Worksheet 7

Mutable Trees, Linked Lists, Orders of Growth

Mutable Trees

Now that we have looked at object-oriented programming, let's look at something we have seen before that we can recreate using objects: Trees!

Let's just quickly look at the code for creating a tree object.

```python
class Tree:
    """A tree is a label and a list of branches."""
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        return '\n'.join(self.indented())

    def indented(self):
        lines = []
        for b in self.branches:
            for line in b.indented():
                lines.append('  ' + line)
        return [str(self.label)] + lines

    def is_leaf(self):
        return not self.branches
```

As you can see, ignoring methods like __repr__, __str__, and indented, (we will learn about these later), this is very similar to our data abstraction version of trees.

Our Tree class has instance attributes: label and branches

It also has a method is_leaf that checks if the Tree is a leaf or not.

You may be asking what's the point of using a Tree object when the abstraction version worked just fine.

With Tree objects, we can write functions that actually edit the tree that is inputted, instead of creating a new tree. Just like lists, these Tree objects are mutable!

Let's look at an example.

```python
def double_wubble(t):
    """Multiply all the entries in the Tree t by 2.
        >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4)])])
        >>> double_wubble(t)
        >>> t
        Tree(2, [Tree(4), Tree(6, [Tree(8)])])
        """
    t.label *= 2
    for b in t.branches:
        double_wubble(b)
```

Notice, there is no return statement. That is because the function EDITS the tree. In the doctest, we see that the function is called on a Tree object, and when we ask for t afterwards, the Tree has different values.

The object itself is changing, which is really interesting.

Let's do some Practice

Practice

```
def second_to_last(t):
    """Prune the second to last node in the tree. Do not prune the root.
        >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5), Tree(6,
[Tree(7)])])])
        >>> second_to_last(t)
        >>> t
        Tree(1, [Tree(2), Tree(4), Tree(5), Tree(7)])
        """
```

Linked Lists

Let's explore another mutable data structure: Linked Lists

Let's look at the code for it.

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(self.first, rest_str)

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ', '
            self = self.rest
        return string + str(self.first) + '>'
```

The basic idea behind a linked list is that each linked list has two items: a **value** (the instance attribute 'first'), and a **rest** (the instance attribute 'rest').

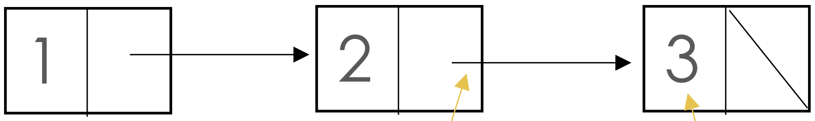
There are two rules when it comes to linked lists:

1. The **value** can be anything, including another Linked List!
2. The **rest** can ONLY be another Linked List, or null (or empty)

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

This is a python list. What would a linked list representation look like?

Let's look at an illustration first.



(The '\' represents null or empty.)

Each box has a left and right box. The left box is the value, in this case numbers. The right box is a pointer to another Linked List (or null).

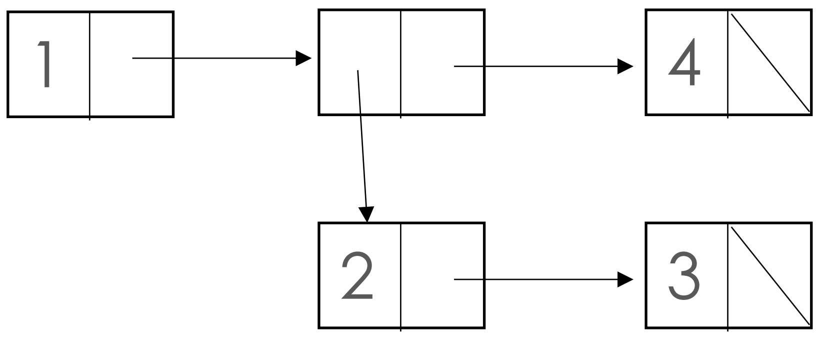
This may seem like an unnecessary waste of time, but it turns out Linked Lists can be incredibly useful.

Let's look at something called a deep linked list.

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

In Python, a deep list looks like this, where there is a list inside a list.

Remember, the **value** of a Linked List can be anything, including another linked list.



Here, the second value in our linked list, is another linked list!

Let's do a problem, similar to the tree question above, dealing with normal and deep linked lists.

Remember, Linked Lists are mutable, just like the mutable trees we saw before.

```
def doubly_linked(l):
    """Double the value of each linked list, including deep linked lists.
        >>> l = Link(1, Link(2, Link(3)))
        >>> doubly_linked(l)
        >>> l
        Link(2, Link(4, Link(6)))
        >>> l2 = Link(1, Link(Link(2, Link(3)), Link(4)))
        >>> doubly_linked(l2)
        >>> l2
        Link(2, Link(Link(4, Link(6)), Link(8)))
        """
    if l == Link.empty:
        return
    else:
        if isinstance(l.first, Link):
            doubly_linked(l.first)
        else:
            l.first *= 2
        doubly_linked(l.rest)
```

This method uses recursion!

Our base case checks if the linked list is empty or null. If it is not, then we first ensure that our value is not a linked list. If it is, we called the function recursively on the value.

If not, then we multiply the value by 2.

In the end, we have our recursive call on the rest of the linked list.

This is a MUTABLE function, so it edits the values in the linked list.

Let’s do a practice question

Practice

```
def backwards_forwards(l):
    """Make the linked lists a backwards version of itself. Remember, this is
        a mutable function. Assume the linked list is not circular.
        >>> l = Link(1, Link(2, Link(3)))
        >>> backwards_forwards(l)
        >>> l
        Link(3, Link(2, Link(1)))
        >>> l2 = Link(1, Link(Link(2, Link(3)), Link(4, Link(5))))
        >>> backwards_forwards(l2)
        >>> l2
        Link(5, Link(4, Link(Link(2, Link(3)), Link(1))))
        """
    beg, end = _____________, ___________________

    while (_____________________________):

        end = _______________________

    while (__________________________________________________):

        ______________, ______________ = ______________, ________________

        beg = __________________________

        _______________, ________________ = end, _________________

        while (_______________________________):

            end = ___________________________
```

Orders of Growth

One of the biggest parts of computer science is the analysis of how much work functions do when large amounts of data is inputted for processing. Some functions may be slower than others.

Let’s use this function as an example:

```python
def add_one(n):
    return n + 1
```

This function adds 1 to whatever the input is.

As we increase the size of n, we have to check how the number of operations increases (in this case addition).

add_one(1) => one operation

add_one(10) => one operation

add_one(10000000) => one operation

add_one(n) => one operation

This is what we call **constant time** or $\Theta(1)$.

No matter how much we increase the size of n, it will take the same amount of time.

Let’s look at another example:

```python
def n_times(n):
    for i in range(n):
        n+= i
    return n
```

This function adds all the numbers from 1 up to n to the number n.

As we increase the size of n, let’s see how the number of operations increases. (Also in this case, the operation is addition) n_times(1) => 1 operation

n_times(10) => 10 operations

n_times(100000) => 100000 operations

n_times(n) => n operations

As we increase the size of n, the number of times the for loop iterates increases as well, causing the number of times we add to the number n to increase.

This is what we call **linear time** or $\Theta(n)$.

As the input increases, so do the number of operations, at a linear rate.

There are many other different types of orders of growth.

Ranked in order from fastest to slowest are a few of them:

- Constant time: $\Theta(1)$
- Logarithmic time: $\Theta(\log(n))$
- Linear time: $\Theta(n)$
- Lineararithmic time: $\Theta(n\log(n))$
- Polynomial time: $\Theta(n^2)$ or $\Theta(n^3)$ etc.
- Exponential time: $\Theta(2^n)$ or $\Theta(3^n)$ etc.

When identifying orders of growth, it is important to remember that constant additions and multiplications don't matter to the overall function.

Let's look at one last example, editing the last example:

```
def n_times(n):
    n += 1
    for i in range(n):
        n+= i
        n+= 2
    return n
```

Here, you might think the order of growth would be:

$\Theta(2n + 1)$

However, we ignore the extraneous constant addition or multiplication, so the actual order of growth describing this function is just:

$\Theta(n)$

Practice

Identify the orders of growth for each function:

```python
def funn_times(n):
    if (n <= 1):
        return 100000
    else:
        return fun_times(-10000) + fun_times(n-1)
```

```python
def funner_times(n):
    if (n <= 1):
        return 1
    else:
        return funner_times(n//2) + funner_times(n//2)
```

```python
def order_of_fun(n):
    while(n >= 0):
        n = n//2
    return 10
```

```python
def for_fun(n):
    for i in range(n):
        for j in range(n):
            print("I love fun")
```