2 == 0:
...         result = result + [i*i]
>>> result
[4, 16]
```

Q1: Even weighted

Write a function that takes a list s and returns a new list that keeps only the even-indexed elements of s and multiplies them by their corresponding index. First approach this problem with a normal for loop (without list comprehension).

```
def even_weighted_loop(s):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted_loop(x)
    [0, 6, 20]
    """
    "*** YOUR CODE HERE ***"
```

Now that you've done it with a for loop, try it with a list comprehension!

```
def even_weighted_comprehension(s):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted_comprehension(x)
    [0, 6, 20]
    """
    return [_________________________________________________]
```

Dictionaries

A dictionary is a Python data structure that maps *keys* to *values*. For a dictionary d:

- Each key maps to exactly one value, which you can access with d[<key>].
- If you try d[<key>] but <key> is not a valid key in d, you'll get an error.
- You can use <key> in d to test (True or False) whether a key is in d.
- d.get(<key>, <default>) will return the value that <key> maps to in d, or <default> if <key> is not a key in d.
- The sequence of keys or values or key-value pairs can be accessed using d.keys() or d.values() or d.items(), respectively.
- Keys can be of any immutable type (like strings, numbers, and tuples) but not mutable types (like lists).
- You can use a d in a for loop (such as for k in d). Doing so will iterate through the keys in d.

Q2: Happy Givers

In a certain discussion section, some people exchange gifts for the holiday season. We call two people **happy givers** if they give gifts to each other. Implement a function `happy_givers`, which takes in a `gifts` dictionary that maps people in the section to the person they gave a gift to. `happy_givers` outputs a list of all the happy givers in the section. The order of the list does not matter.

Note that if someone received but did not give a gift, they will not appear in the `gifts` dictionary as a key. (They'll appear only as a value.) You can assume that no one gives themself a gift.

Once you've found a solution, as a challenge, attempt to implement a solution in one line using a list comprehension.

Optional: Imagine an alternate case where the dictionary where each person is a key, and its value is a list of all the people they gave a gift to. Attempt to implement a solution to find a list of all happy givers with this configuration.

```
def happy_givers(gifts):
    """
    >>> gift_recipients = {
    ...     "Alice": "Eve", # Alice gave a gift to Eve
    ...     "Bob": "Finn",
    ...     "Christina": "Alice",
    ...     "David": "Gina", # Gina is not a key because she didn't give anyone a gift
    ...     "Eve": "Alice",
    ...     "Finn": "Bob",
    ... }
    >>> list(sorted(happy_givers(gift_recipients))) # Order does not matter
    ['Alice', 'Bob', 'Eve', 'Finn']
    """
    "*** YOUR CODE HERE ***"
```

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

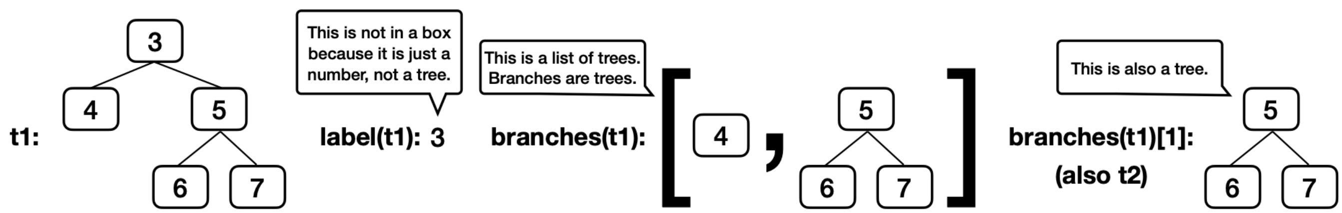
Trees

For a tree `t`:

- Its root label can be any value, and `label(t)` returns it.
- Its branches are trees, and `branches(t)` returns a list of branches.
- An identical tree can be constructed with `tree(label(t), branches(t))`.
- You can call functions that take trees as arguments, such as `is_leaf(t)`.
- That's how you work with trees. No `t == x` or `t[0]` or `x in t` or `list(t)`, etc.
- There's no way to change a tree (that doesn't violate an abstraction barrier).

Here's an example tree `t1`, for which its branch `branches(t1)[1]` is `t2`.

```
t2 = tree(5, [tree(6), tree(7)])
t1 = tree(3, [tree(4), t2])
```



Example Tree

A path is a sequence of trees in which each is the parent of the next.

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_leaf(tree):
    return not branches(tree)

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
```

Q3: Warm Up

What value is bound to `result`?

```
result = label(min(branches(max([t1, t2], key=label)), key=label))
```

How convoluted!

Here's a quick refresher on how `key` functions work with `max` and `min`,

`max(s, key=f)` returns the item `x` in `s` for which `f(x)` is largest.

```
>>> s = [-3, -5, -4, -1, -2]
>>> max(s)
-1
>>> max(s, key=abs)
-5
>>> max([abs(x) for x in s])
5
```

Therefore, `max([t1, t2], key=label)` returns the tree with the largest label, in this case `t2`.

In case you're wondering, this expression does not violate an abstraction barrier. `[t1, t2]` and `branches(t)` are both lists (not trees), and so it's fine to call `min` and `max` on them.

Q4: Has Path

Implement `has_path`, which takes a tree `t` and a list `p`. It returns whether there is a path from the root of `t` with labels `p`. For example, `t1` has a path from its root with labels `[3, 5, 6]` but not `[3, 4, 6]` or `[5, 6]`.

Important: Before trying to implement this function, discuss these questions from lecture about the recursive call of a tree processing function: - What small initial choice can I make (such as which branch to explore)? - What recursive call should I make for each option? - How can I combine the results of those recursive calls? - What type of values do they return? - What do those return values mean?

```
def has_path(t, p):
    """Return whether tree t has a path from the root with labels p.

    >>> t2 = tree(5, [tree(6), tree(7)])
    >>> t1 = tree(3, [tree(4), t2])
    >>> has_path(t1, [5, 6])        # This path is not from the root of t1
    False
    >>> has_path(t2, [5, 6])        # This path is from the root of t2
    True
    >>> has_path(t1, [3, 5])        # This path does not go to a leaf, but that's ok
    True
    >>> has_path(t1, [3, 5, 6])     # This path goes to a leaf
    True
    >>> has_path(t1, [3, 4, 5, 6])  # There is no path with these labels
    False
    """
    if p == ____:  # when len(p) is 1
        return True
    elif label(t) != ____:
        return False
    else:
        "*** YOUR CODE HERE ***"
```

Q5: Find Path

Implement `find_path`, which takes a tree `t` with unique labels and a value `x`. It returns a list containing the labels of the nodes along a path from the root of `t` to a node labeled `x`.

If `x` is not a label in `t`, return `None`. Assume that the labels of `t` are unique.

```
def find_path(t, x):
    """
    >>> t2 = tree(5, [tree(6), tree(7)])
    >>> t1 = tree(3, [tree(4), t2])
    >>> find_path(t1, 5)
    [3, 5]
    >>> find_path(t1, 4)
    [3, 4]
    >>> find_path(t1, 6)
    [3, 5, 6]
    >>> find_path(t2, 6)
    [5, 6]
    >>> print(find_path(t1, 2))
    None
    """
    if _______________________________________:
        return _______________________________________
    _______________________________________:
        path = _______________________________________
        if path:
            return _______________________________________
    return None
```

Q6: Pruning Leaves

Implement `prune_leaves`, which takes a tree `t` and a tuple of values `vals`. It returns a version of `t` with all its **leaves** whose labels are in `vals` removed. Do not remove non-leaf nodes and do not remove leaves that do not match any of the items in `vals`. Return `None` if pruning the tree results in there being no nodes left in the tree.

```
def prune_leaves(t, vals):
    """Return a version of t (a new tree) with all leaves that have a label
    that appears in vals removed. Return None if the entire tree is
    pruned away.

    >>> t = tree(2)
    >>> print(prune_leaves(t, (1, 2)))
    None
    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7)])])
    >>> print_tree(numbers)
    1
      2
      3
        4
        5
      6
        7
    >>> print_tree(prune_leaves(numbers, (3, 4, 6, 7)))
    1
      2
      3
        5
      6
    """
    "*** YOUR CODE HERE ***"
```