Worksheet 4:

Lists and Trees

Review: Lists

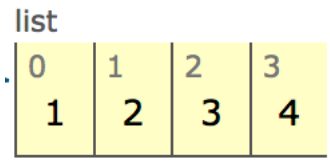
Lists are containers for values.

Eg.

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

Is the list containing the numbers 1, 2, 3, 4

A box and pointer diagram for the above list would look like

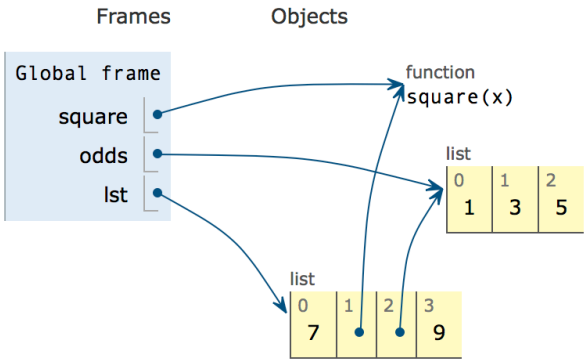


The great thing about lists is that lists can contain any type of value, whether it be numbers, booleans, functions, or even other lists.

Eg.

```
def square (x):
        return x*x
odds = [1, 3, 5]
lst = [7, square, odds, 9]
```

What would the environment diagram look like?



We can also get the values of any item in each list using bracket notation.

Using the above example,

>>> lst[0]

7

>>> lst[2][2]

5

Since lst[2] is a list, we can index into it using a second set of brackets.

There are also some important list operations:

Pop

Pop REMOVES an element from a list and RETURNS it.

```
>>> lst = [1, 2, 3]
>>> lst.pop() #removes the last element
3
>>> lst.pop(0) #removes the element at index 0
1
>>> lst
[2]
```

Remove

Remove REMOVES the element specified from a list and RETURNS None

```
>>> lst = [1, 2, 3, 2]
>>> lst.remove(2) #removes the first occurence of the number 2
>>> y = lst.remove(2) #removes the first occurence of the number 2
>>> y == None
True
>>> lst
[1, 3]
```

Append

Append ADDS the element specified to a list and RETURNS None

```
>>> lst = [1, 2, 3]
>>> lst.append(4) #adds the number 4 to the end of lst
>>> y = lst.append([5, 6]) #adds the list as the 5th item of lst
>>> y == None
True
>>> lst
[1, 2, 3, 4, [5, 6]]
```

Extend

Extend ADDS the items specified in the list passed in to a list and RETURNS None

```
>>> lst = [1, 2, 3]
>>> lst.extend([4]) #adds the number 4 to the end of lst
>>> y = lst.extend([5, 6]) #adds the numbers 5 and 6 to the end of lst
>>> y == None
True
>>> lst
[1, 2, 3, 4, 5, 6]
```

Practice:

Draw the box and pointer diagrams for the following lines of code

```
bros = ["bro1", "bro2", "bro3"]
frat = ["theta", "kappa", ["tau", "psi"], "beta"]
frat[2][0] = bros.pop()
bros.append(frat)
frat.extend(frat[2])
bros.remove("bro1")
```

```
sisters = ["sis1", "sis2", "sis3"]
srat = ["alpha", "gamma", ["phi", "phi", sisters], "phi"]
srat[2].remove("phi")
sisters.append(srat[3])
sisters.extend(sisters)
sisters[0] = srat[2][1].pop(6)
```

List Comprehension

We can use for loops to iterate through lists!

For Loops:

For loops follow the form of

```
for <name> in <list>
```

where <name> is the name of a variable and <list> is the list we iterate over. Let's look at an example

```
>>> lst = [1, 2, 3]
>>> x = 0
>>> for y in lst:
        x += y
>>> x
6
```

Here, the for loop goes through each item in **lst** and adds them to **x**. How does it work? The for loop assigns the first value of the list to **y** and executes the body of the for loop. At the end of the for loop, **y** is reassigned to the next item in the list, and the process repeats. The contents of **lst** never changes.

List Comprehension

We can make new lists by iterating over old lists using list comprehension!

We use the form:

```
<expr applied to each item> for <name> in <list> if <predicate>
```

Where the if statement is optional.

Let's look at an example:

```
>>> lst = [1, 2, 3]
>>> lst2 = [x+1 for x in lst if x%2 == 0]
>>> lst2
[1, 3, 3]
>>> lst
[1, 2, 3]
```

What is happening here?

We are iterating over the items of **lst.** If x%2 == 0 becomes true, we would add 1 to the value in **lst** and place it in **lst2.** Notice **lst** stays the same, since we are making a NEW list.

What would happen if we took out the if clause?

We would add one to each value, since there is no predicate condition.

```
>>> lst = [1, 2, 3]
>>> lst2 = [x+1 for x in lst]
>>> lst2
[2, 3, 4]
>>> lst
[1, 2, 3]
```

Practice:

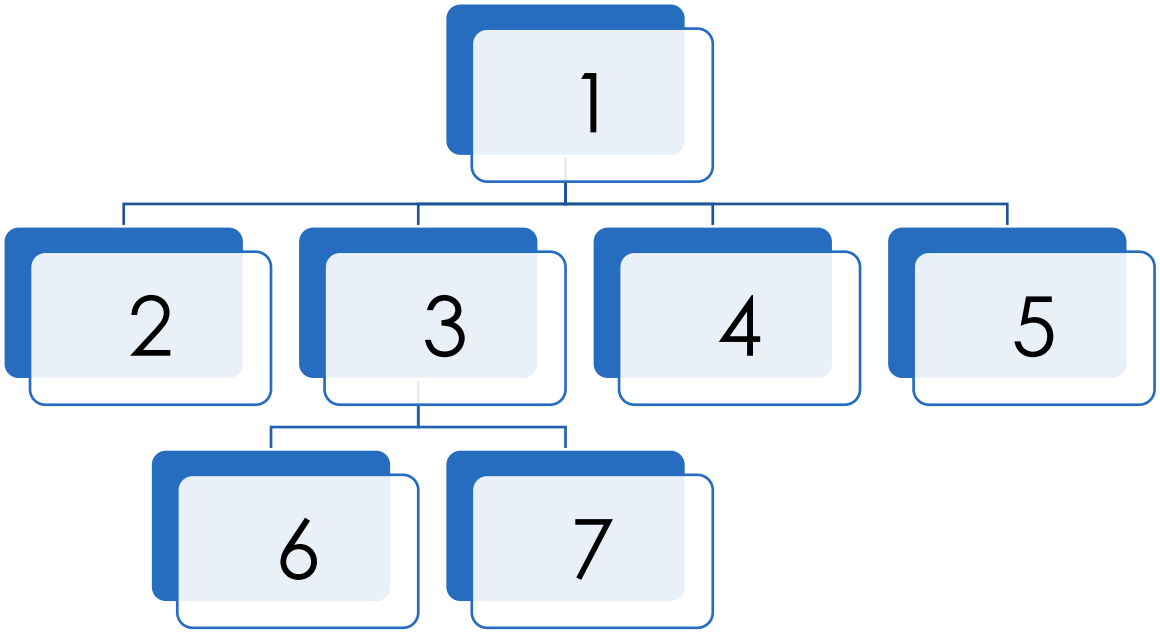
WWPD

```
>>> [2*x for x in [2, 4, 6, 8] if x%2 != 0]


>>> ["enigma" for b in ["i", "love", "cs"] if b == "love"]


>>> [fun%work for fun, work in [[1, 2], [3, 4], [10, 2]]]
```

Trees

Trees are great!

Let's look at an example tree.



Definitions:

Parent: Any node that has branches. Only one parent per child.

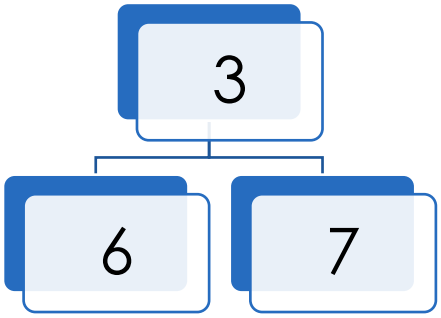
Child: Any node with a parent.

Root: The top node of a tree. (In this example, 1 is the root)

Label: The value at each node.

Leaf: Any node with NO branches. Leaves in this example include 2, 6, 7, 4, and 5.

Branch: A sub-tree of the root. Each branch can be described as a tree itself! This makes trees recursive structures. For example, the 2nd branch of the above tree, can be described as another tree:



Depth: How far away any given node is away from the root. 1 has a depth of 0, while 7 has a depth of 2.

Height: The depth of the lowest leaf. Therefore, the original tree has a height of 2.

So how can we implement this amazing structure in python? We can use data abstraction!

```
# Constructors
def tree(label, branches=[]):
        for branch in branches:
                assert is_tree(branch)
        return [label] + list(branches)

# Selectors
def label(tree):
        return tree[0]
def branches(tree):
        return tree[1:]

# For convenience
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
```

So how can we start creating trees?

Let's make the given tree above.

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

or, in a straight line

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

So, what can we even do with these trees? Turns out, a lot of different things. Let's do an example.

Eg.

Create a function that multiplies the value at each node by 2.

How can we complete this without breaking any abstraction barriers?

```
def double_trouble(t):
"""Returns a tree where the values of each node in t are multiplied by 2.
        >>> t = tree(1, [tree(2), tree(3, [tree(6), tree(7)]), tree(4),
        tree(5)])
        >>> t2 = tree(2, [tree(4), tree(6, [tree(12), tree(14)]), tree(8),
       tree(10)])
       >>> check_t = double_trouble(t)
      >>> t2 == check_t
       True
       """
```

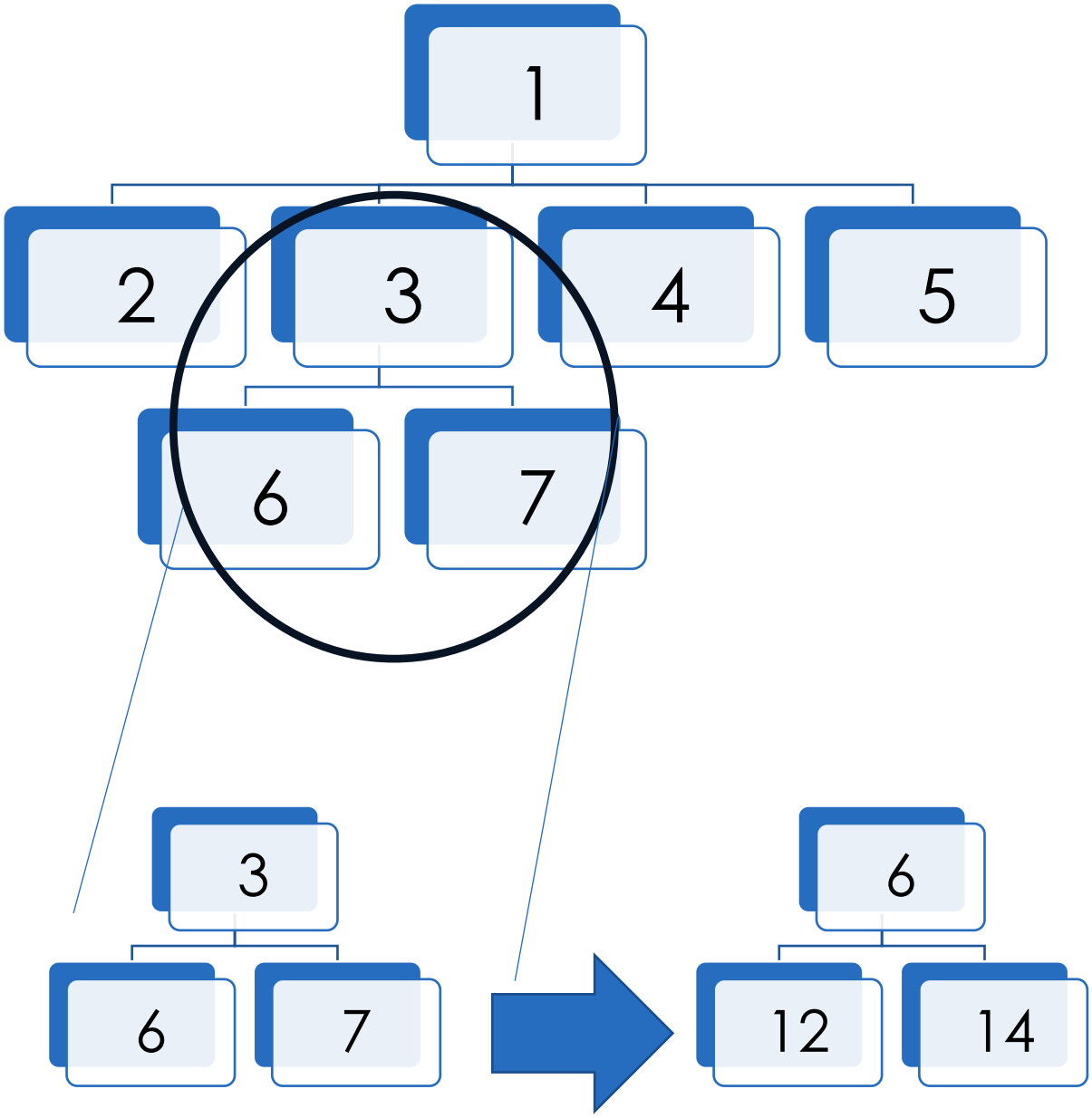
We know this will be recursive, so we first need a BASE CASE:

We know that if we pass in just a leaf, it will double the label, and return the new leaf. Therefore, our base case is:

```
if (is_leaf(t)):
        return tree(label(t) * 2)
```

Now, we need to ensure that this works for each branch of the tree. We can take the RECURSIVE LEAP OF FAITH, and apply **double_trouble** to each branch of our tree.

This may be hard to visualize, so, let's take a look at our visualization of our tree.



If we assume that our function works, we can apply this function to each branch. Since each branch is just another tree, we can assume that calling the function on the branch will double all the values in the branch.

Since we know that branches are a list, we can use LIST COMPREHENSION to call **double_trouble** on each branch. It would look something like this:

```
return tree(label(t) * 2, [double_trouble(b) for b in branches(t)])
```

Where, we double the label of our tree, and then call **double_trouble** on each branch.

Our complete function would look like:

```
if (is_leaf(t)):
        return tree(label(t) * 2)
else:
        return tree(label(t) * 2, [double_trouble(b) for b in branches(t)])
```

However, there is a way to make this even simpler using the power of for loops!

Our solution actually looks like:

```
return tree(label(t) * 2, [double_trouble(b) for b in branches(t)])
```

What? What about the base case? Why does this work?

Well, let's see what happens if we call this function on a leaf, such as tree(2).

We would double the label, then call the recursive call on each branch of our tree. But wait! We don't have any branches in our tree. So, the for loop *never runs*. Therefore, a recursive call is never made, and we have hit a base case.

(Remember, we can have implicit base cases in ONLY CERTAIN tree problems. It always depends on what the problem is asking.)

Let's do some practice.

Practice:

```
def trim(t):
        """Returns a tree where the leaves are all 0.
        >>> t = tree(1, [tree(2), tree(3, [tree(6), tree(7)]), tree(4),
         tree(5)])
        >>> t2 = tree(1, [tree(0), tree(3, [tree(0), tree(0)]), tree(0),
         tree(0)])
        >>> check_t = trim(t)
        >>> t2 == check_t
        True
        """
```

```
def twos(t):
        """Checks if the tree has node values that are all multiples of 2.
        >>> t = tree(4, [tree(2), tree(22, [tree(6), tree(92)]), tree(10),
         tree(12)])
        >>> t2 = tree(1, [tree(3), tree(4, [tree(8), tree(0)]), tree(99),
         tree(10)])
        >>> twos(t)
        True
        >>> twos(t2)
        False
        """
        if _______________:

                return _________________

        else:

                for ___________________________________:

                        if ____________________:

                                ____________________

                ____________________
```