QUICK CONTENT REVIEW

• Linked Lists

- one implementation of sequences in python (it is a recursive structure)
  - ↳ we have defined a class Link that we use
    - ↳ each linked list is either empty (Link.empty) or a Link object containing a first value and the rest of the linked list

Link object:

first (this is a value)

rest (this is a pointer to the next link object in the sequence or null if no more objects)

ex.) If I have a sequence of #s: {1, 2, 3, 4}
I can represent it using a linked list:

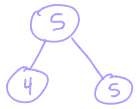
this is how we represent null

To get the first element of L
↳ L.first
To get "remaining" list (2 → 3 → 4)
↳ L.rest
To get second element of L
↳ (L.rest).first → first element in the rest of the list

• Trees (new implementation using objects)

- same as trees from before (conceptually)
- only difference now is that since each tree is an object we can directly mutate a tree (by assigning different values to attributes)

To create a new tree:
t = Tree(5, [Tree(4), Tree(5)]) →



To get the label of the root of a tree:
t.label → 5 (label of root of tree)

To get the list of branches:
t.branches → [Tree(4), Tree(5)]

WARM UP QUESTIONS

Q3: Inheritance Review

Q3: (Tutorial) Inheritance Review: That's a Constructor, __init__?

Let's say we want to create a class `Monarch` that inherits from another class, `Butterfly`. We've partially written an `__init__` method for `Monarch`. For each of the following options, state whether it would correctly complete the method so that every instance of `Monarch` has all of the instance attributes of a `Butterfly` instance? You may assume that a monarch butterfly has the default value of 2 wings.

```
class Butterfly():
    def __init__(self, wings=2):
        self.wings = wings

class Monarch(Butterfly):
    def __init__(self):
        ____________________________________
        self.colors = ['orange', 'black', 'white']
```

Superclass → Subclass

```
super.__init__()
```

Your Answer:

No

Butterfly.__init__(self)

```
super().__init__()
```

Your Answer:

Yes

b = Butterfly(2)
b.__init__()
class.method(self) = object

```
Butterfly.__init__()
```

Your Answer:

No

```
Butterfly.__init__(self)
```

Your Answer:

Yes

Some butterflies like the Owl Butterfly (https://en.wikipedia.org/wiki/Owl_butterfly) have adaptations that allow them to mimic other animals with their wing patterns. Let's write a class for these `MimicButterflies`. In addition to all of the instance variables of a regular `Butterfly` instance, these should also have an instance variable `mimic_animal` describing the name of the animal they mimic. Fill in the blanks in the lines below to create this class.

```
class MimicButterfly(_____________):
    def __init__(self, mimic_animal):
        _____________.init()
        _____________ = mimic_animal
```

come back to only if time!

What expression completes the first blank? **Your Answer:**

Butterfly ↑ want to inherit everything that regular butterflies have

What expression completes the second blank? **Your Answer:**

super() ← want to call Butterfly's init method (notice it passes no parameters so we cannot put Butterfly here)

What expression completes the third blank? **Your Answer:**

self.mimic_animal

↑ set's this instance's attribute!

Q4: LL warmup

Q4: (Tutorial) Warmup: The Hy-rules of Linked Lists

In this question, we are given the following Linked List:

```
ganondorf = Link('zelda', Link('young link', Link('sheik', Link.empty)))
```

What expression would give us the value 'sheik' from this Linked List?

Your Answer:

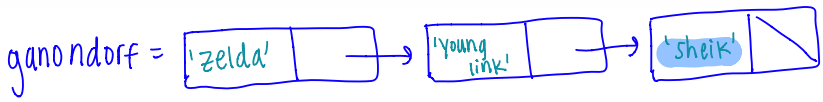
ganondorf.rest.rest.first

① = ②

What is the value of `ganondorf.rest.first` ?

Your Answer:

young link



PRACTICE PROBLEMS

Q6: Flip Two (option 2: harder)

input

Tutorial: Write a recursive function `flip_two` that takes as input a linked list `s` and mutates `s` so that every pair is flipped.

→ what we want to do to our input inside the function

```
def flip_two( s ):
    """
    >>> 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)))))
    """
```

lnk.first → 1
lnk.rest → Link(2, Link(3, Link(4, Link(5))))

Individual work time: 5 min

collaborative solution: 5 min

Collabedit!

SOLUTION:

```
def flip_two(s):
    if s is Link.empty:
        return
    if s.rest is Link.empty:
        return
    s.first, s.rest.first = s.rest.first, s.first
    flip_two(s.rest.rest)
```

always compare your CL to Link.empty to check if it is null

→ if the linked list is empty, nothing to switch

→ if the linked list only has one element, cannot switch with anything

→ good way to swap values w/out having to store intermeadiate values

→ already swapped first two elements so make recursive call on remaining LL starting with 3rd element

we know this isn't null cuz of our base cases

Q5: Multiply lnks (option 1: medium difficulty)

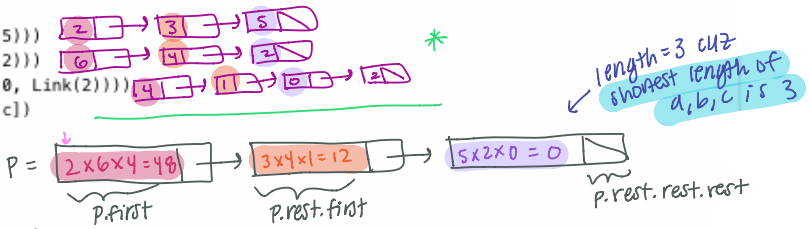
Q5: (Tutorial) Multiply Lnks

input

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.

```
1   def multiply_lnks(lst_of_lnks):
2       """
3       >>> a = Link(2, Link(3, Link(5)))
4       >>> b = Link(6, Link(4, Link(2)))
5       >>> c = Link(4, Link(1, Link(0, Link(2))))
6       >>> p = multiply_lnks([a, b, c])
7       >>> p.first
8       48
9       >>> p.rest.first
10      12
11      >>> p.rest.rest.rest is Link.empty
12      True
13      """
14      # Implementation Note: you might not need all lines in this skeleton code
15      mul_links = 1
16      for item in lst_of_lnks:
17          if item is Link.empty:
18              return Link.empty
19          mul_links *= item.first
20      new_lst = [lnk.rest for lnk in lst_of_lnks]
21      return Link(mul_links, multiply_lnks(new_lst))
```



Link(___, recursive call)

Individual: 5 min.
walk through: 7 min.

Hint: LL is recursive data structure!

4:33

Solution:

Link(value, LL)

```
def multiply_lnks(lst_of_lnks):
    # Implementation Note: you might not need all lines in this skeleton code
    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))
```

get the correct value for 'first' element in new LL I want to return

for every LL in my list, I will take care of appropriately updating the 'first' value and then will rely on recursion for 'rest'

← base case

this is the 'rest' of the lists that will be passed in for my recursive call

← return a new LL

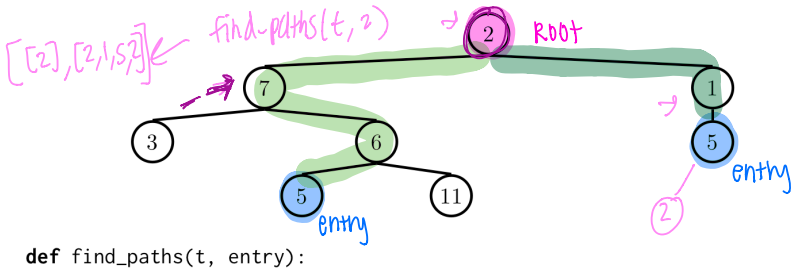
int Recursive LL

Q8: Find Paths

HINT: similar to find_paths on Disc. 05 (2/24/21)

3.3 **Tutorial:** Define the procedure `find_paths` that, given a Tree `t` and an `entry`, returns a list of lists containing the nodes along each path from the root of `t` to `entry`. You may return the paths in any order.

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



Two paths from root to entry:
[[2,7,6,5],
[2,1,5]]

```
def find_paths(t, entry):
    >>> tree_ex = Tree(2, [Tree(7, [Tree(3), Tree(6, [Tree(5), Tree(11)])]), Tree(1, [Tree(5)])])
    >>> find_paths(tree_ex, 5)
    [[2, 7, 6, 5], [2, 1, 5]]
    >>> find_paths(tree_ex, 12)
    []

    paths = []
    if t.label == entry:
        paths.append([t.label])
    for b in t.branches:
        for path in find_paths(b, entry):
            paths.append([t.label] + path)
    return paths
```

label(t) branches(t)

nested list

[0 [...], [...]]

Individual work time: 5 min
collaborative solution: 7 min

4:50

[2, [3,5], [7,6,5]]
=
[[2,3,5], [2,7,6,5]]

SOLUTION:

```
def find_paths(t, entry):
    paths = []
    if t.label == entry:
        paths.append([t.label])
    for b in t.branches:
        for path in find_paths(b, entry):
            paths.append([t.label] + path)
    return paths
```

simplest case is if root of tree is the entry we're trying to get to, in that case what's the path from root to entry? just a list of t.label right?

find the path from the root of each subtree (branch) to entry

just add on the label of the root to this path from subtree to entry & add that as valid list into paths

we know we need to return a list of lists so if paths is our list of lists we just return that (modify/add to it in body of function)