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 __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)
    def __repr__(self):
        if self.rest is Link.empty:
            return 'Link({})'.format(self.first)
        else:
            return 'Link({}, {})'.format(self.first, repr(self.rest))
Questions

1.1 Write a recursive function \texttt{flip\_two} that takes as input a linked list \texttt{lnk} and mutates \texttt{lnk} so that every pair is flipped.

\begin{verbatim}
def flip_two(lnk):
    
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))

Recursive solution:

    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
    flip_two(lnk.rest.rest)

If there's only a single item (or no item) to flip, then we're done.

Otherwise, we swap the contents of the first and second items in the list. Since we've handled the first two items, we then need to recurse on

Although the question explicitly asks for a recursive solution, there is also a fairly similar iterative solution:

    while lnk is not Link.empty and lnk.rest is not Link.empty:
        lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
        lnk = lnk.rest.rest

We will advance \texttt{lnk} until we see there are no more items or there is only one more Link object to process. Processing each Link involves swapping the contents of the first and second items in the list (same as the recursive solution).

Notice that the code is remarkably similar to the recursive implementation of \texttt{flip\_two}.

Video walkthrough

1.2 Write a function \texttt{remove\_duplicates} that takes as input a sorted linked list of integers, \texttt{lnk}, and mutates \texttt{lnk} so that all duplicates are removed.

\begin{verbatim}
def remove_duplicates(lnk):
    
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> unique = remove_duplicates(lnk)
    >>> len(unique)
    2
\end{verbatim}
Recursive solution:

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

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 (`lnk.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 `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
cur = lnk
while cur is not Link.empty and cur.rest is not Link.empty:
    if cur.first == cur.rest.first:
        cur.rest = cur.rest.rest
    else:
        cur = cur.rest

return lnk
```

The loop condition guarantees that we have at least one item left to consider with `cur`.

For each item in the linked list, we pause and remove all adjacent items that have the same value. Once we see that `cur.first != cur.rest.first`, we can safely advance to the next item. Once again, this takes advantage of the property that our input linked list is sorted.
1.3 Define `reverse`, which takes in a linked list and reverses the order of the links. The function may *not* return a new list; it must mutate the original list. Return a pointer to the head of the reversed list.

```python
def reverse(lnk):
    """
    >>> a = Link(1, Link(2, Link(3)))
    >>> r = reverse(a)
    >>> r.first
    3
    >>> r.rest.first
    2
    """

Recursive solution:

```python
    if lnk is Link.empty or lnk.rest is Link.empty:
        return lnk
    rest_rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = Link.empty
    return rest_rev
```

For the base case, a linked list with no items or a single item is trivial to reverse.

Let’s formally name our variables to make the explanation of the following process a bit easier. The original list is `lnk`, we reverse `lnk.rest` recursively and get back a pointer to the head of the reversed version of `lnk.rest`, which is `rest_rev`.

Notice that `lnk.rest` is the last item of the list referred to by `rest_rev`. All we have to do is attach the first item of `lnk` to the end of the reversed rest, and then make sure that `lnk.rest` is the empty list as it is now the last item in the reversed list.

Video walkthrough

Iterative solution (1):

```python
    if lnk is Link.empty:
        return lnk
    cur = lnk
    nxt = lnk.rest
    cur.rest = Link.empty

    while nxt is not Link.empty:
        after = nxt.rest
        nxt.rest = cur
        cur = nxt
        nxt = after
    return cur
```

The iterative solution is quite different from the recursive solution. We go through every item in our linked list, and reattach them in reverse order. That is, we attach
the second item back to the first item, the third item back to the second item, and so on.

The tricky part is figuring what information we need to keep track of in order to do this. We use a two pointer method that tracks a current and a following position in a linked list. The logic is not too complicated, but the best way to understand it is to work through an example with a box and pointer diagram.

**Iterative solution (2):**

```python
new_lnk = Link.empty
while lnk is not Link.empty:
    new_lnk, lnk.rest, lnk = lnk, new_lnk, lnk.rest
return new_lnk
```

Here’s yet another iterative approach, different from both the previous iterative and recursive approaches.

We begin by asking how we’d gather the values in a linked list into a list (not a linked list, but a regular Python list) in reverse order.

```python
>>> xs = Link(1, Link(2, Link(3)))
>>> xs
Link(1, Link(2, Link(3)))
>>> reverse(xs)
[3, 2, 1]
```

We could do this by iterating through the linked list and inserting the values into the front of the list:

```python
def reverse(lnk):
    new_list = []
    while lnk is not Link.empty:
        new_list.insert(0, lnk.first) # Insert the link value before all existing elements of new_list
        lnk = lnk.rest
    return new_list
```

This works because if value A comes before value B in the original list, then B will be inserted before value A in our new list.

We could keep this same approach, but have `new_lnk` be a linked list instead of an ordinary list.

```python
def reverse(lnk):
    new_lnk = Link.empty
    while lnk is not Link.empty:
        new_lnk = Link(lnk.first, new_lnk) # Create a new link and insert it before all existing links in new_lnk
        lnk = lnk.rest
    return new_lnk
```

This works, but we are not allowed to create new `Link` instances.
So, instead of copying \texttt{lnk} by constructing a new \texttt{Link} with the same first attribute as \texttt{lnk}, we should just make \texttt{lnk} the new head of \texttt{new\_lnk}. But that means we simultaneously need to make three updates:

- \texttt{new\_lnk} should point at what \texttt{lnk} was pointing at (because \texttt{lnk} is the new head of \texttt{new\_lnk}).
- \texttt{lnk\_rest} should point at what \texttt{new\_lnk} was pointing at (this makes \texttt{lnk} the new head of \texttt{new\_lnk}).
- \texttt{lnk} should point at what \texttt{lnk\_rest} was pointing at (because we still need to iterate through the original list).

This leads us the final solution presented earlier.

1.4 Write \texttt{multiply\_lnks}, which takes in a Python list of \texttt{Link} objects and multiplies them element-wise. It should return a new linked list. If not all of the \texttt{Link} objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the \texttt{Link} objects are shallow linked lists, and that \texttt{lst\_of\_lnks} contains at least one linked list.

\begin{verbatim}
def multiply\_lnks(lst\_of\_lnks):
    
    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))
\end{verbatim}

For our base case, if we detect that any of the lists in the list of \texttt{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 \texttt{Links}. Then, the subproblem we use here is the rest of all the linked lists in our list of \texttt{Links}. Remember that the result of calling \texttt{multiply\_lnks} will be a linked list! We'll use the product we've built so far as the first item in the returned \texttt{Link}, and then the result of the recursive call as the rest of that \texttt{Link}. 

Iterative solution:

```python
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`. 
2 Midterm Review

2.1 Define a function `even_weighted` that takes in a list `lst` and returns a new list that keeps only the even-indexed elements of `lst` and multiplies each of those elements by the corresponding index.

```python
def even_weighted(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """
    return [i * lst[i] for i in range(len(lst)) if i % 2 == 0]
```

Alternatively, we can take advantage of the step size for `range` to make sure we only consider even numbered indices:

```python
def even_weighted(lst):
    return [i * lst[i] for i in range(0, len(lst), 2)]
```

The key point to note is that instead of iterating over each element in the list, we must instead iterate over the indices of the list. Otherwise, there’s no way to tell if we should keep a given element.

One way of solving these problems is to try and write your solution as a for loop first, and then transform it into a list comprehension. The for loop solution might look something like this:

```python
result = []
for i in range(len(lst)):
    if i % 2 == 0:
        result.append(i * lst[i])

return result
```

Video walkthrough

2.2 The `quicksort` sorting algorithm is an efficient and commonly used algorithm to order the elements of a list. We choose one element of the list to be the pivot element and partition the remaining elements into two lists: one of elements less than the pivot and one of elements greater than the pivot. We recursively sort the two lists, which gives us a sorted list of all the elements less than the pivot and all the elements greater than the pivot, which we can then combine with the pivot for a completely sorted list.

First, implement the `quicksort_list` function. Choose the first element of the list as the pivot. You may assume that all elements are distinct.

```python
def quicksort_list(lst):
    """
```
```python
>>> quicksort_list([3, 1, 4])
[1, 3, 4]

```
We can also use quicksort to sort linked lists! Implement the `quicksort_link` function, without constructing additional `Link` instances.

You can assume that the `extend_links` function is already defined. It takes two linked lists and mutates the first so that the ending node points to the second. `extend_link` returns the head of the first linked list.

```python
>>> l1, l2 = Link(1, Link(2)), Link(3, Link(4))
>>> l3 = extend_links(l1, l2)
>>> l3
Link(1, Link(2, Link(3, Link(4))))
>>> l1 is l3
True
```

```python
def quicksort_link(link):
    ""
    >>> s = Link(3, Link(1, Link(4)))
    >>> quicksort_link(s)
    Link(1, Link(3, Link(4)))
    ""
    if ____________________________________________________:
        return link

    pivot, _______ = ________________________________________

    less, greater = __________________________________________

    while link is not Link.empty:
        curr, rest = link, link.rest

        if ____________________________________________________:
            ________________________________________________

        else:
            ________________________________________________

        link = ______________________________________________

    less = ________________________________________________

    greater = _____________________________________________

    _______________________________________________________
    return ________________________________________________
```
```python
def quicksort_link(link):
    if link is Link.empty or link.rest is Link.empty:
        return link
    pivot, link = link, link.rest
    less, greater = Link.empty, Link.empty
    while link is not Link.empty:
        curr, rest = link, link.rest
        if curr.first < pivot.first:
            less, curr.rest = curr, less
        else:
            greater, curr.rest = curr, greater
        link = rest
    less = quicksort_link(less)
    greater = quicksort_link(greater)
    pivot.rest = greater
    return extend_links(less, pivot)
```

While the solution may look very different, doing quicksort on a linked list is very similar to doing quicksort on a Python list. The main difference is that we cannot use list comprehensions to easily split off elements of our list into less and greater sublists. At the end, we have to use `extend_links` to concatenate the linked lists.
2.4 Implement the functions `max_product`, which takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```python
def max_product(lst):
    """Return the maximum product that can be formed using lst without using any consecutive numbers
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
    if lst == []:
        return 1
    elif len(lst) == 1: # Base case optional
        return lst[0]
    else:
        return max(max_product(lst[1:]), lst[0]*max_product(lst[2:]))
```

At each step, we choose if we want to include the current number in our product or not:

- If we include the current number, we cannot use the adjacent number.
- If we don’t use the current number, we try the adjacent number (and obviously ignore the current number).

The recursive calls represent these two alternate realities. Finally, we pick the one that gives us the largest product.

**Video walkthrough**

2.5 An **expression tree** is a tree that contains a function for each non-leaf node, which can be either `+` or `*`. All leaves are numbers. Implement `eval_tree`, which evaluates an expression tree to its value. You may want to use the functions `sum` and `prod`, which take a list of numbers and compute the sum and product respectively.

```python
def eval_tree(tree):
    """Evaluates an expression tree with functions the root.
    >>> eval_tree(tree(1))
    1
    >>> expr = tree('*', [tree(2), tree(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(tree('+', [expr, tree(4), tree(5)]))
    15
    """
```
if is_leaf(tree):
    return label(tree)
args = [eval_tree(branch) for branch in branches(tree)]
if label(tree) == '+':
    return sum(args)
else: # label(tree) == '*'
    return prod(args)

Leaf values are guaranteed to be a number, so we can just return their label.

Otherwise, we have to evaluate each of the branches and then combine their result using whichever operator we have in our current root.

If you want to try this out yourself, note that `prod` isn't actually a built-in operator in Python. You can write it yourself using something like the following:

```python
def prod(iterable):
    from functools import reduce
    from operator import mul
    return reduce(mul, iterable, 1)
```

Video walkthrough
Implement *widest_level*, which takes a *Tree* instance and returns the elements at the depth with the most elements.

In this problem, you may find it helpful to use the second optional argument to *sum*, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of *sum* starting with a list, which allows you to concatenate items in a list.

```python
def widest_level(t):
    ""
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
               ... Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    ""
    levels = []
    x = [t]

    while _____________________________________________:
        _________________________________ = sum(_______________________________, [])

        return max(levels, key=_______________________________)
```

**Main idea:** we’ll traverse each level of the tree and keep track of the elements of the levels. After we’re done, we return the level with the most items.

Here, *x* keeps track of the trees in the current level. To get the next level of trees, we take all the branches from all the trees in the current level. The special *sum* call is needed to make sure we get a list of trees, instead of a list of branches (since branches are a list of trees themselves).

Finally, we use *max* with a key to select the list with the longest length from our list of levels.

Complete *redundant_map*, which takes a tree *t* and a function *f*, and applies *f* to the node ($2^d$) times, where *d* is the depth of the node. The root has a depth of 0.
def redundant_map(t, f):
    ""
    >>> double = lambda x: x*2
    >>> tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
    >>> print_levels(redundant_map(tree, double))
    [2] # 1 * 2 ^ (1); Apply double one time
    [4, 8] # 1 * 2 ^ (2), 2 * 2 ^ (2); Apply double two times
    [16] # 1 * 2 ^ (2 * 2); Apply double four times
    [256] # 1 * 2 ^ (2 * 3); Apply double eight times
    ""
    t.label = _______________________________
    
    new_f = ______________________________________________
    
    t.branches = ___________________________________________
    
    return t

def redundant_map(t, f):
    ""
    >>> double = lambda x: x*2
    >>> tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
    >>> print_levels(redundant_map(tree, double))
    [2]
    [4, 8]
    [16]
    [256]
    ""
    t.label = f(t.label)
    new_f = lambda x: f(f(x))
    t.branches = [redundant_map(branch, new_f) for branch in t.branches]
    return t

Every time we recurse, we transform our map function into one that is one level deeper in terms of calls to input function f. To see why this will achieve the result we want, let’s look at what happens to some input function f.

- The first call to redundant_map will call f once.
- This means on the second call to redundant_map, we pass in a function g that causes the original f to be called two times.
- On the third call to redundant_map, we pass in a function h that causes g to be called two times. Remember that g calls original f twice, so h will end up calling original f four times.

Therefore, each level will have double the calls to f as the previous level, which matches the requirements.