Lab 5: Trees, Data Abstraction, Python Lists **lab05.zip (lab05.zip)**

Due by 11:59pm on Wednesday, September 28.

Starter Files

Download lab05.zip (lab05.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Data Abstraction

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:

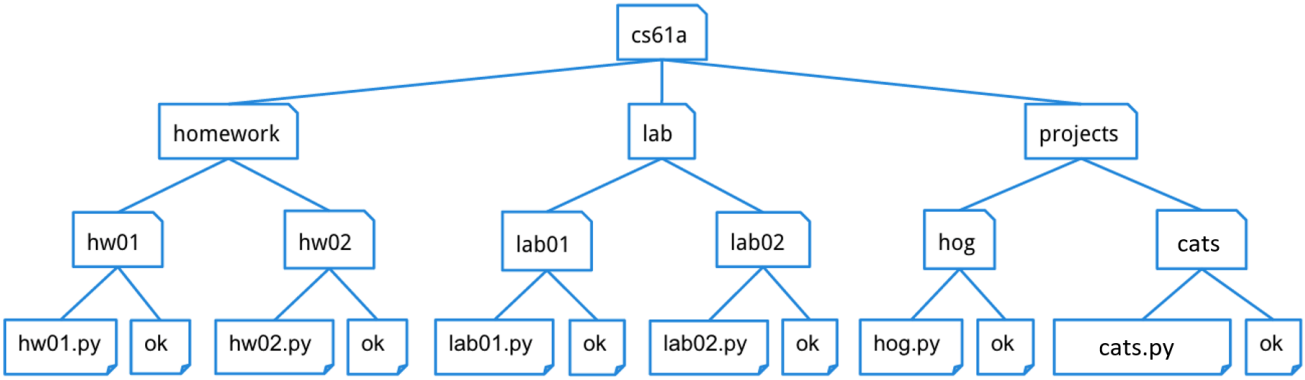
- **Constructors**: functions that build the abstract data type.
- **Selectors**: functions that retrieve information from the data type.

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

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.



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.

Some tree terminology:

- **root**: the node at the top of the tree
- **label**: the value in a node
- **branches**: a list of trees directly under the tree's root
- **leaf**: a tree with zero branches
- **node**: any location within the tree (e.g., root node, leaf nodes, etc.)

Our `tree` abstract data type consists of a root and a list of its `branches`. To create a tree and access its root value and branches, use the following constructor and selectors:

- Constructor
  - `tree(label, branches=[])`: creates a tree object with the given `label` value at its root node and list of `branches`. Notice that the second argument to this constructor, `branches`, is optional - if you want to make a tree with no branches, leave this argument empty.
- Selectors

    - `label(tree)`: returns the value in the root node of `tree`.
    - `branches(tree)`: returns the list of branches of the given `tree`.
- Convenience function
    - `is_leaf(tree)`: returns `True` if `tree`'s list of `branches` is empty, and `False` otherwise.

For example, the tree generated by

```
number_tree = tree(1,
         [tree(2),
          tree(3,
               [tree(4),
                tree(5)]),
          tree(6,
               [tree(7)])])
```

would look like this:

```
   1
 / | \
2  3  6
  / \  \
 4   5  7
```

To extract the number `3` from this tree, which is the label of the root of its second branch, we would do this:

```
label(branches(number_tree)[1])
```

The `print_tree` function prints out a tree in a human-readable form. The exact form follows the pattern illustrated above, where the root is unindented, and each of its branches is indented one level further.

```
def print_tree(t, indent=0):
    """Print a representation of this tree in which each node is
    indented by two spaces times its depth from the root.

    >>> print_tree(tree(1))
    1
    >>> print_tree(tree(1, [tree(2)]))
    1
      2
    >>> 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('  ' * indent + str(label(t)))
    for b in branches(t):
        print_tree(b, indent + 1)
```

Required Questions

Getting Started Videos

Lists

Q1: Flatten

Write a function `flatten` that takes a list and returns a "flattened" version of it. The list could be a deep list, meaning that there could be a multiple layers of nesting within the list.

Make sure your solution does not mutate the input list.

For example, one use case of `flatten` could be the following:

```
>>> lst = [1, [[2], 3], 4, [5, 6]]
>>> flatten(lst)
[1, 2, 3, 4, 5, 6]
```

Hint: you can check if something is a list by using the built-in `type` function. For example:

```
>>> type(3) == list
False
>>> type([1, 2, 3]) == list
True
```

```
def flatten(s):
    """Returns a flattened version of list s.

    >>> flatten([1, 2, 3])     # normal list
    [1, 2, 3]
    >>> x = [1, [2, 3], 4]     # deep list
    >>> flatten(x)
    [1, 2, 3, 4]
    >>> x # Ensure x is not mutated
    [1, [2, 3], 4]
    >>> x = [[1, [1, 1]], 1, [1, 1]] # deep list
    >>> flatten(x)
    [1, 1, 1, 1, 1, 1]
    >>> x
    [[1, [1, 1]], 1, [1, 1]]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q flatten
```

Data Abstraction

Say we have an abstract data type for cities. A city has a name, a latitude coordinate, and a longitude coordinate.

Our data abstraction has one **constructor**:

- `make_city(name, lat, lon)` : Creates a city object with the given name, latitude, and longitude.

We also have the following **selectors** in order to get the information for each city:

- `get_name(city)` : Returns the city's name
- `get_lat(city)` : Returns the city's latitude
- `get_lon(city)` : Returns the city's longitude

Here is how we would use the constructor and selectors to create cities and extract their information:

```
>>> berkeley = make_city('Berkeley', 122, 37)
>>> get_name(berkeley)
'Berkeley'
>>> get_lat(berkeley)
122
>>> new_york = make_city('New York City', 74, 40)
>>> get_lon(new_york)
40
```

All of the selector and constructor functions can be found in the lab file, if you are curious to see how they are implemented. However, the point of data abstraction is that we do not need to know how an abstract data type is implemented, but rather just how we can interact with and use the data type.

Q2: Distance

We will now implement the function `distance`, which computes the distance between two city objects. Recall that the distance between two coordinate pairs `(x1, y1)` and `(x2, y2)` can be found by calculating the `sqrt` of `(x1 - x2)**2 + (y1 - y2)**2`. We have already imported `sqrt` for your convenience. Use the latitude and longitude of a city as its coordinates; you'll need to use the selectors to access this info!

```
from math import sqrt
def distance(city_a, city_b):
    """
    >>> city_a = make_city('city_a', 0, 1)
    >>> city_b = make_city('city_b', 0, 2)
    >>> distance(city_a, city_b)
    1.0
    >>> city_c = make_city('city_c', 6.5, 12)
    >>> city_d = make_city('city_d', 2.5, 15)
    >>> distance(city_c, city_d)
    5.0
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q distance
```

Q3: Closer city

Next, implement `closer_city`, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is relatively closer to the provided latitude and longitude.

You may only use the selectors and constructors introduced above and the `distance` function you just defined for this question.

> **Hint**: How can you use your `distance` function to find the distance between the given location and each of the given cities?

```
def closer_city(lat, lon, city_a, city_b):
    """
    Returns the name of either city_a or city_b, whichever is closest to
    coordinate (lat, lon). If the two cities are the same distance away
    from the coordinate, consider city_b to be the closer city.

    >>> berkeley = make_city('Berkeley', 37.87, 112.26)
    >>> stanford = make_city('Stanford', 34.05, 118.25)
    >>> closer_city(38.33, 121.44, berkeley, stanford)
    'Stanford'
    >>> bucharest = make_city('Bucharest', 44.43, 26.10)
    >>> vienna = make_city('Vienna', 48.20, 16.37)
    >>> closer_city(41.29, 174.78, bucharest, vienna)
    'Bucharest'
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q closer_city
```

Q4: Don't violate the abstraction barrier!

> Note: this question has no code-writing component (if you implemented the previous two questions correctly).

When writing functions that use an data abstraction, we should use the constructor(s) and selector(s) whenever possible instead of assuming the data abstraction's implementation. Relying on a data abstraction's underlying implementation is known as *violating the abstraction barrier*, and we never want to do this!

It's possible that you passed the doctests for the previous questions even if you violated the abstraction barrier. To check whether or not you did so, run the following command:

Use Ok to test your code:

```
python3 ok -q check_city_abstraction
```

The `check_city_abstraction` function exists only for the doctest, which swaps out the implementations of the original abstraction with something else, runs the tests from the previous two parts, then restores the original abstraction.

The nature of the abstraction barrier guarantees that changing the implementation of an data abstraction shouldn't affect the functionality of any programs that use that data abstraction, as long as the constructors and selectors were used properly.

If you passed the Ok tests for the previous questions but not this one, the fix is simple! Just replace any code that violates the abstraction barrier with the appropriate constructor or selector.

Make sure that your functions pass the tests with both the first and the second implementations of the data abstraction and that you understand why they should work for both before moving on.

Trees

Q5: Finding Berries!

The squirrels on campus need your help! There are a lot of trees on campus and the squirrels would like to know which ones contain berries. Define the function `berry_finder`, which takes in a tree and returns `True` if the tree contains a node with the value `'berry'` and `False` otherwise.

> **Hint**: To iterate through each of the branches of a particular tree, you can consider using a `for` loop to get each branch.

```
def berry_finder(t):
    """Returns True if t contains a node with the value 'berry' and
    False otherwise.

    >>> scrat = tree('berry')
    >>> berry_finder(scrat)
    True
    >>> sproul = tree('roots', [tree('branch1', [tree('leaf'), tree('berry')]), tree('l
    >>> berry_finder(sproul)
    True
    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7)])])
    >>> berry_finder(numbers)
    False
    >>> t = tree(1, [tree('berry',[tree('not berry')])])
    >>> berry_finder(t)
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q berry_finder
```

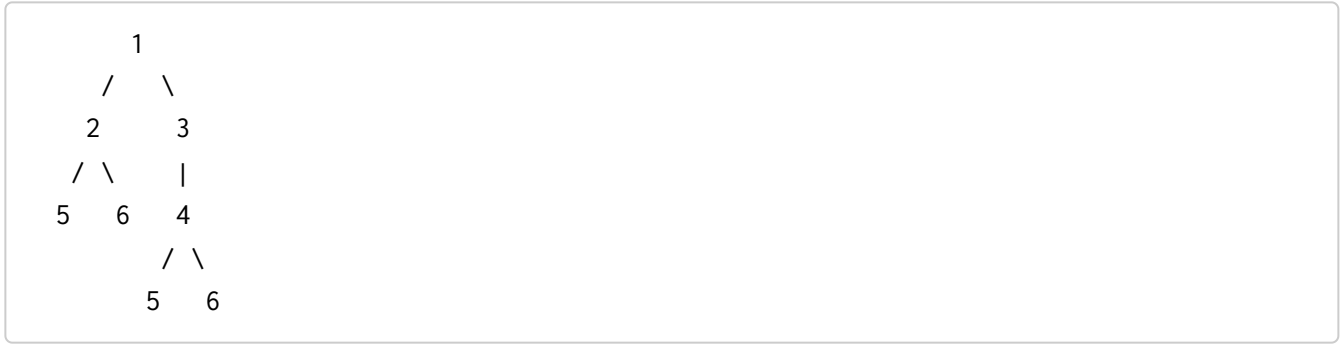
Q6: Sprout leaves

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:




```
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 ***"
```

Use Ok to test your code:

```
python3 ok -q sprout_leaves
```

Q7: Don't violate the abstraction barrier!

> Note: this question has no code-writing component (if you implemented the previous two questions correctly).

When writing functions that use an data abstraction, we should use the constructor(s) and selector(s) whenever possible instead of assuming the data abstraction's implementation. Relying on a data abstraction's underlying implementation is known as *violating the abstraction barrier*, and we never want to do this!

It's possible that you passed the doctests for the previous questions even if you violated the abstraction barrier. To check whether or not you did so, run the following command:

Use Ok to test your code:

```
python3 ok -q check_abstraction
```

The `check_abstraction` function exists only for the doctest, which swaps out the implementations of the original abstraction with something else, runs the tests from the previous two parts, then restores the original abstraction.

The nature of the abstraction barrier guarantees that changing the implementation of an data abstraction shouldn't affect the functionality of any programs that use that data abstraction, as long as the constructors and selectors were used properly.

If you passed the Ok tests for the previous questions but not this one, the fix is simple! Just replace any code that violates the abstraction barrier with the appropriate constructor or selector.

Make sure that your functions pass the tests with both the first and the second implementations of the data abstraction and that you understand why they should work for both before moving on.

Submit

Make sure to submit this assignment by running:

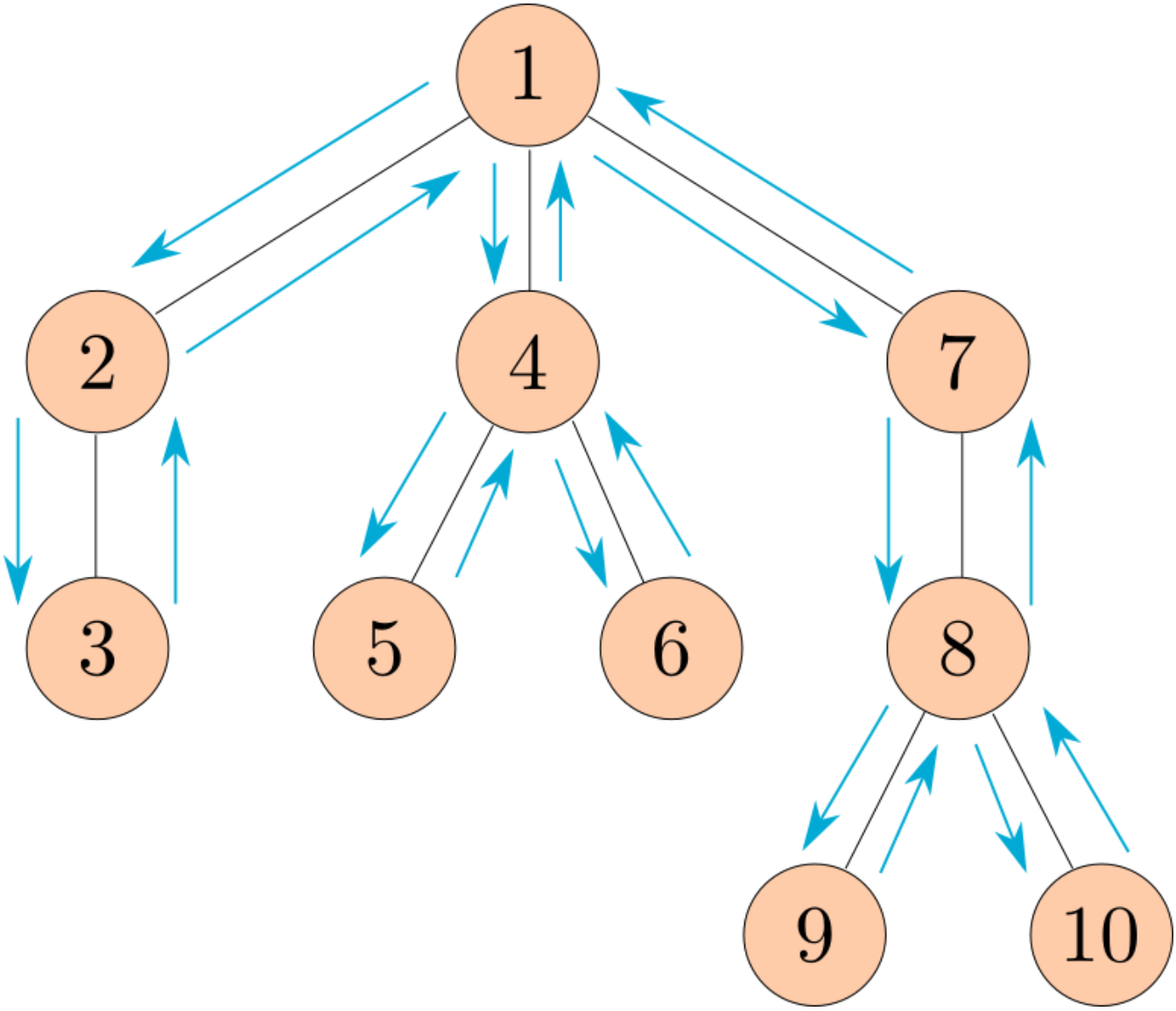
```
python3 ok --submit
```

Optional Questions

Q8: 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.



> 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]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q preorder
```

Q9: Add trees

Define the function `add_trees`, which takes in two trees and returns a new tree where each corresponding node from the first tree is added with the node from the second tree. If a node at any particular position is present in one tree but not the other, it should be present in the new tree as well. At each level of the tree, nodes correspond to each other starting from the leftmost node.

> *Hint*: You may want to use the built-in zip function to iterate over multiple sequences at once.
>
> *Note*: If you feel that this one's a lot harder than the previous tree problems, that's totally fine! This is a pretty difficult problem, but you can do it! Talk about it with other students, and come back to it if you need to.

```
def add_trees(t1, t2):
    """
    >>> numbers = tree(1,
    ...                [tree(2,
    ...                      [tree(3),
    ...                       tree(4)]),
    ...                 tree(5,
    ...                      [tree(6,
    ...                            [tree(7)]),
    ...                       tree(8)])])
    >>> print_tree(add_trees(numbers, numbers))
    2
      4
        6
        8
      10
        12
          14
        16
    >>> print_tree(add_trees(tree(2), tree(3, [tree(4), tree(5)])))
    5
      4
      5
    >>> print_tree(add_trees(tree(2, [tree(3)]), tree(2, [tree(3), tree(4)])))
    4
      6
      4
    >>> print_tree(add_trees(tree(2, [tree(3, [tree(4), tree(5)])]), \
    tree(2, [tree(3, [tree(4)]), tree(5)])))
    4
      6
        8
        5
      5
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q add_trees
```