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

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) + '>'
Questions

1.1 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
    """

Recursive solution:

    product = 1
    for lnk in lst_of_lnks:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lnks_rests = [lnk.rest for lnk in lst_of_lnks]
    return Link(product, multiply_lnks(lst_of_lnks_rests))

For our base case, if we detect that any of the lists in the list of Links is empty, we can return the empty linked list as we're not going to multiply anything.

Otherwise, we compute the product of all the firsts in our list of Links. Then, the subproblem we use here is the rest of all the linked lists in our list of Links. Remember that the result of calling multiply_lnks will be a linked list! We'll use the product we've built so far as the first item in the returned Link, and then the result of the recursive call as the rest of that Link.

Iterative solution:

    import operator
    from functools import reduce
    def prod(factors):
        return reduce(operator.mul, factors, 1)

    head = Link.empty
tail = head
while Link.empty not in lst_of_lnks:
all_prod = prod([l.first for l in lst_of_lnks])
if head is Link.empty:
    head = Link(all_prod)
    tail = head
else:
    tail.rest = Link(all_prod)
    tail = tail.rest
lst_of_lnks = [l.rest for l in lst_of_lnks]
return head

The iterative solution is a bit more involved than the recursive solution. Instead of building the list “backwards” as in the recursive solution (because of the order that the recursive calls result in, the last item in our list will be finished first), we’ll build the resulting linked list as we go along.

We use head and tail to track the front and end of the new linked list we’re creating. Our stopping condition for the loop is if any of the Links in our list of Links runs out of items.

Finally, there’s some special handling for the first item. We need to update both head and tail in that case. Otherwise, we just append to the end of our list using tail, and update tail.

1.2 Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """

Recursive solution:

    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
        remove_duplicates(lnk)
    else:
        remove_duplicates(lnk.rest)

For a list of one or no items, there are no duplicates to remove.

Now consider two possible cases:

• If there is a duplicate of the first item, we will find that the first and second items in the list will have the same values (that is, lnk.first == lnk.rest.first). We can confidently state this because we were told that the input linked list
is in sorted order, so duplicates are adjacent to each other. We'll remove the second item from the list.

Finally, it's tempting to recurse on the remainder of the list (\(\text{lnk}.\text{rest}\)), but remember that there could still be more duplicates of the first item in the rest of the list! So we have to recurse on \(\text{lnk}\) instead. Remember that we have removed an item from the list, so the list is one element smaller than before. Normally, recursing on the same list wouldn't be a valid subproblem.

- Otherwise, there is no duplicate of the first item. We can safely recurse on the remainder of the list.

Iterative solution:

```python
while \(\text{lnk}\) is not \(\text{Link}.\text{empty}\) and \(\text{lnk}.\text{rest}\) is not \(\text{Link}.\text{empty}\):
    if \(\text{lnk}.\text{first}\) == \(\text{lnk}.\text{rest}.\text{first}\):
        \(\text{lnk}.\text{rest} = \text{lnk}.\text{rest}.\text{rest}\)
    else:
        \(\text{lnk} = \text{lnk}.\text{rest}\)
```

The loop condition guarantees that we have at least one item left to consider with \(\text{lnk}\).

For each item in the linked list, we pause and remove all adjacent items that have the same value. Once we see that \(\text{lnk}.\text{first} \neq \text{lnk}.\text{rest}.\text{first}\), we can safely advance to the next item. Once again, this takes advantage of the property that our input linked list is sorted.
2 Interfaces

In computer science, an **interface** is a shared set of attributes, along with a specification of the attributes’ behavior. For example, an interface for vehicles might consist of the following methods:

- **def drive(self)**: Drives the vehicle if it has stopped.
- **def stop(self)**: Stops the vehicle if it is driving.

Data types can implement the same interface in different ways. For example, a **Car** class and a **Train** can both implement the interface described above, but the **Car** probably has a different mechanism for drive than the **Train**.

The power of interfaces is that other programs don’t have to know how each data type implements the interface – only that they have implemented the interface. The following **travel** function can work with both **Cars** and **Trains**:

```python
def travel(vehicle):
    while not at_destination():
        vehicle.drive()
        vehicle.stop()
```

**Magic Methods**

Python defines many interfaces that can be implemented by user-defined classes. For example, the interface for arithmetic consists of the following methods:

- **def __add__(self, other)**: Allows objects to do `self + other`.
- **def __sub__(self, other)**: Allows objects to do `self - other`.
- **def __mul__(self, other)**: Allows objects to do `self * other`.

In addition, there is also an interface for sequences:

- **def __len__(self)**: Allows objects to do `len(self)`.
- **def __getitem__(self, index)**: Allows objects to do `self[i]`.
Questions

2.1 What would Python display?

```python
class A():
    def __init__(self, x):
        self.x = x
    def __repr__(self):
        return self.x
    def __str__(self):
        return self.x * 2

class B():
    def __init__(self):
        print("boo!")
        self.a = []
    def add_a(self, a):
        self.a.append(a)
    def __repr__(self):
        print(len(self.a))
        ret = ""
        for a in self.a:
            ret += str(a)
        return ret

>>> A("one")
one
>>> print(A("one"))
oneone
>>> repr(A("two"))
'two'
>>> b = B()
boo!
>>> b.add_a(A("a"))
>>> b.add_a(A("b"))
>>> b
2
aabb
```

```python
>>> c = A("c")
>>> b.add_a(c)
>>> str(b)
```
3
\texttt{\textasciitilde aabbcc}
2.2 Write the function `is_palindrome` such that it works for any data type that implements the sequence interface.

Assume that the Link class has implemented the `__len__` method and a `__getitem__` method which takes in integers.

```python
def is_palindrome(seq):
    """ Returns True if the sequence is a palindrome. A palindrome is a sequence that reads the same forwards as backwards
    >>> is_palindrome(Link("l", Link("i", Link("n", Link("k")))))
    False
    >>> is_palindrome(["l", "i", "n", "k"])
    False
    >>> is_palindrome("link")
    False
    >>> is_palindrome(Link.empty)
    True
    >>> is_palindrome([])
    True
    >>> is_palindrome(""")
    True
    >>> is_palindrome(Link("a", Link("v", Link("a"))))
    True
    >>> is_palindrome(["a", "v", "a"])
    True
    >>> is_palindrome("ava")
    True
    """

    for i in range(len(seq)//2):
        if seq[i] != seq[len(seq) - 1 - i]:
            return False
    return True
```

3 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

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

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

3.1 Assuming that every value in t is a number, let's define `average(t)`, which returns the average of all the values in t.

```python
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 = t.label, 1
    for b in t.branches:
        b_total, b_count = sum_helper(b)
        total += b_total
        count += b_count
    return total, count

total, count = sum_helper(t)
return total / count
```
3.2 Implement `long_paths`, which returns a list of all `paths` in a tree with length at least `n`. A path in a tree is a linked list of node values that starts with the root and ends at a leaf. Each subsequent element must be from a child of the previous value’s node. The `length` of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are listed in order from left to right. See the doctests for some examples.

```python
def long_paths(tree, n):
    """Return a list of all paths in tree with length at least n.

    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])]), Tree(9)])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    <0 1 2>
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 6 9>
    <0 11 12 13 14>
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 11 12 13 14>
    >>> long_paths(whole, 4)
    [Link(0, Link(11, Link(12, Link(13, Link(14))))))]
    ""
```

```python
paths = []
if n <= 0 and tree.is_leaf():
    paths.append(Link(tree.label))
for b in tree.branches:
    for path in long_paths(b, n - 1):
        paths.append(Link(tree.label, path))
return paths
```