Python doesn't have a built-in implementation of linked lists, we have to build it on our own.

- We start by creating the blueprint for each node

In [53]:

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

When we instanciate a node, the class will pass the data we want the node to hold. The data passed during this process will be stored in self.data, and the self.next will work as a pointer to the next node on the list, which will always point to null (None) when we create a new node.

- Then, we create the class LinkedList: will have head and tail pointers;
- Optional: store a value of length to help keeping track of the linked list size

Upon creation, the linked list will be empty without any nodes to point to, so head will point to 'None' at this stage. Also because it's empty, the tail will point to whatever the head is pointing to('None').

In [54]:

```
class LinkedList():
    def __init__(self):
        self.head = None
        self.tail = self.head
        self.length = 0

# Next, we will add the append method: adds nodes to the end of the linked list
    def append(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
            self.tail = self.head
            self.length = 1
        else:
            self.tail.next = new_node
            self.tail = new_node
            self.length += 1
```

```python
# We pass the data we want to append and the method creates a new instance of the Node class (creating a new node
# Then, the method checks whether the list is empty; if so, it points the head to the newly created node, the tail
# If the list is not empty: the 'next' pointer of the last node (pointed by tail) will reference the new node, the

# Next, we will implement the prepend method: adds a node to the head of the linked list
    def prepend(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
            self.tail = self.head
            self.length += 1
        else:
            new_node.next = self.head
            self.head = new_node
            self.length += 1
# We pass the data and the method creates a new instance of the Node class (just as above)
# The 'next' pointer of new_node will reference the head, which is currently pointing towards the first node of th
# Then, the head will point to new_node, because we want it to become the first node - the head
# Finally, we increase length by 1

# Afterwards, we create a function that prints the values in the nodes of the linked lists
    def print_list(self):
        if self.head == None:
            print('Empty')
        else:
            current_node = self.head
            while current_node != None:
                print(current_node.data, end = ' ')
                current_node = current_node.next
        print()
# First, it checks if the list is empty; if so, the output will be 'Empty'
# Otherwise, it creates current_node pointing to the head of the linked list
# Then, it loops until the node we created becomes 'None'
# Inside the loop, it will print the data of current_node. Then, it will make current_node equal to the node refer
# Because we traverse the full length of the linked list, the time complexity of this operation is O(n)

# Then, we created the insert method: inserts data at a specified position
    def insert(self, position, data):
        if position >= self.length:
            if position > self.length:
                print('Position unavailable. Appending at the end of the list.')
```

```
            new_node = Node(data)
            self.tail.next = new_node
            self.tail = new_node
            self.length += 1
        elif position == 0:
            new_node = Node(data)
            new_node.next = self.head
            self.head = new_node
            self.length += 1
        else:
            new_node = Node(data)
            current_node = self.head
            for i in range(position - 1):
                current_node = current_node.next
            new_node.next = current_node.next
            current_node.next = new_node
            self.length += 1
# If the position is greater than the length of the list, it follows the append operation (adds node to end of lis
# If it's equal to 0, it follows the prepend operation (adds node to the beginning of list)
# If it's somewhere between the previous positions, it creates a temporary node which traverses the list up to the
# The new 'next' pointer of the temporary node will refer to the next node on the list, which is the position requ
# The temporary node and the new node will be pointing to the same position
# Then, it updates the 'next' pointer of the temporary node towards the new node which makes it take up the intend
# The node that previously occupying that position gets pushed towards the next position
# The time complexity of this operation will be O(n), because it requires traversal of the list

# Next, we create a method that allows the user to enter a value: if found, the method will delete it; if found mu
    def delete_by_value(self, data):
        if self.head == None:
            print('Nothing to delete. Linked list is empty.')
            return
        current_node = self.head
        if current_node.data == data:
            self.head = self.head.next
            if self.head == None or self.head.next == None:
                self.tail = self.head
            self.length -= 1
            return
        while current_node.next != None and current_node.next.data != data:
            if current_node.data == data:
                previous_node.next = current_node.next
```

```
                return
            current_node = current_node.next
        if current_node.next != None:
            current_node.next = current_node.next.next
            if current_node.next == None:
                self.tail = current_node
            self.length -= 1
            return
        else:
            print('Given value not found.')
# For starters, we check if the list is empty. If so, the relevant message is output. If not, a temporary node is
# After that, we verify if the value of the head equals the value the user requested to delete.
# If so, the head will become the node pointed by the 'next' pointer of the head.
# Then, we check if there is only one node or none in the list.
# If so, the tail will equal the head: original head no longer attached to the list and the second node becomes he
# If neither situations occur, we traverse the list and check every node.
# This is done by looping until either the current node becomes 'None'(end of list), or until the data of the node
# After coming out of the loop, if the current node is not equal to 'None', then the next node of the current node
# Therefore, the 'next' pointer of the current node will point to the node 2 positions next to the current node, b
# We establish a connection between the current node and the node 2 positions next to the current node, which disc
# After deleting it, we check if the current node's 'next' point refers to 'None' (if it's the tail); if so, then
# If the current node is equal to 'None', we traversed the entired list and the value could not be found.
# The time complexity of this operation is O(n).

# At last, we create a method that allows us to delete a node based on its position, similarly to the delete_by_va
    def delete_by_position(self, position):
        if self.head == None:
            print('Nothing to delete. Linked List is empty.')
            return
        if position == 0:
            self.head = self.head.next
            if self.head == None or self.head.next == None:
                self.tail = self.head
            self.length -= 1
            return
        if position >= self.length:
            position = self.length - 1
        current_node = self.head
        for i in range(position - 1):
            current_node = current_node.next
        current_node.next = current_node.next.next
```

```
        self.length -= 1
        if current_node.next == None:
            self.tail = current_node
        return
# Instead of traversing the list until the current node becomes 'None' or the next node equals the required data,
# Then, we bypass the next node to the current node and connect it to the node 2 positions after the current node.
# Like the delete_by_value method, we also check for tail and update it accordingly.
# The time complexity of this operation is O(n).
```

In [55]:

```
my_linked_list = LinkedList()
my_linked_list.print_list()
```

```
Empty
```

In [56]:

```
my_linked_list.append(5)
my_linked_list.append(2)
my_linked_list.append(9)
my_linked_list.print_list()
```

```
5 2 9
```

In [57]:

```
my_linked_list.prepend(4)
my_linked_list.print_list()
```

```
4 5 2 9
```

In [58]:

```
my_linked_list.insert(2,7)
my_linked_list.print_list()
```

```
4 5 7 2 9
```

In [59]:

```
my_linked_list.insert(0,0)
my_linked_list.insert(6,0)
my_linked_list.insert(9,3)
my_linked_list.print_list()
```

```
Position unavailable. Appending at the end of the list.
0 4 5 7 2 9 0 3
```

In [60]:
```python
my_linked_list.delete_by_value(3)
my_linked_list.print_list()
```

```
0 4 5 7 2 9 0
```

In [61]:
```python
my_linked_list.delete_by_value(0)
my_linked_list.print_list()
```

```
4 5 7 2 9 0
```

In [62]:
```python
my_linked_list.delete_by_position(3)
my_linked_list.print_list()
```

```
4 5 7 9 0
```

In [63]:
```python
my_linked_list.delete_by_position(0)
my_linked_list.print_list()
```

```
5 7 9 0
```

In [64]:
```python
my_linked_list.delete_by_position(8)
my_linked_list.print_list()
```

```
5 7 9
```

In [65]:
```python
print(my_linked_list.length)
```

```
3
```