

Getting Started

In this discussion, don't use a Python interpreter to run code until you are confident your solution is correct. Figure things out and check your work by *thinking* about what your code will do. Not sure? Draw an environment diagram!

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]
    """
    result = []
    for i in range(len(s)):
        if i % 2 == 0:
            result = result + [i * s[i]]
    return result
```

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 [i * s[i] for i in range(len(s)) if i % 2 == 0]
```

The key point to note is that instead of iterating over each element in the list, we must instead iterate over the indices of the list. Otherwise, there's no way to tell if we should keep a given element.

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 themselves a gift.

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

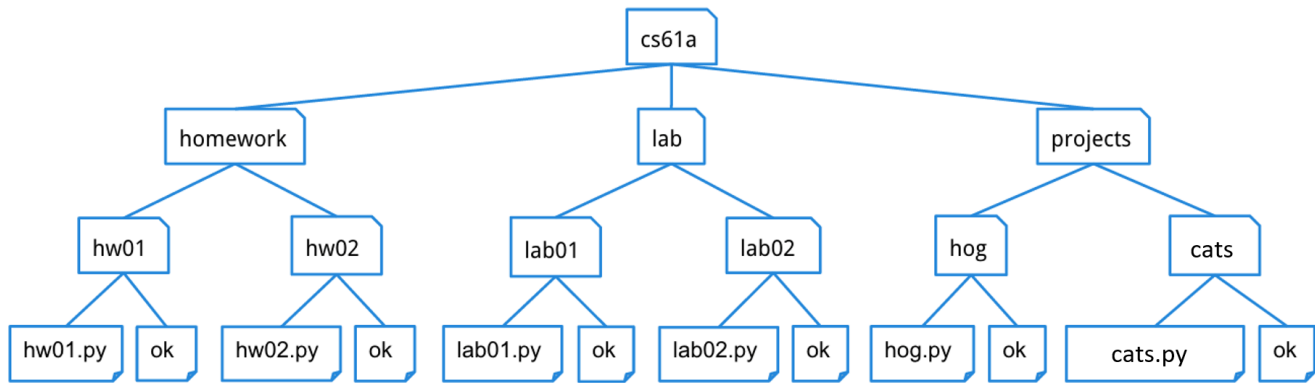
```
def happy_givers(gifts):
    """
    >>> gift_recipients = {
    ...     "Alice": "Eve", # Alice gave a gift to Eve
    ...     "Bob": "Finn",
    ...     "Christina": "Alice",
    ...     "David": "Gina", # Gina is not a key because she didn't give anyone a gift
    ...     "Eve": "Alice",
    ...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

    output = []
    for giver in gifts:
        recipient = gifts[giver]
        if recipient in gifts and gifts[recipient] == giver:
            output = output + [giver]
    return output

# Alternate solution using list comprehension
# return [giver for giver in gifts if gifts[giver] in gifts and gifts[ gifts[giver] ]
# == giver]
```

Trees

A **tree** is a data structure that represents a hierarchy of information. A file system is a good example of a tree structure. For example, within your **cs61a** folder, you have folders separating your **projects**, **lab** assignments, and **homework**. The next level is folders that separate different assignments, **hw01**, **lab01**, **hog**, etc., and inside those are the files themselves, including the starter files and **ok**. Below is an incomplete diagram of what your **cs61a** directory might look like.



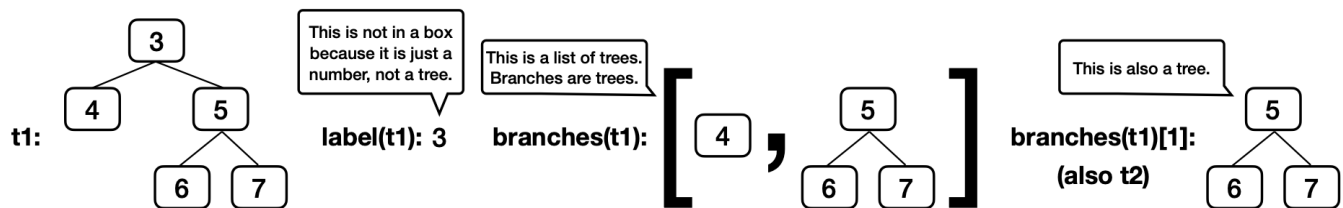
cs61a_tree

As you can see, unlike trees in nature, the tree abstract data type is drawn with the root at the top and the leaves at the bottom.

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.

You don't need to know how **tree**, **label**, and **branches** are implemented in order to use them correctly, but here is the implementation from lecture.

```

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: 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
    """
    # Non-list comprehension solution
    if is_leaf(t):
        return label(t)
    highest_sum = 0
    for b in branches(t):
        highest_sum = max(max_path_sum(b), highest_sum)
    return label(t) + highest_sum

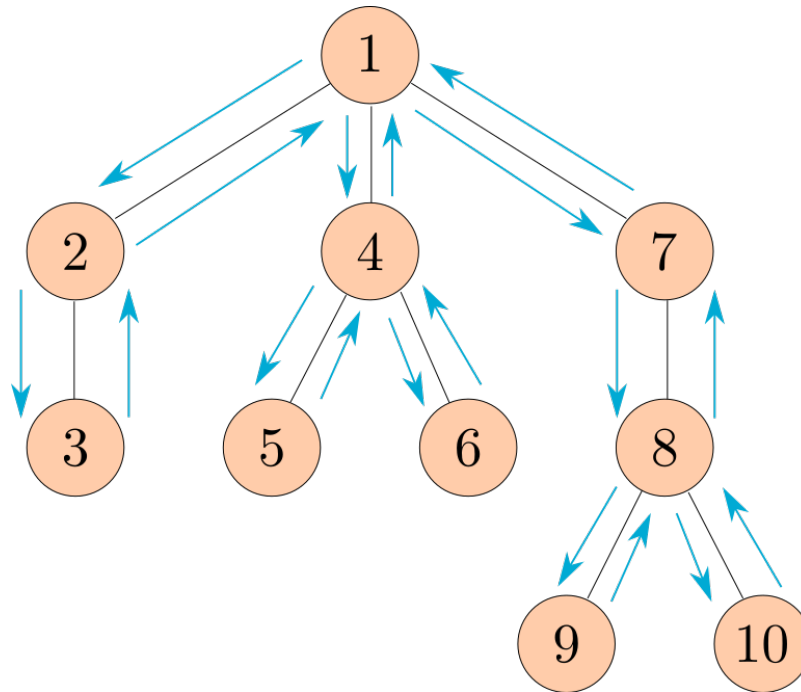
    # List comprehension solution
    if is_leaf(t):
        return label(t)
    else:
        return label(t) + max([max_path_sum(b) for b in branches(t)])

```

Q4: Preorder

Define the function `preorder`, which takes in a tree as an argument and returns a list of all the entries in the tree in the order that `print_tree` would print them.

The following diagram shows the order that the nodes would get printed, with the arrows representing function calls.

**Preorder Traversal**

Note: This ordering of the nodes in a tree is called a preorder traversal.

```

def preorder(t):
    """Return a list of the entries in this tree in the order that they
    would be visited by a preorder traversal (see problem description).

    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7)])])
    >>> preorder(numbers)
    [1, 2, 3, 4, 5, 6, 7]
    >>> preorder(tree(2, [tree(4, [tree(6)])]))
    [2, 4, 6]
    """
    flattened_branches = []
    for child in branches(t):
        flattened_branches += preorder(child)
    return [label(t)] + flattened_branches

# Alternate solution
from functools import reduce

def preorder_alt(t):
    return reduce(add, [preorder_alt(child) for child in branches(t)], [label(t)])

```


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 label(t) == x:
        return [label(t)]
    for b in branches(t):
        path = find_path(b, x)
        if path:
            return [label(t)] + path
    return None
```

The base case return value `[label(t)]` creates a one-element list of the labels along a path that starts at the root of `t` and also ends there, since the root is labeled `x`.

The assignment `path = find_path(b, x)` allows the return value of this recursive call to be used twice: once to check if it's `None` (which is a false value) and again to build a longer list.

The expression `[label(t)] + path` for a tree `t` and list `path` creates a longer list that starts with the label of `t` and continues with the elements of `path`.

You're done! Excellent work this week. Please be sure to fill out your TA's attendance form to get credit for this discussion!