1 Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a `first` value and the `rest` of the linked list.

To check if a linked list is an empty linked list, compare it against the class attribute `Link.empty`:

```python
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

Implementation

```python
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(repr(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) + '>'
```

A Brief Note on Str and Repr

There are two main ways to produce the "string" of an object in Python: `str()` and `repr()`. While the two are similar, they are used for different purposes. `str()` is used to describe the object to the end user while `repr()` is mainly used for debugging and development.

When we define a class in Python, `str()` and `repr()` are both built-in functions for the class. We can call an object's `str()` and `repr()` by using their respective functions. In addition, the `print()` function calls the `str()` function of the object, while simply calling the object in interactive mode calls the `repr()` function. Here's an example:

```python
class Test:
    def __str__(self):
        return "str"
    def __repr__(self):
        return "repr"

>>> a = Test()
>>> str(a)
'str'
>>> repr(a)
'repr'
>>> print(a)
str
>>> a
repr
```
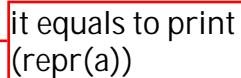
it equals to print (repr(a))

Questions

1.1   Write a function that takes in a a linked list and returns the sum of all its elements. You may assume all elements in `lnk` are integers.

```
def sum_nums(lnk):
    """
    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """
```

```
def sum_nums(lnk):
    if lnk is Link.empty:
        return 0
    return lnk.first + sum_nums(lnk.rest)
a = Link(1, Link(6, Link(7)))
sum_nums(a)
```

14

1.2   Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
```

```
def multiply_lnks(lst_of_lnks):
    end = False
    ans = Link(1)
    curr = ans
    while not end:
        # note that to mutate the elements of the list, we should use lst[i]
        for i in range(len(lst_of_lnks)):
            if lst_of_lnks[i].rest is Link.empty:
                end = True
            curr.first *= lst_of_lnks[i].first
            lst_of_lnks[i] = lst_of_lnks[i].rest
        if not end:
            curr.rest = Link(1)
            curr = curr.rest
    return ans
a = Link(2, Link(3, Link(5)))
b = Link(6, Link(4, Link(2)))
c = Link(4, Link(1, Link(0, Link(2))))
p = multiply_lnks([a, b, c])
print(p.first)
print(p.rest.first)
print(p.rest.rest.rest is Link.empty)
```

```
48
12
True
```

1.3   Implement `filter_link`, which takes in a linked list `link` and a function `f` and
returns a generator which yields the values of `link` for which f returns `True`.

Try to implement this both using a while loop and without using any form of
iteration.

```
def filter_link(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> g = filter_link(link, lambda x: x % 2 == 0)
    >>> next(g)
    2
    >>> next(g)
    StopIteration
    >>> list(filter_link(link, lambda x: x % 2 != 0))
    [1, 3]
    """

    while _____:

        if _____:

            _____

        _____
```

```
def filter_link(link, f):
    while link is not Link.empty:
        if f(link.first):
            yield link.first
        link = link.rest
link = Link(1, Link(2, Link(3)))
g = filter_link(link, lambda x: x % 2 == 0)
print(next(g))
#print(next(g))
print(list(filter_link(link, lambda x: x % 2 != 0)))
```

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

```
def filter_no_iter(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> list(filter_no_iter(link, lambda x: x % 2 != 0))
    [1, 3]
    """

    if _____:

        return

    elif _____:

        _____

        _____
```

```
def filter_no_iter(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> list(filter_no_iter(link, lambda x: x % 2 != 0))
    [1, 3]
    """
    if link is Link.empty:
        return
    elif f(link.first):
        yield link.first
    yield from filter_no_iter(link.rest, f)
link = Link(1, Link(2, Link(3)))
g = filter_link(link, lambda x: x % 2 == 0)
print(next(g))
#print(next(g))
print(list(filter_link(link, lambda x: x % 2 != 0)))
```

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

2 Trees

Recall the tree abstract data type: a tree is defined as having a label and some branches. Previously, we implemented the tree abstraction using Python lists. Let's look at another implementation using objects instead:

```python
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

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

Notice that with this implementation we can mutate a tree using attribute assignment, which wasn't possible in the previous implementation using lists.

```python
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.label = 5
>>> t.label
5
```

Questions

2.1 Define a function make_even which takes in a tree t whose values are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```python
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.label
    2
    >>> t.branches[0].branches[0].label
    4
    """
```

```python
def make_even(t):
    if t.label % 2 == 1:
        t.label += 1
    for branch in t.branches:
        make_even(branch)
t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
make_even(t)
print(t.label)
print(t.branches[0].branches[0].label)
```

```
2
4
```

2.2   Define a function `square_tree(t)` that squares every value in the non-empty tree t. You can assume that every value is a number.

```
def square_tree(t):
    """Mutates a Tree t by squaring all its elements."""
```

```
def square_tree(t):
    """Mutates a Tree t by squaring all its elements."""
    t.label *= t.label
    for branch in t.branches:
            square_tree(branch)
```

2.3   Define the procedure `find_path` that, given a Tree t and an `entry`, returns a list containing the nodes along the path required to get from the root of t to `entry`. If `entry` is not present in t, return `False`.

Assume that the elements in t are unique. Find the path to an element.

For instance, for the following tree `tree_ex`, `find_path` should return:
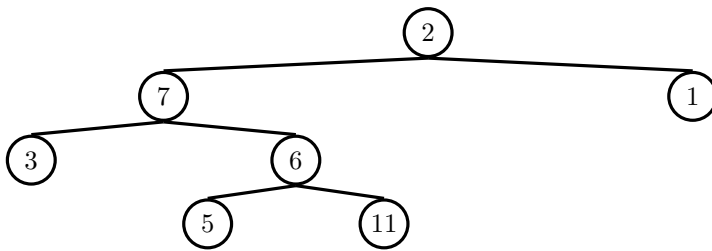

```
>>> tree_ex = Tree(2, [Tree(7, [Tree(3), Tree(6, [Tree(5), Tree(11)])]), Tree(1)])
>>> find_path(tree_ex, 5)
[2, 7, 6, 5]
```

```
def find_path(t, entry):
```

```
def find_path(t, entry):
    if t.label == entry:
        return [t.label]
    if t.branches:
        return [t.label] + sum([find_path(branch, entry) for branch in t.branches if find_path(branch, entry)], [])
tree_ex = Tree(2, [Tree(7, [Tree(3), Tree(6, [Tree(5), Tree(11)])]), Tree(1)])
find_path(tree_ex, 5)
```

```
[2, 7, 6, 5]
```

2.4 Assuming that every value in `t` is a number, define `average(t)`, which returns the average of all the values in `t`. You may not need to use all the provided lines.

```
def average(t):
    """
    Returns the average value of all the nodes in t.
    >>> t0 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
    >>> average(t0)
    1.5
    >>> t1 = Tree(8, [t0, Tree(4)])
    >>> average(t1)
    3.0
    """
    def sum_helper(t):

        total, count = _____


        for _____:


            _____


            _____


            _____


        return total, count


    total, count = _____

    return total / count
```

```
def average(t):
    def sum_helper(t):
        total, count = t.label, 1
        for branch in t.branches:
            subtotal, subcount = sum_helper(branch)
            total += subtotal
            count += subcount
        return total, count
    total, count = sum_helper(t)
    return total / count
t0 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
print(average(t0))
t1 = Tree(8, [t0, Tree(4)])
print(average(t1))
```

1.5
3.0

2.5   Write a function that combines the values of two trees `t1` and `t2` together with the
`combiner` function. Assume that `t1` and `t2` have identical structure. This function
should return a new tree.

Hint: consider using the `zip()` function, which returns an iterator of tuples where
the first items of each iterable object passed in form the first tuple, the second items
are paired together and form the second tuple, and so on and so forth.

```
def combine_tree(t1, t2, combiner):
    """
    >>> a = Tree(1, [Tree(2, [Tree(3)])])
    >>> b = Tree(4, [Tree(5, [Tree(6)])])
    >>> combined = combine_tree(a, b, mul)
    >>> combined.label
    4
    >>> combined.branches[0].label
    10
    """
```

```
def combine_tree(t1, t2, combiner):
    if t1.is_leaf():
        return Tree(combiner(t1.label, t2.label))
    return Tree(combiner(t1.label, t2.label), [combine_tree(branch1, branch2, combiner)
                                    for branch1, branch2 in zip(t1.branches, t2.branches)])
```

2.6   Implement the `alt_tree_map` function that, given a function and a `Tree`, applies the
function to all of the data at every other level of the tree, starting at the root.

```
def alt_tree_map(t, map_fn):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
    >>> negate = lambda x: -x
    >>> alt_tree_map(t, negate)
    Tree(-1, [Tree(2, [Tree(-3)]), Tree(4)])
    """
```

```
def alt_tree_map(t, map_fn):
    t.label = map_fn(t.label)
    for branch in t.branches:
        for b in branch.branches:
            alt_tree_map(b, map_fn)
    return t
t = Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
negate = lambda x: -x
alt_tree_map(t, negate)
```

```
Tree(-1, [Tree(2, [Tree(-3)]), Tree(4)])
```