Name:_____ **Partner:** _____

Python Activity 43: Linked Lists – Special Methods

Let's build more of our own data types, using a recursive class!

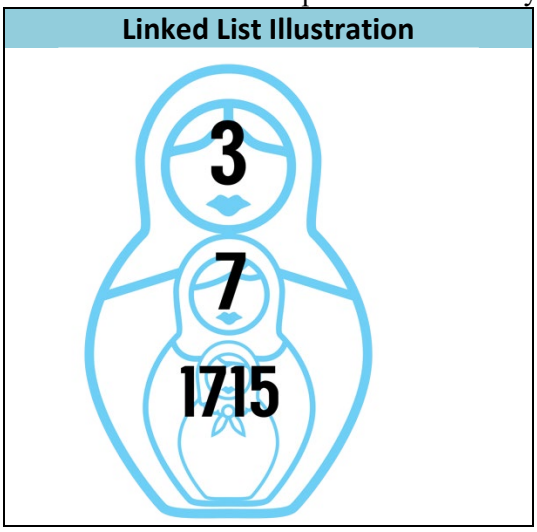
Learning Objectives
Students will be able to:
Content:
- Define a **linked list**
- Identify the **value** and **rest** of a linked list

Process:
- Write code that modifies a recursive list class
- Write code that iterates over a recursive list's values.

Prior Knowledge
- Python concepts: Recursion, User-defined types, Special methods

Concept Model:
We've encountered Python lists before, but now we're going to implement our own lists using a well-known data structure design called ***Linked Lists***.

CM1. This illustration represents the underlying class structure for the list, `ll = [3, 7, 1715]`.



a. How many elements are in `ll`?
_____
How many nesting dolls are shown on the left?
_____

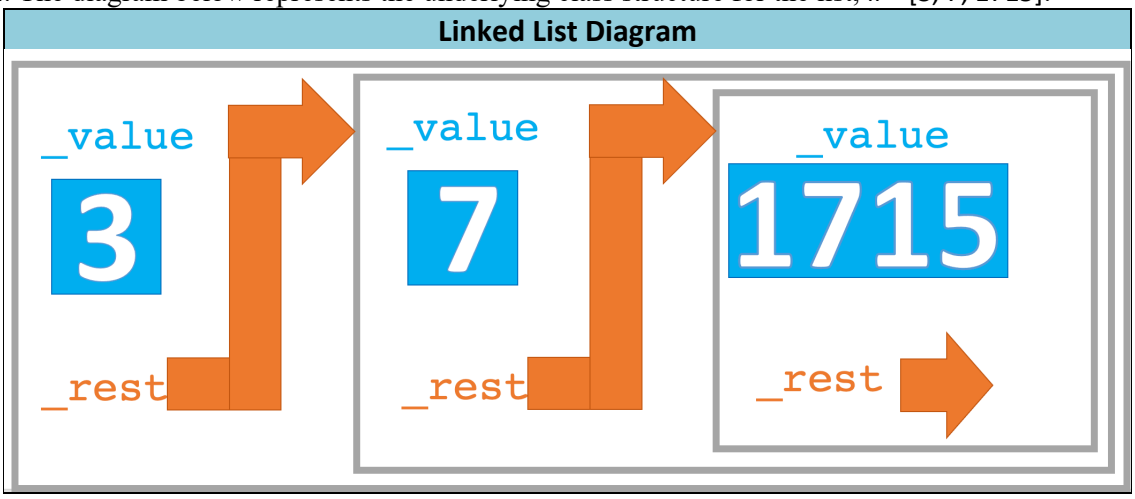
b. What is the value of the 0^{th} element of `ll`?
_____
What is the value of the 0^{th} nesting doll?
_____

c. How do we know which nesting doll comes after the 0^{th}-index doll?

d. What might the 3th-index doll contain? _____

e. Draw a 3th-index doll to the above illustration, with the value `None`.

CM2. The diagram below represents the underlying class structure for the list, ll = [3, 7, 1715]:



a. What are the two attributes of the `LinkedList` class? _____

b. What is stored in the `_value` attribute of the 0th LinkedList of this list? _____

c. What is stored in the `_rest` attribute of the 0th LinkedList of this list?

_____

d. Draw on the diagram with what you think is stored in the `_rest` attribute of the *last* LinkedList of this list.

e. What does the `_rest` attribute represent?

_____

FYI: Any instance of a class that is created by using another instance of the class is a ***recursive class***.

Critical Thinking Questions:

1. The following code creates a LinkedList version of our list:

```
ll1     = LinkedList(3, LinkedList(7, LinkedList(1715)))
```

a. What does the *first* parameter of a new `LinkedList` instance represent?

_____

b. What does the *second* parameter of a new `LinkedList` instance represent?

_____

c. How might we write a line of code to make a new list, ll2, which is the same as ll1 but has the string "today" as the value of the first element?

_____

2. Examine the following example `__init__` method from the `LinkedList` class:

```
class LinkedList:

    def __init__(self, value=None, rest=None):
        self._value = value
        self._rest = rest
```

a. What *type* of object might `_value` be? _____

b. What *type* of object must `_rest` be? _____

c. Write a line of code for the body of the `get_value(self)` method:

_____

3. Examine the following example method from the `LinkedList` class:

```
def mystery(self):
    if self._rest is None:
        return str(self._value)
    else:
        return str(self._value) + ", " + self._rest.mystery()
```

a. What does the following line do?: `if self._rest is None:`

_____

b. How do we know this method is recursive?

_____

c. For this recursive method, what is the base case / stopping condition?

_____

d. For this recursive method, how is the longer journey broken down/shortened?

_____

e. What is the small step we take in `mystery` for each recursive call?

_____

f. For our example list, `ll1`, what will this `mystery` method return?

_____

g. What should the `mystery` method be renamed to?

_____

h. Rewrite the *last* line of our example code to *implicitly* call this renamed method:

_____

4. We want to write a recursive `__len__` method for our `LinkedList` class that will have the following behavior:

```
>>> ll1 = LinkedList(3, LinkedList(7, LinkedList(1715)))
>>> ll1.__len__()
3
```

a. How might we call `__len__` *implicitly* on a `LinkedList` object?

_____

b. For this recursive method, what is the base case / stopping condition?
(*Hint: There might be more than 1!*)

_____

c. For this recursive method, how is the longer journey broken down/shortened?

_____

d. What is the small step we must take in each recursive call?

_____

e. Below is the implementation of the `__len__` method. Place a *star* next to the base cases. *Circle* where we make the journey smaller. *Underline* where we take our repeated small step.

```
def __len__(self):
    if self._rest is None and self._value is None:
        return 0
    elif self._rest is None and self._value is not None:
        return 1
    else:
        return 1 + len(self._rest)
```

FYI: It is preferred to use ***is*** or ***is not*** operators (as opposted to ***==*** or ***!=***) when comparing a user-defined objects to a ***None*** value.

f. Why might we need two base cases for this method?

_____

5. Match up special methods on the left-hand column with the code that implicitly calls them in the right-hand column (make educated guesses using special method names and parameters!):

Special Method	Called By
a. `__len__(self)`	`lLst = LinkedList()`
b. `__init__(self)`	`len(lLst)`
c. `__str__(self)`	`lLst[1]`
d. `__contains__(self, item)`	`lLst == lLst2`
e. `__add__(self, other)`	`lLst [0] = "founded"`
f. `__getitem__(self, item)`	`lLst + lLst2`
g. `__setitem__(self, item, val)`	`1715 in lLst`
h. `__eq__(self, other)`	`str(lLst)`

(*There's many more special methods, we've seen others before!*)

Confirm your responses by checking the python3 documentation:

https://docs.python.org/3/reference/datamodel.html#special-method-names

6. Examine the following example code:

```
def __contains__(self, val):
    if self._value == val:
        return True
    elif self._rest is None:
        return False
    else:
        return val in self._rest
```

a. For this recursive method, what is the base case / stopping condition?

_____

b. For this recursive method, how is the longer journey broken down/shortened?

_____

c. What is the small step we take in `__contains__` for each recursive call?

_____

d. Circle the *recursive call* in this method.

7. We want to write a `__getitem__(self, index)` method for our `LinkedList` class that will returns the value at the *index*, `index`, of our `LinkedList`:

a. For this recursive method, what is the base case / stopping condition?

_____

b. For this recursive method, how is the longer journey broken down/shortened?

_____

c. What is the small step we must take in each recursive call?

_____

d. Below is the partially completed implementation of the `insert` method. Fill in the lines below the `(i)`, and `(ii)` comments with Python code.

```
def __getitem__(self, index):
    # (i) if index is 0, we found the item
    if



    # (ii) else recurse until index reaches 0
    else:
```

Application Questions: Use the Python Interpreter to check your work

1. Write a recursive `LinkedList` method that changes the value located at index, `ind`, to `val`.

```
def __setitem__(self, ind, val):
```

_____

_____

_____

_____

_____

_____

2. Write the `__add__(self, other)` method for our `LinkedList` class so that we can concatenate two `LinkedLists` together. When considering the recursion, determine (1) what is the stopping condition, (2) what is the small step we should take with each recursive call, and (3) how do we break the journey down into a smaller journey::

```
def __add__(self):
```

3. Write the `__eq__(self, other)` method for our `LinkedList` class so that we compare whether two lists are equivalent:

```
def __eq__(self, other):
```