Linked List Summary – Why Linked LIsts

- Throughout your python career we have made many instances of the built-in 'list' class of python.
- In the example below we will *instantiate* an *object* of the *class* 'list', and assign it to the variable 'week'

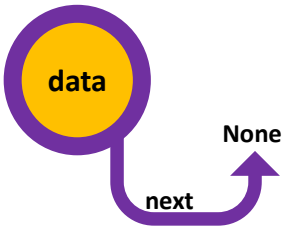
```
>>> week = list()
>>>
>>>
>>> week
[]
```

- Using dot notation we can use the *methods* of the *class* 'list' to add *items* to 'week'

```
>>> week.append("Monday")
>>> week.append("Tuesday")
>>> week.append("Wednesday")
>>> week.append("Thursday")
>>> week
['Monday', 'Tuesday', 'Wednesday', 'Thursday']
>>> week.remove('Thursday')
>>> week
['Monday', 'Tuesday', 'Wednesday']
```

- Please note that 'append' and 'remove' don't work automatically for every object. They only work for lists because lists have that method. Try it with a string – it won't work.
- Lists are pretty useful but they the items of a list can only be accessed based on their position. The items in the list have no relationship to each other. There might be situations, such as our gerrymandering simulation where we need to maintain the relationship amongst items. Using pointers allows for those relationships to be encoded
- A more common situation where the shortfall of the built-in class list comes up is in the context of really big lists. If we want to add something to the middle of a list, we have to 'move' every other item over to make space for the new item. This can be computationally expensive in very large lists. The use of pointers can avoid this, we can just change what the pointers are referring to.

Nodes – the basic unit of a linked list



A node is a user defined class that will act as the foundation of a linked list.
Nodes have two attributes (1) data, which will keep track of the data, and (2) a pointer, 'next' which will keep a reference to another Node object.

```
class Node:
  def __init__(self, initialData):
    self.data = initialData
    self.next = None
```

Observe that there is no input argument for next in the constructor. When you instantiate a Node object, it has no value for next as a default.

We establish a method 'setNext' to have change the attribute next to point to another Node.

We also have methods for changing and updating and returning the data and the pointer.

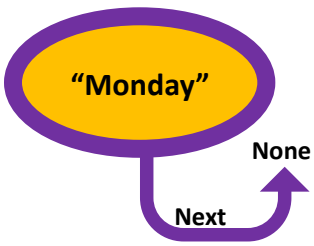
```
class Node:
  def __init__(self, initialData):
    self.data = initialData
    self.next = None

  def setNext(self, newnext):
    self.next = newnext

  def setData(self, newData):
    self.data = newData

  def getData(self):
    return self.data

  def getNext(self):
    return self.next
```

Now this is a fully functional Node class.

We can make some objects of the class Node.

```
node1 = Node(initialData = "Monday")
node2 = Node(initialData = "Tuesday")
```

But these objects start out pointing to None. (remember that's the default).



We can use the setNext method to get node1 to point to node2.

```
print(node1)
print(node2)
```

```
<__main__.Node object at 0x7ea932321f90>
<__main__.Node object at 0x7ea932321660>
```

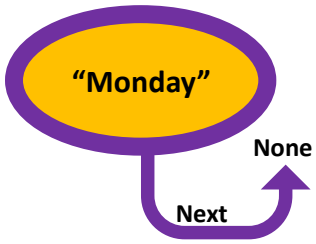
```
print(node1.next)
```

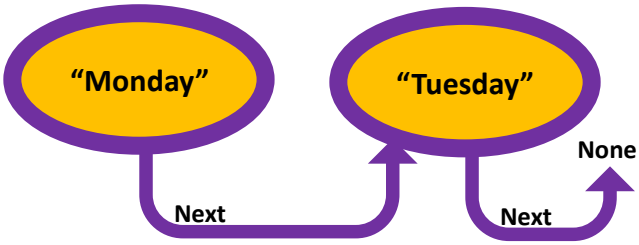
```
<__main__.Node object at 0x7ea932321660>
```

```
node1.setNext(node2)
```

```
print(node1.next)
```

```
<__main__.Node object at 0x7ea932321660>
```





Our nodes are working great, and we can manually string a bunch of nodes together, but its not really a linked list yet. We can't easily iterate over node1 and node2 right now. To do that we have to establish a new class classed LinkedList.

LinkedList

This may seem strange at first, but LinkedList like most advanced data structures are initialized as empty objects

```
class Linkedlist():
  def __init__(self):
    self.head = None

  def append(self, NodeToAdd):
    if self.head == None:
      self.head = NodeToAdd
    else:
      last_node = self.head
      while last_node.next:
        last_node = last_node.next
      last_node.next = NodeToAdd
```

There are no input arguments you can add when you create this. But there are methods you can use to add Nodes to the LinkedList. The first method we need is 'append' which does the same as append in traditional lists – it adds to the end.
Lets look at the append code in more detail.

Append has 1 input argument – NodeToAdd which is a Node that you want to add to your LinkedList.

```
def append(self, NodeToAdd):
  if self.head == None:
    self.head = NodeToAdd
  else:
    last_node = self.head
    while last_node.next:
      last_node = last_node.next
    last_node.next = NodeToAdd
```

In our case lets create a LinkedList called week

```
week = LinkedList()
```

week starts out empty, but we can use append to add node1 and node2 from before to week.

When we add node1, the appending is straight forward. week.head is None so the append method simply assigns node1 to the week.head attribute

```
print(week.head)

None
```

```
week.append(node1)
print(week.head)

<__main__.Node object at 0x7ea932321f90>
```

Now when we add node2 to week it will be a bit more complicated. week.head is no longer None, so instead we have to move to the else block of the conditional

Inside that conditional it assigns self.head to a temporary variable called last_node.

```
  else:
    last_node = self.head
```

Then it checks to see whether last_node.next is == None or if it points to another node. This can be accomplished with another if statement, but to simplify the code people typically use a while True statement.

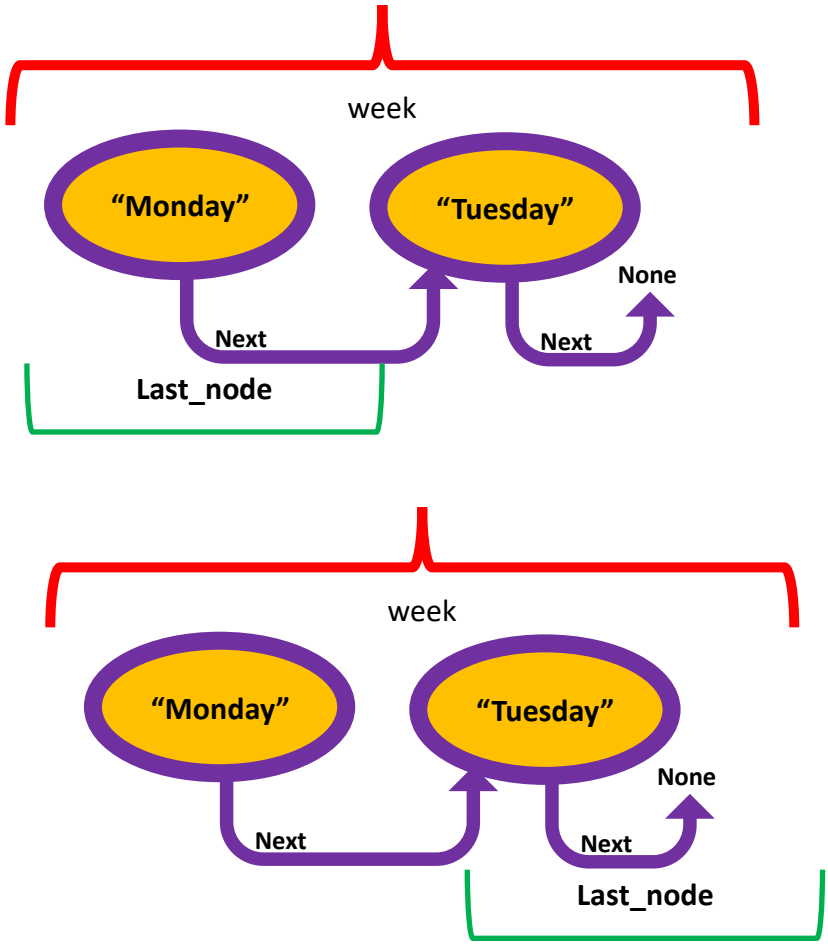
```
while last_node.next:
  last_node = last_node.next
```

if last_node.next == None the while statement is evaluated as False, otherwise it remains True. If its true, we simply update the identify of the variable last_node and continue. When we finally find a last_node.next that is None we can escape the while block and append our node2.

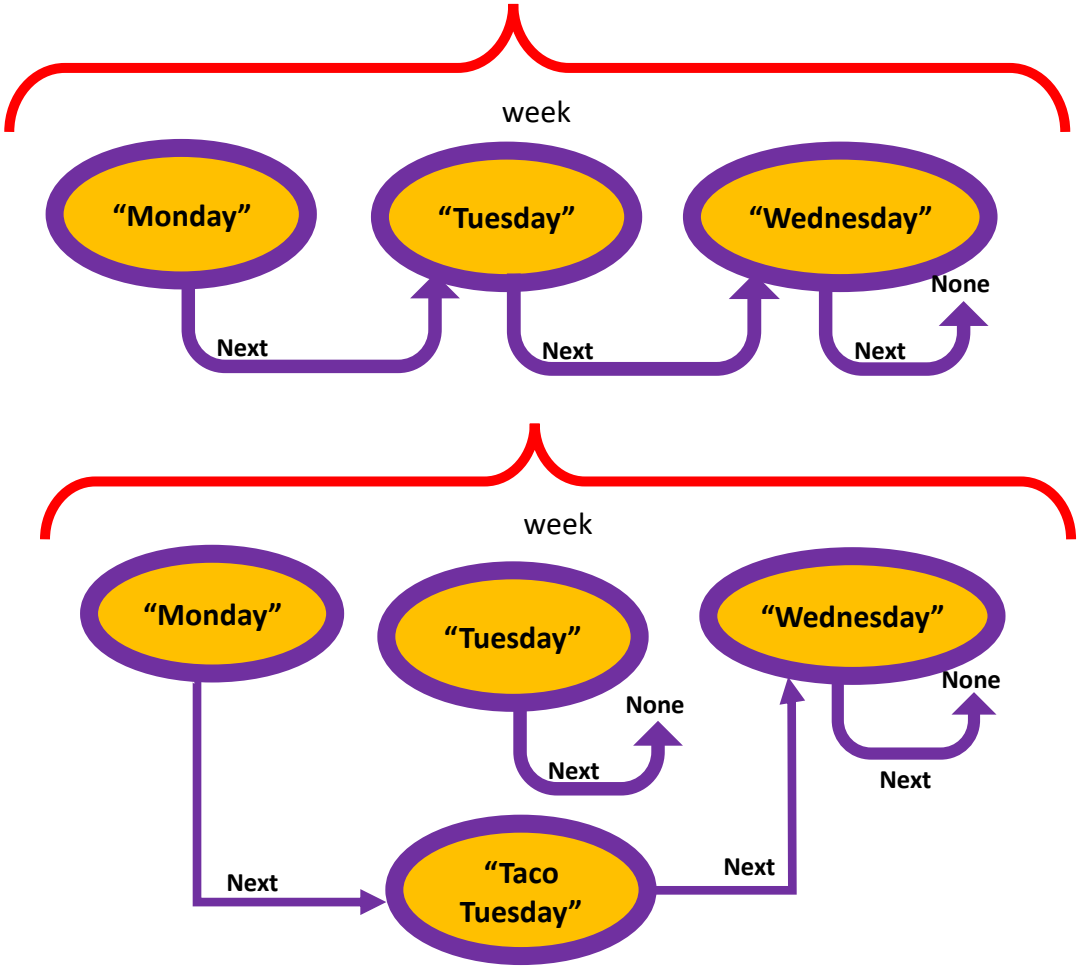
```
last_node.next = NodeToAdd
```

Note the use of '.next'. That's a node attribute. Its important to notice that we are using a different class inside this class. That's no stranger than using a string inside a list.

It might be helpful to look at how that conditional in the append method graphically



This is a way we can iteratively navigate these advanced data structures until we find the right position to do what we want. We will return to this process over and over. The syntax will get a bit more complex but the core principle is the same



Lets look at a little more complicated situation. Imagine we want to replace the "Tuesday" node with a "Taco Tuesday" Node.

Basically we have to 'traverse' the LinkedList as before, until we find the one we want to replace, and then we have to change the pointers before and after. Just like with append, dealing with the head is a different situation than dealing with the others. Look at the code at the bottom of tinyurl.com/LinkedList507 and fix the error