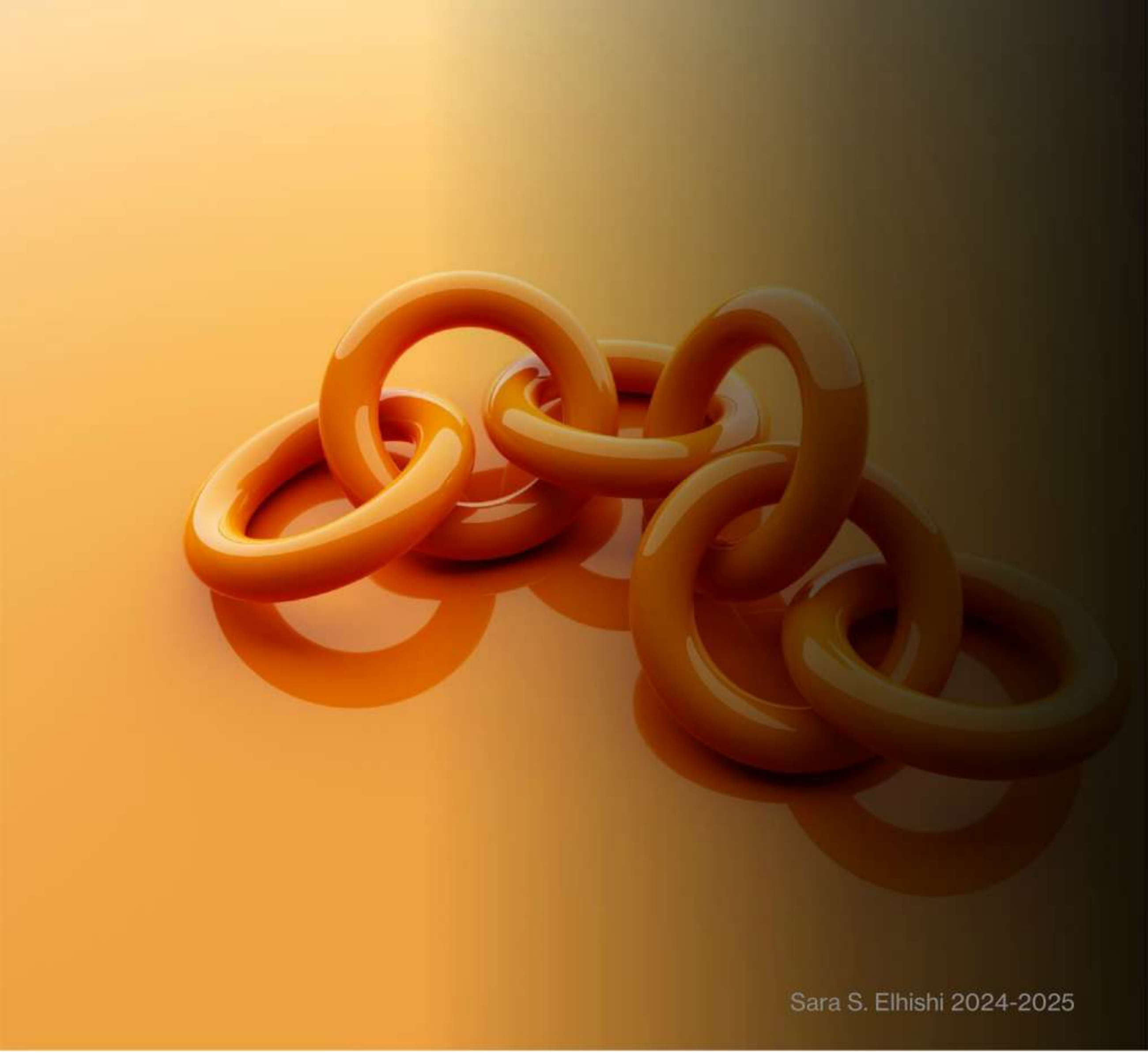


Lecture 6 Linked Lists

Dr. Sara S. Elhishi

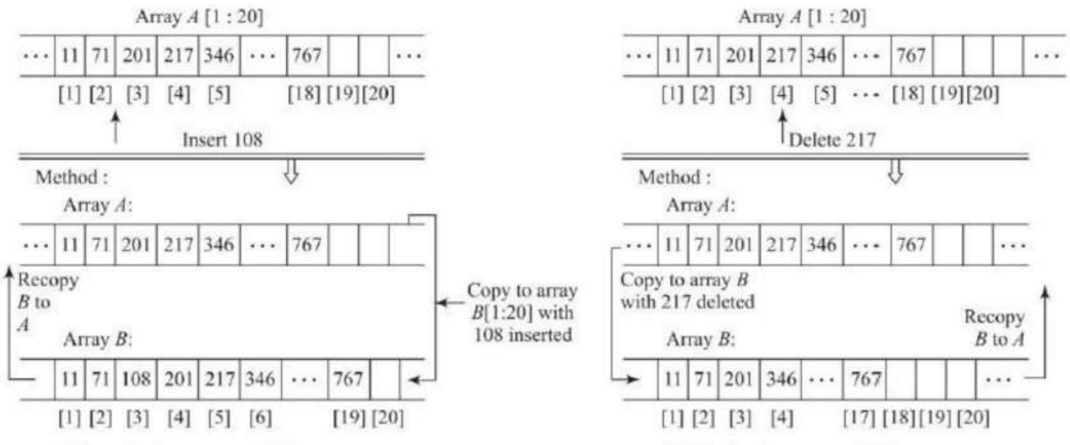
Information systems Dept.

Mansoura University

Sara_shaker2008@mans.edu.eg

Array Limitations

- Searching in an unordered array is inefficient, but insertion in an ordered array is time-consuming.
- The size of an array is immutable once it has been initialized.
- Modifying array structure requires several shifts which is time consuming
- Storage memory inefficiency since it forces you to allocate chunks of free memory
- Poorly implemented insertion/deletion operations

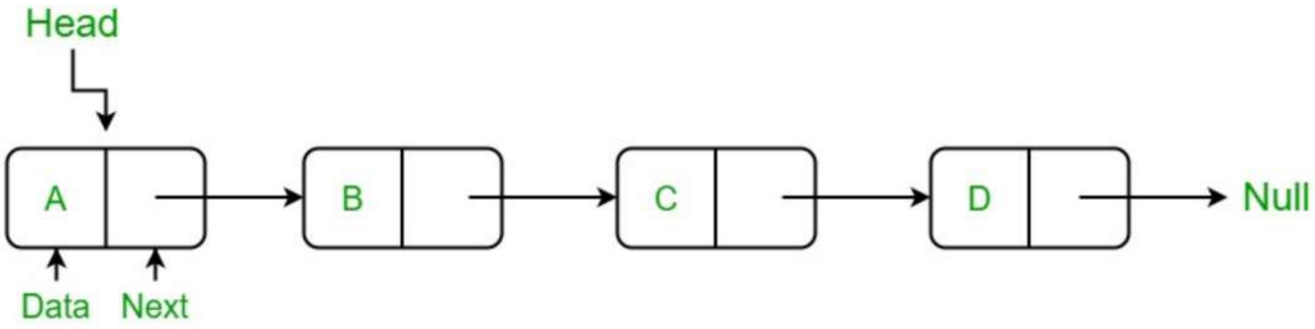


(a) Insertion in a sequential data structure

(b) Deletion in a sequential data structure

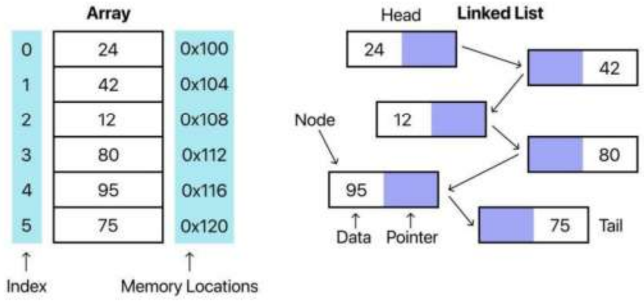
Linked Structure Benefits

- The execution of Insertion and deletion operations does not involve any data movement between neighboring elements
- Memory fragmentation is less likely to occur during the operation and administration of linked data structures.



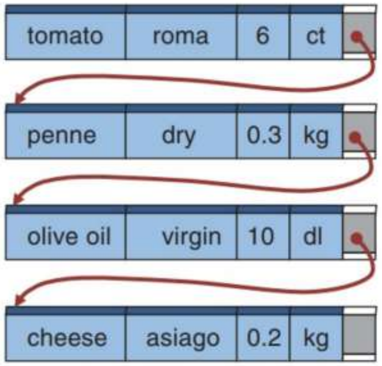
Array Vs. Linked List

- A linked list is characterized by its Links.
- Each element has a notion of what the next element is, but not necessarily where it is in the list.
- An array's different. There is nothing in one element of the array that says “Here’s your next element", instead an array knows what the next element is by what the next index is.

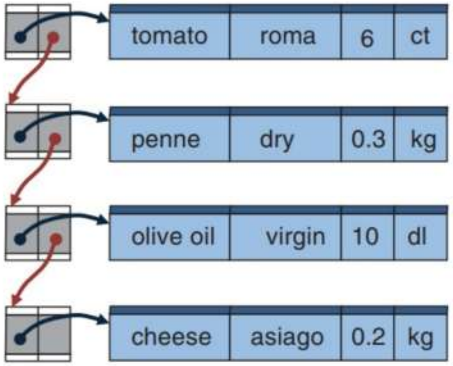


Links

- Each element in a linked list is encapsulated within a **Node/link** structure.
- Every link contains **data** and a **reference** to access the subsequent link in the sequence.
- A **reference**, essentially a pointer directing to another record.
- The reference can be stored within the data record or within a field within a two-field structure specifically built as a versatile list.



Records linked into a list

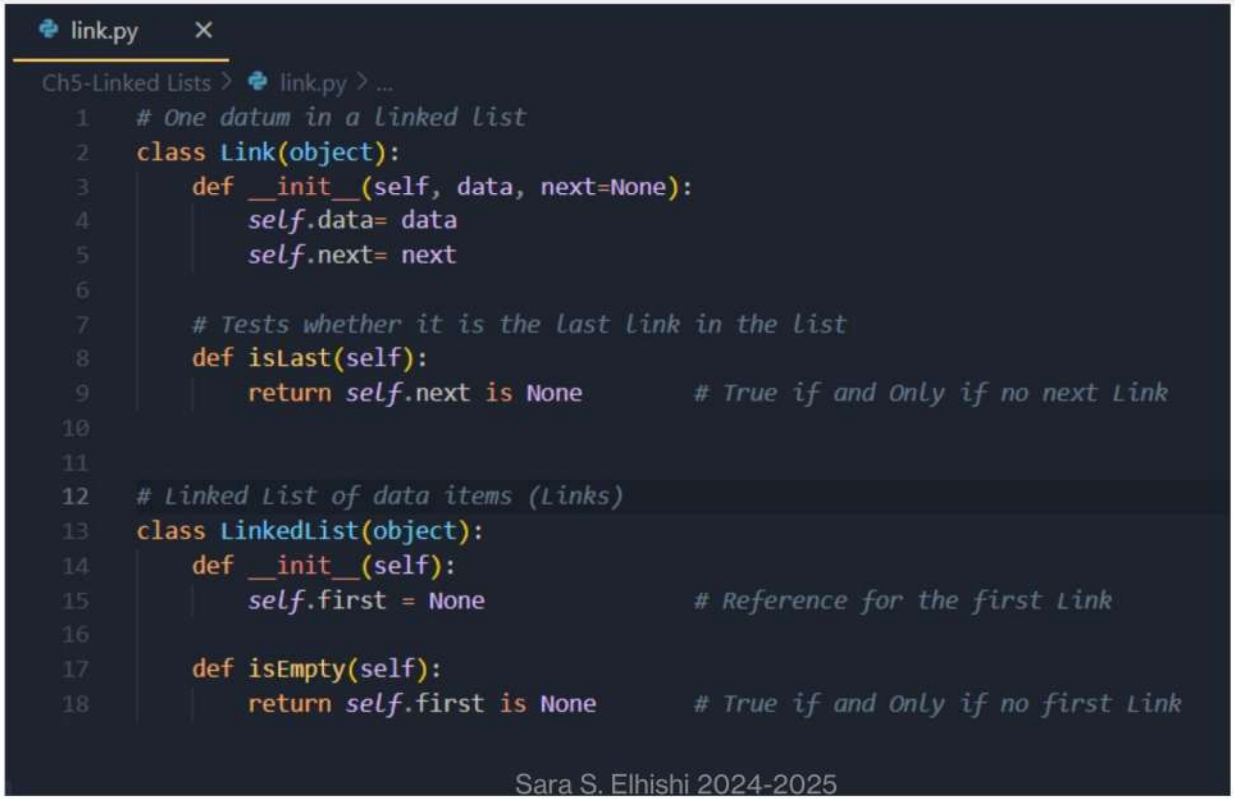


Linked list of records

The Link and LinkedList Classes

- We'll implemented the linked list class part by part.
- First, let's create a simple Link and LinkedList classes
- Every Link consists of two fields: one for the data and one for the subsequent link.
- The LinkedList requires only a single attribute, serves as a reference to the first Link object in the list if one exists.

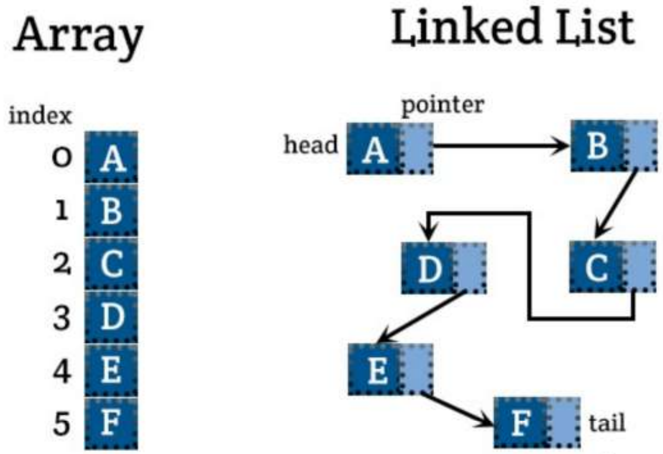
The Link and LinkedList Classes



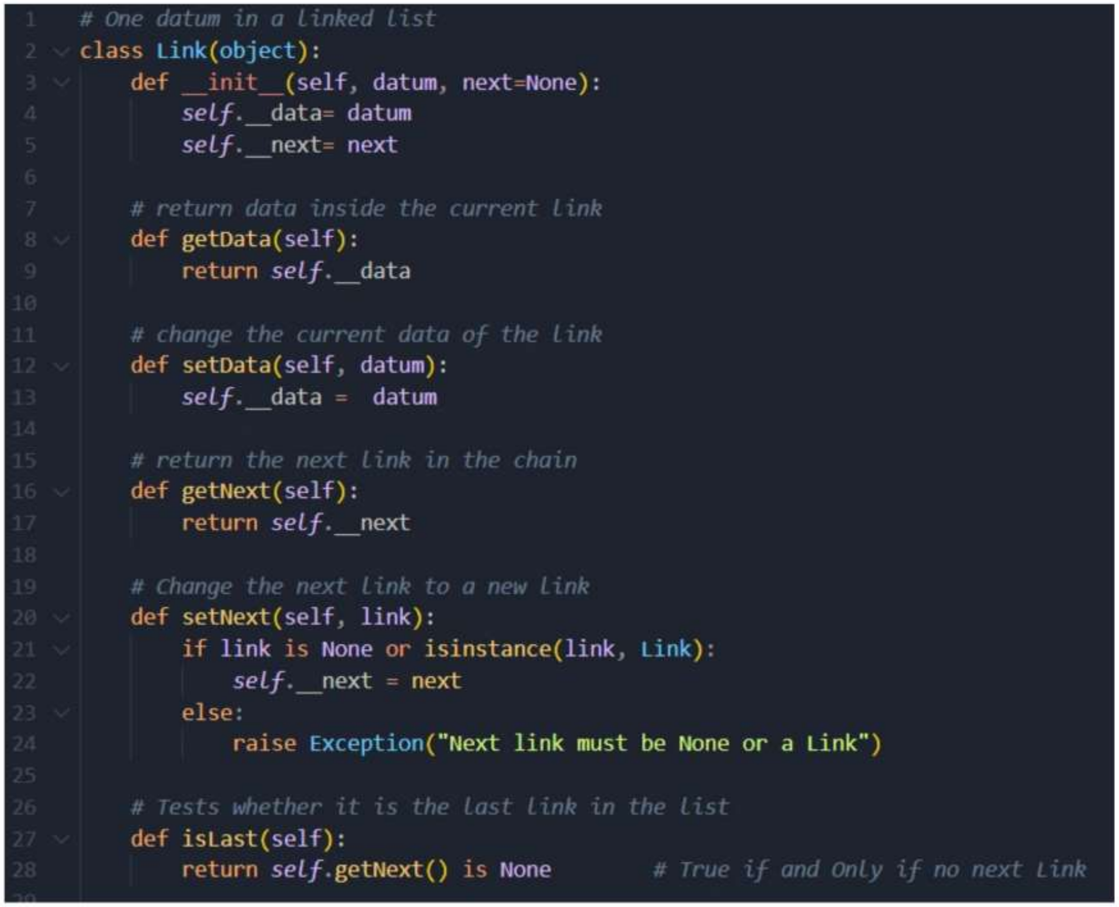
```python
# One datum in a linked list
class Link(object):
    def __init__(self, data, next=None):
        self.data= data
        self.next= next

    # Tests whether it is the last link in the list
    def isLast(self):
        return self.next is None       # True if and Only if no next Link


# Linked List of data items (Links)
class LinkedList(object):
    def __init__(self):
        self.first = None              # Reference for the first Link

    def isEmpty(self):
        return self.first is None      # True if and Only if no first Link
```

Relationship Not Position

- Each element in an array is assigned a position (index).
- Linked Lists use relationships to locate a specific item; you cannot access it directly.
- You start with the first item, go to the second, then the third, until you find what you're looking for or the end of the relationship.

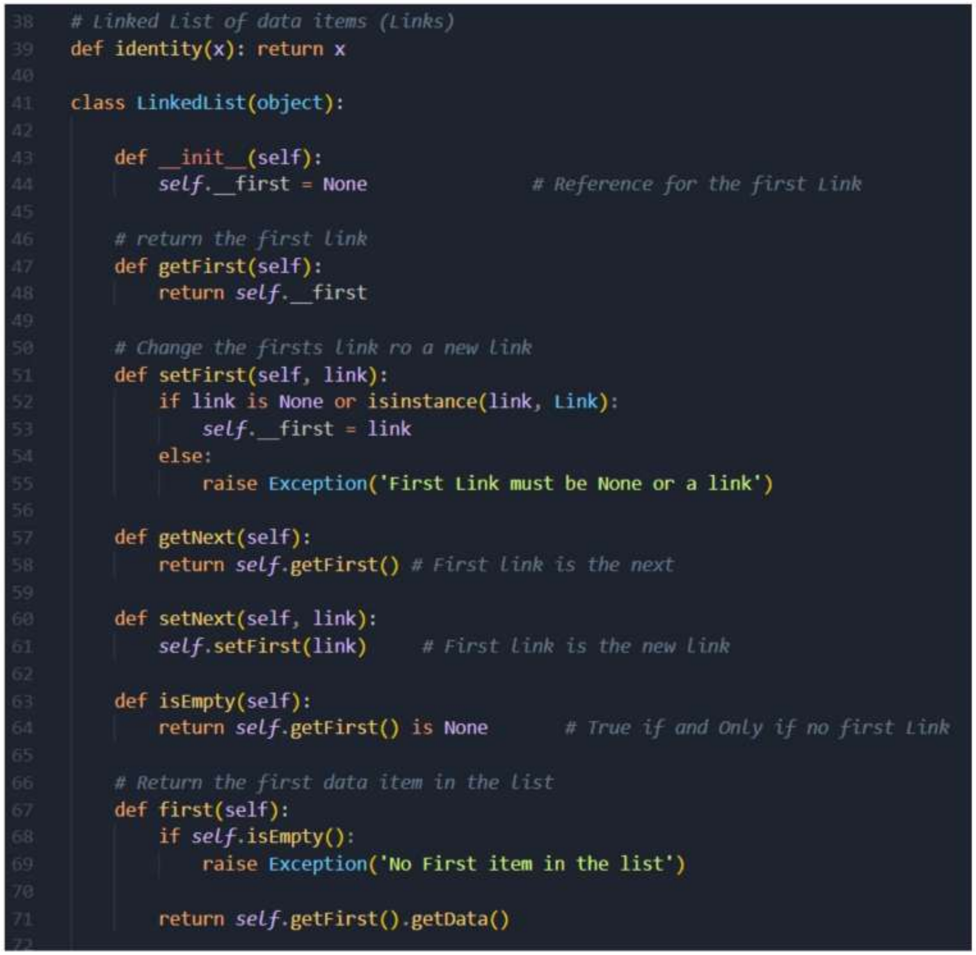


The Link Class Helper Methods



```python
# One datum in a linked list
class Link(object):
    def __init__(self, datum, next=None):
        self.__data= datum
        self.__next= next

    # return data inside the current link
    def getData(self):
        return self.__data

    # change the current data of the link
    def setData(self, datum):
        self.__data =  datum

    # return the next link in the chain
    def getNext(self):
        return self.__next

    # Change the next link to a new link
    def setNext(self, link):
        if link is None or isinstance(link, Link):
            self.__next = next
        else:
            raise Exception("Next link must be None or a Link")

    # Tests whether it is the last link in the list
    def isLast(self):
        return self.getNext() is None        # True if and Only if no next Link
```

The LinkedList Helper Methods

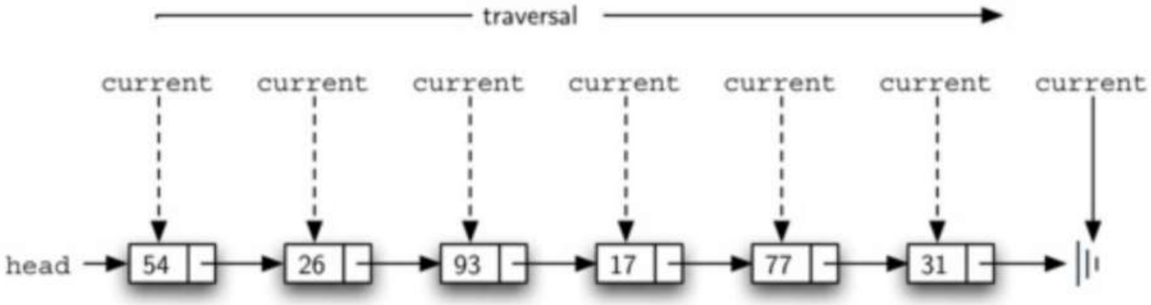


```python
# Linked List of data items (Links)
def identity(x): return x

class LinkedList(object):

    def __init__(self):
        self.__first = None                 # Reference for the first Link

    # return the first link
    def getFirst(self):
        return self.__first

    # Change the firsts link ro a new link
    def setFirst(self, link):
        if link is None or isinstance(link, Link):
            self.__first = link
        else:
            raise Exception('First Link must be None or a link')

    def getNext(self):
        return self.getFirst() # First link is the next

    def setNext(self, link):
        self.setFirst(link)     # First link is the new link

    def isEmpty(self):
        return self.getFirst() is None       # True if and Only if no first Link

    # Return the first data item in the list
    def first(self):
        if self.isEmpty():
            raise Exception('No First item in the list')

        return self.getFirst().getData()
```

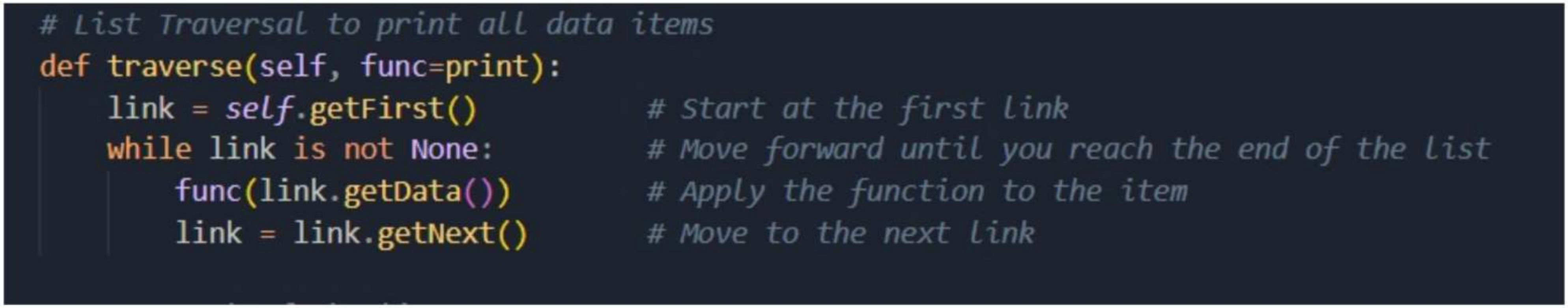
LinkedList Operations

Traversing a LinkedList

- Traversal is a process of visiting each element in a linked list to perform tasks such as printing a list, searching, or computing it length.

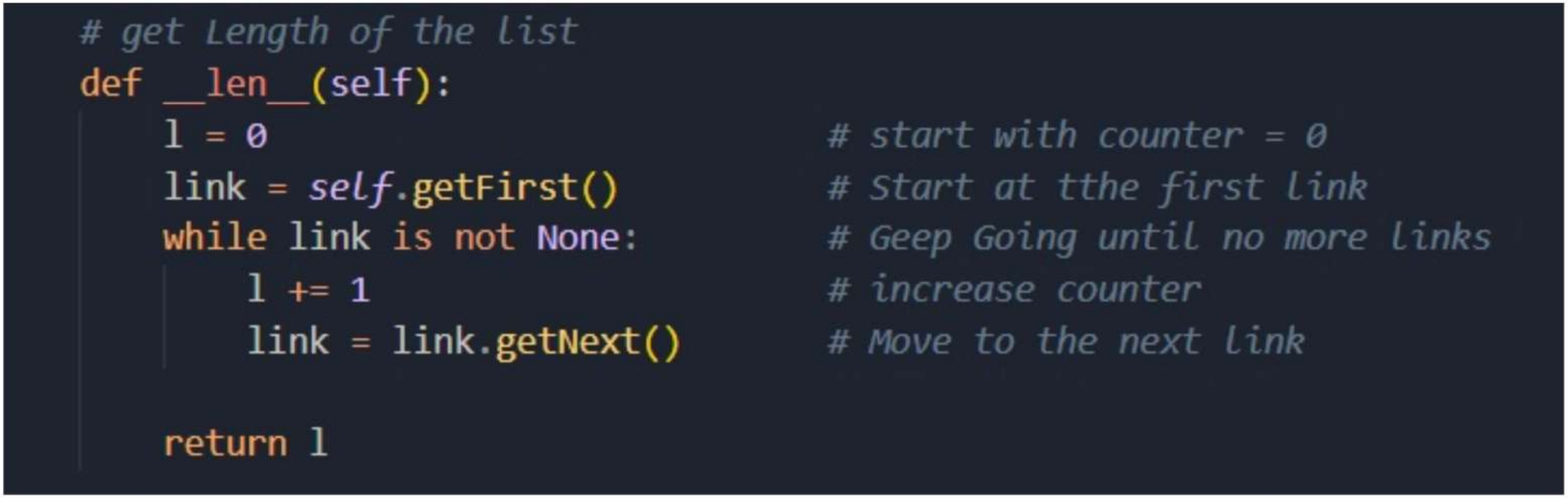


Traversing a LinkedList



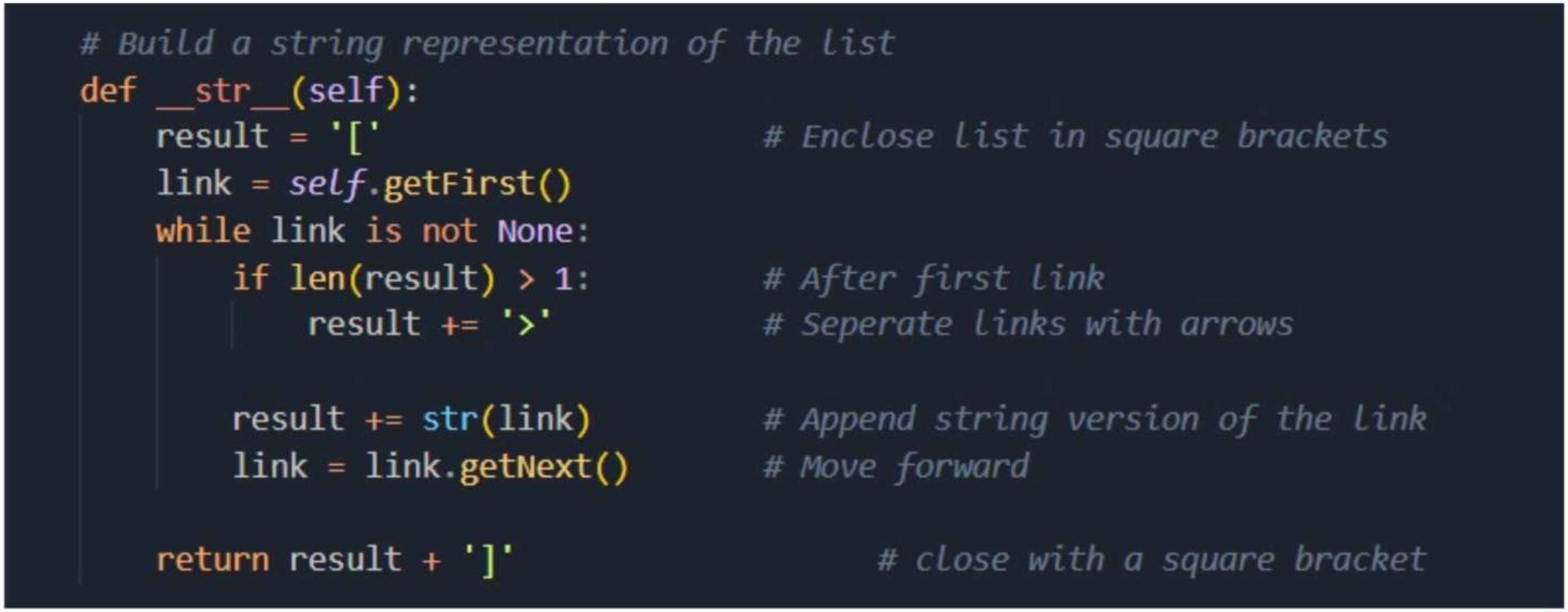
```python
# List Traversal to print all data items
def traverse(self, func=print):
    link = self.getFirst()          # Start at the first link
    while link is not None:         # Move forward until you reach the end of the list
        func(link.getData())        # Apply the function to the item
        link = link.getNext()       # Move to the next link
```

Length of LinkedList



```
# get Length of the list
def __len__(self):
    l = 0                           # start with counter = 0
    link = self.getFirst()          # Start at tthe first link
    while link is not None:         # Geep Going until no more links
        l += 1                      # increase counter
        link = link.getNext()       # Move to the next link

    return l
```

Printing a LinkedList

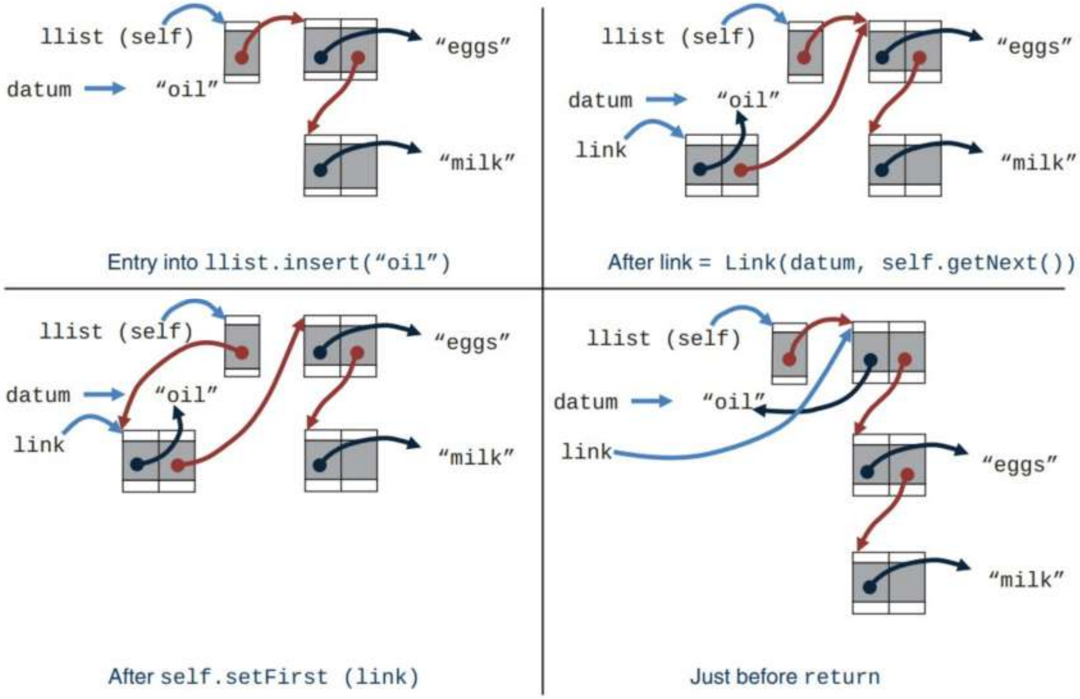


```
# Build a string representation of the list
def __str__(self):
    result = '['                    # Enclose list in square brackets
    link = self.getFirst()
    while link is not None:
        if len(result) > 1:         # After first link
            result += '>'           # Seperate links with arrows

        result += str(link)         # Append string version of the link
        link = link.getNext()       # Move forward

    return result + ']'                   # close with a square bracket
```

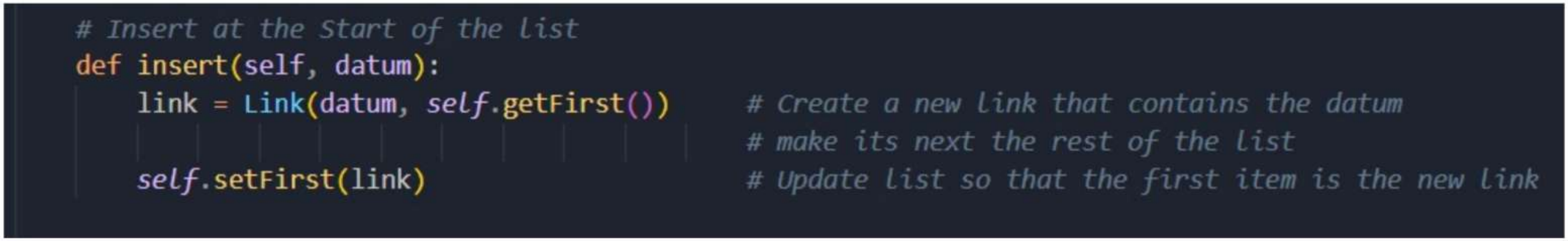
Insertion

Insert at The Beginning

- Create a new “Link” object
- Assign it the new value as its datum
- Assign the rest of the linked list as its next.
- Set the newLink as the First link in the linked list



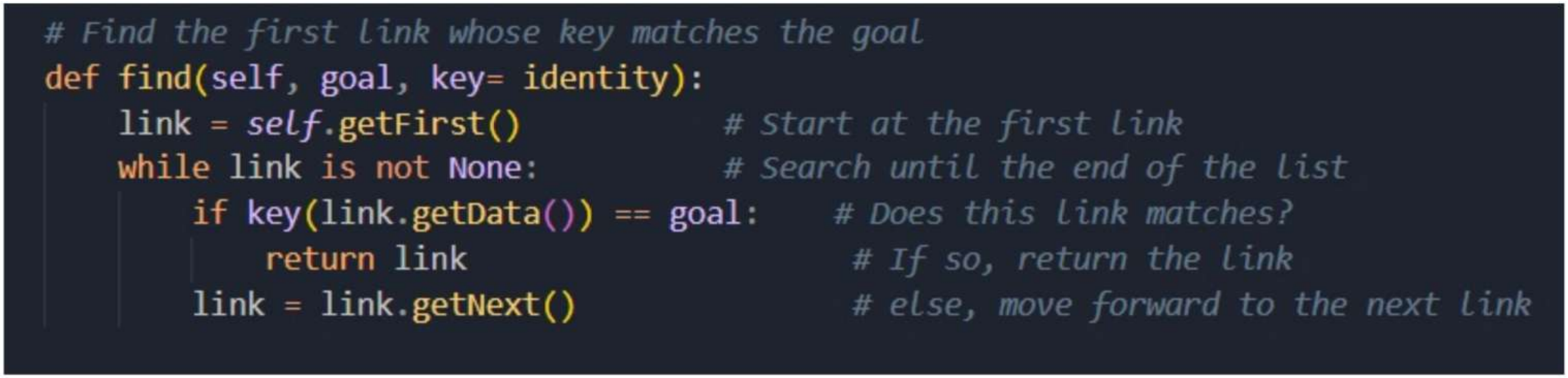
Insert at The Beginning



```
# Insert at the Start of the list
def insert(self, datum):
    link = Link(datum, self.getFirst())     # Create a new link that contains the datum
                                            # make its next the rest of the list
    self.setFirst(link)                     # Update list so that the first item is the new link
```

InsertAfter- Locate position

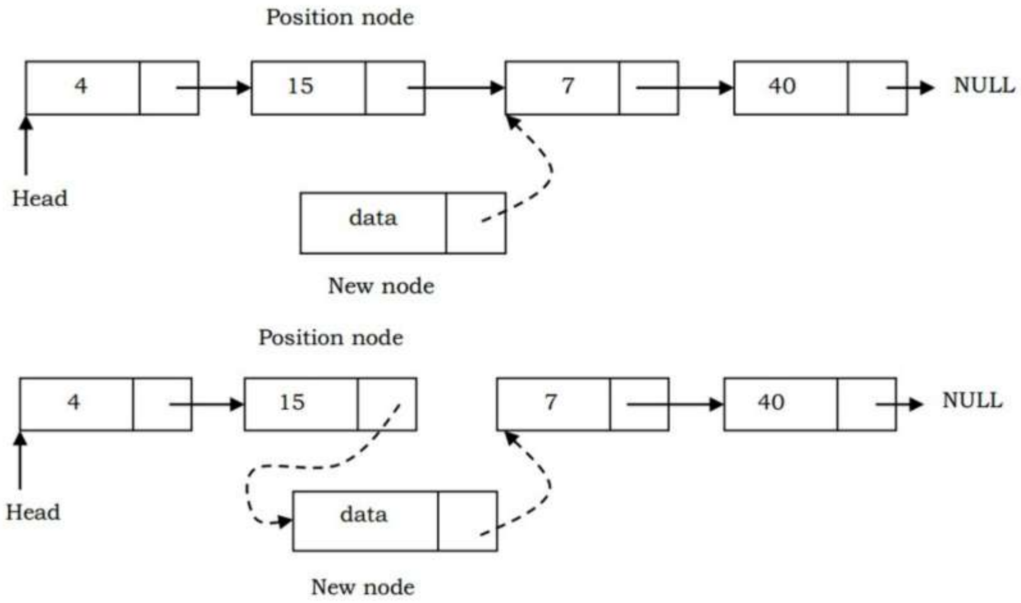
- If the linked list only allows insertion and deletion at the beginning, the data structure would function as a stack.
- To perform actions such as inserting an element at a specific position in a list,
- you must locate components that possess a specific key, precisely the purpose of the find() method.



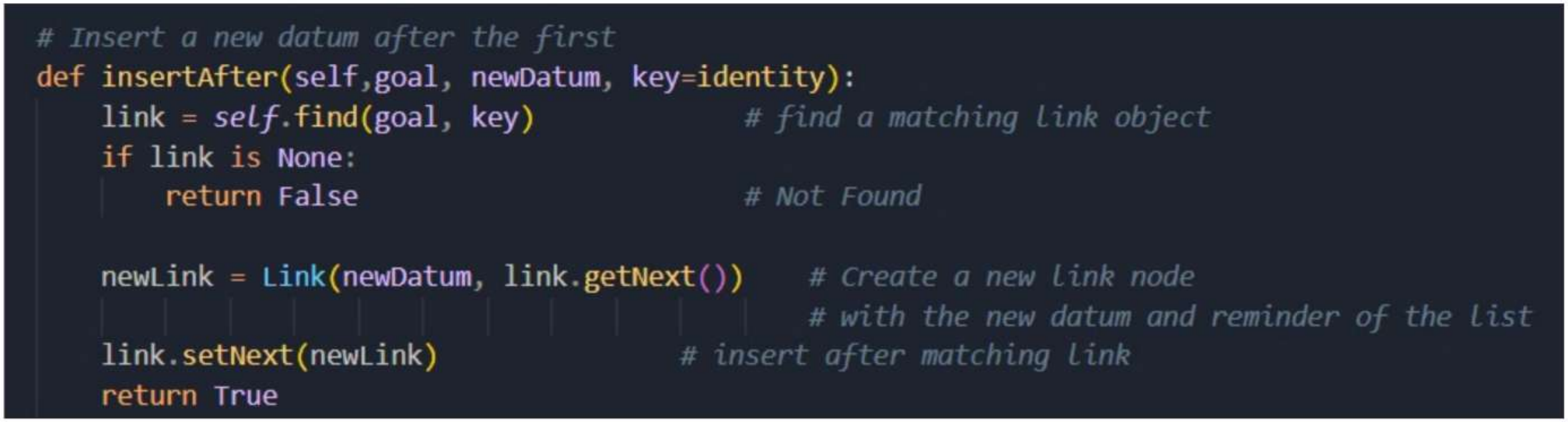
```python
# Find the first link whose key matches the goal
def find(self, goal, key= identity):
    link = self.getFirst()           # Start at the first link
    while link is not None:          # Search until the end of the list
        if key(link.getData()) == goal:    # Does this link matches?
            return link                     # If so, return the link
        link = link.getNext()               # else, move forward to the next link
```

InsertAfter

- Insert after a specific link.
- The find() method is utilized to find the specific link key (e.g., 15).
- Create a newLink node (e.g., New nodel) whose next is the rest of the linkedlist.
- Update the next link of "15" to the newlink node



InsertAfter(beforeLink, newLink)

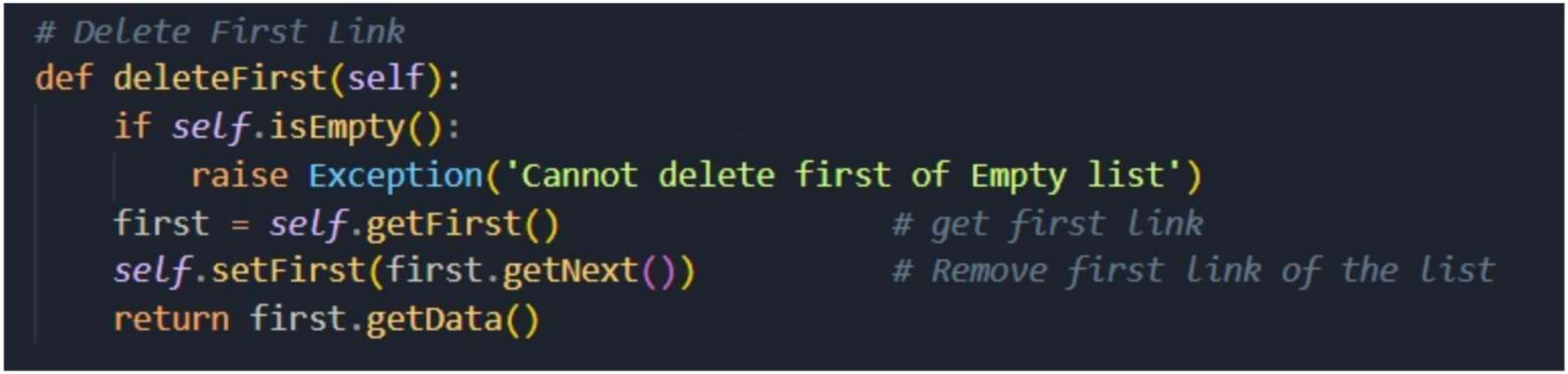


```python
# Insert a new datum after the first
def insertAfter(self,goal, newDatum, key=identity):
    link = self.find(goal, key)             # find a matching link object
    if link is None:
        return False                        # Not Found

    newLink = Link(newDatum, link.getNext())    # Create a new link node
                                                # with the new datum and reminder of the list
    link.setNext(newLink)               # insert after matching link
    return True
```

Deletion

Delete First Link

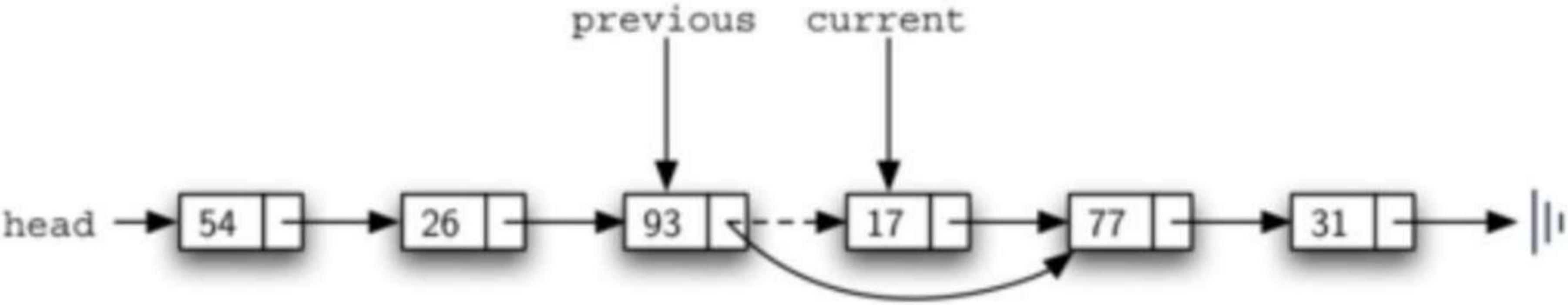
Set the First link in the list as the next of the old first.



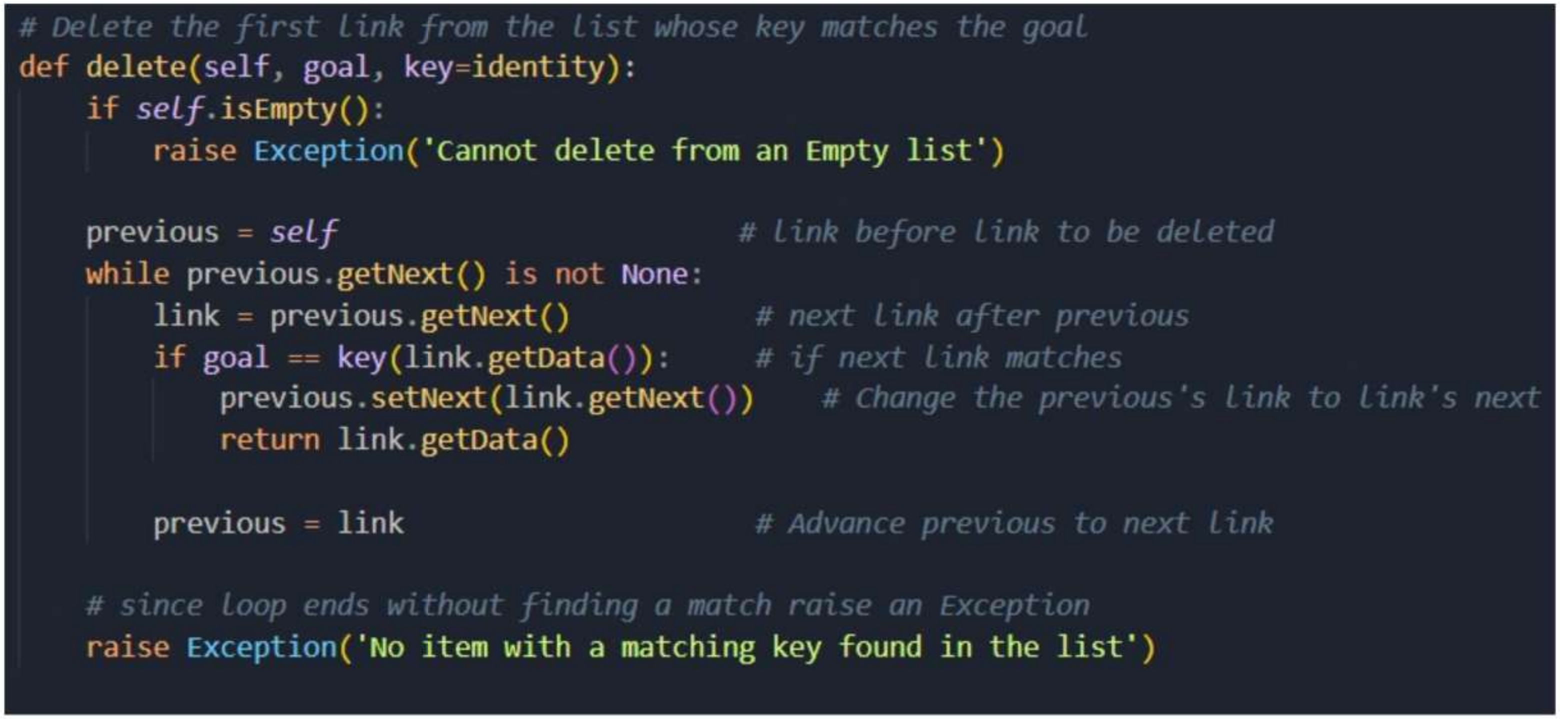
```
# Delete First Link
def deleteFirst(self):
    if self.isEmpty():
        raise Exception('Cannot delete first of Empty list')
    first = self.getFirst()                     # get first link
    self.setFirst(first.getNext())              # Remove first link of the list
    return first.getData()
```

Delete a Specific Link

- First locate the Link where the deletion should occur (find())
- Then modify the pointers accordingly

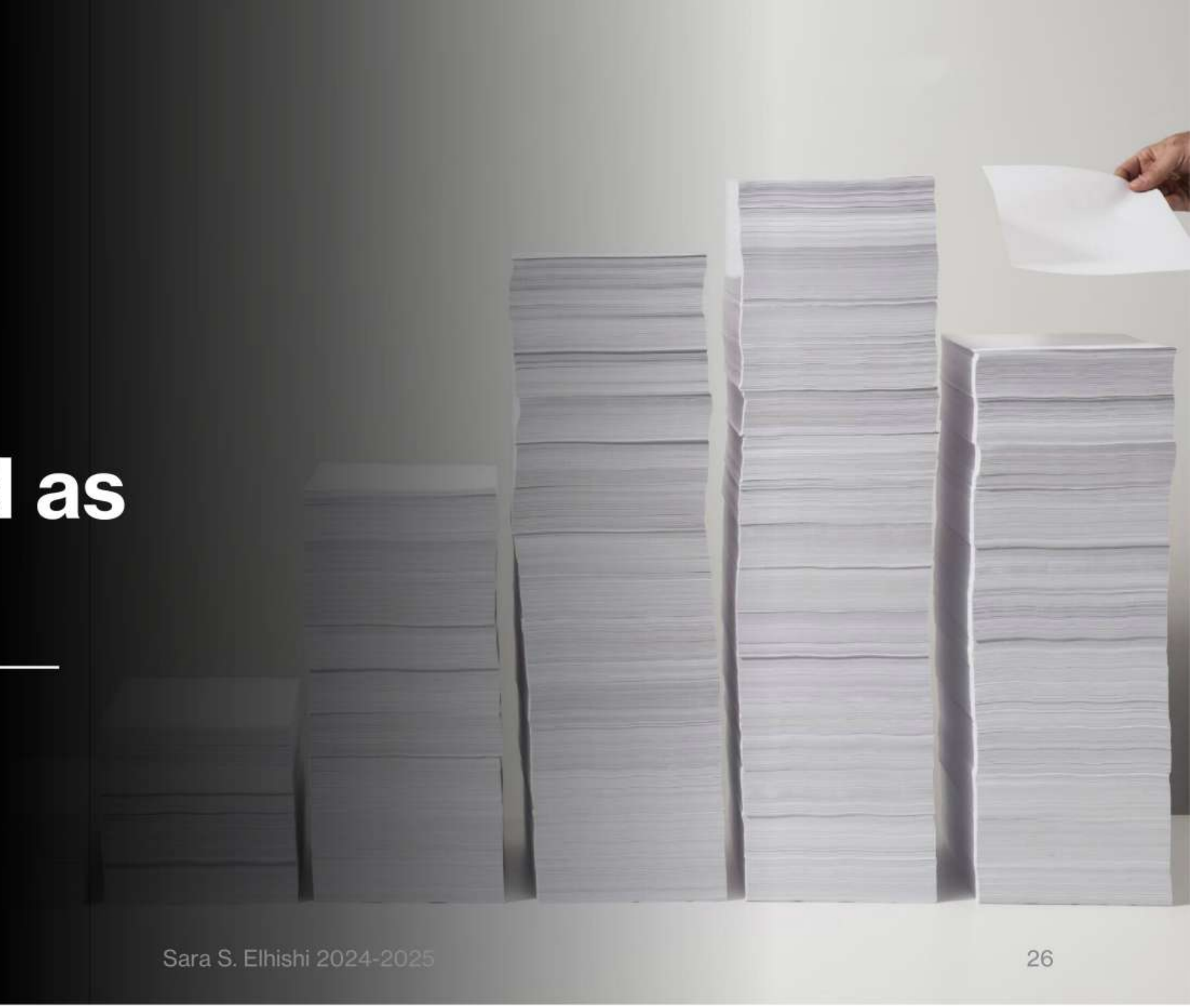


Delete()



```python
# Delete the first link from the list whose key matches the goal
def delete(self, goal, key=identity):
    if self.isEmpty():
        raise Exception('Cannot delete from an Empty list')

    previous = self                        # link before link to be deleted
    while previous.getNext() is not None:
        link = previous.getNext()           # next link after previous
        if goal == key(link.getData()):     # if next link matches
            previous.setNext(link.getNext())    # Change the previous's link to link's next
            return link.getData()

        previous = link                     # Advance previous to next link

    # since loop ends without finding a match raise an Exception
    raise Exception('No item with a matching key found in the list')
```

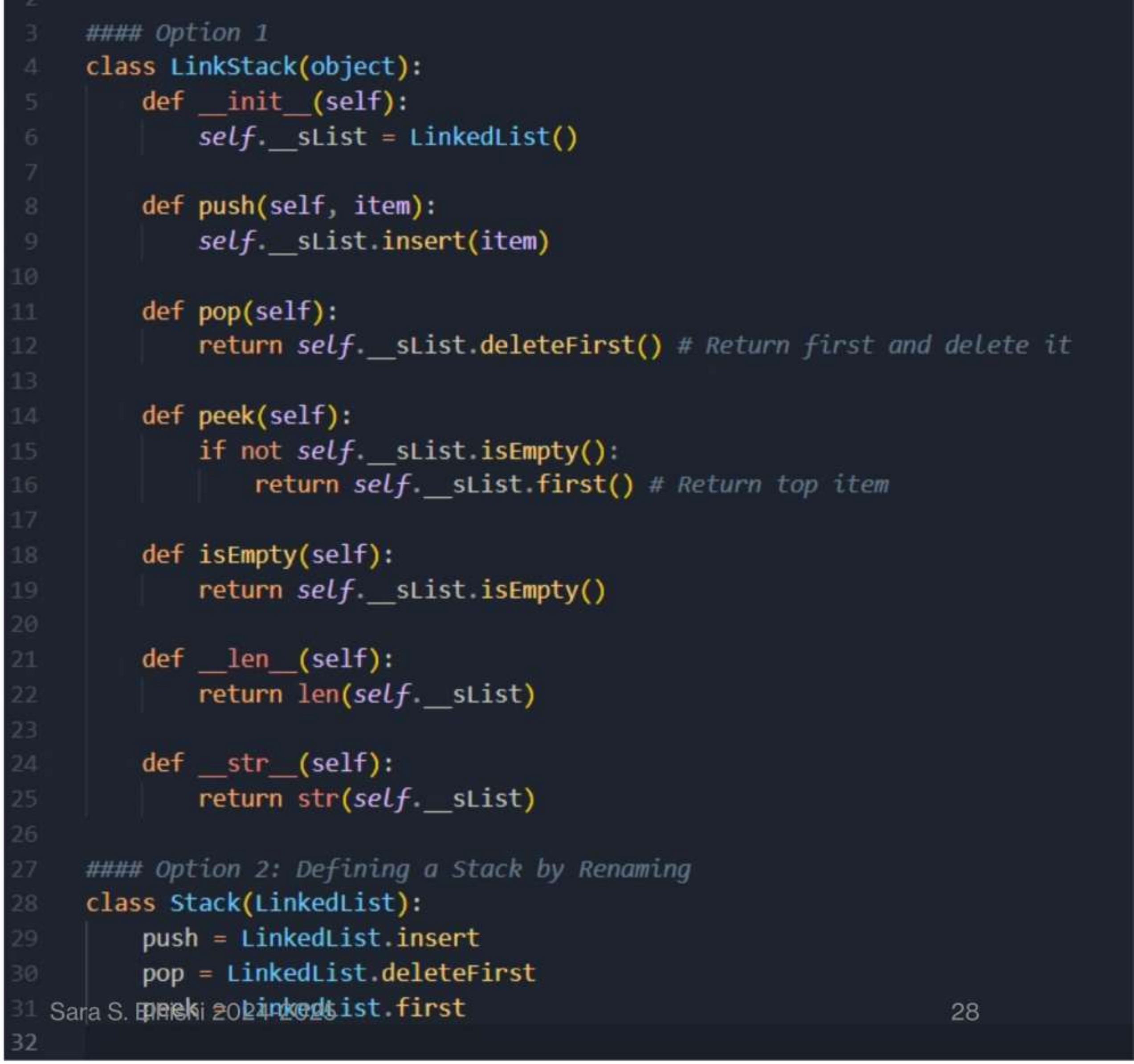
Stack Implemented as a Linked List



Dynamic Stack

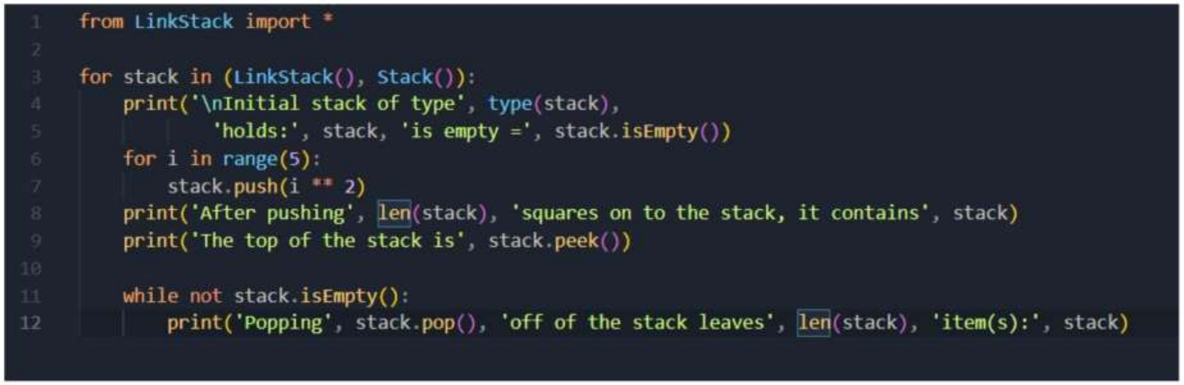
- Stack Implemented using lists can grow or shrink as required by the application at hand.
- Utilize the LinkedList class to modify push(), pop(), and peek() methods.

Stack implemented using LinkedList

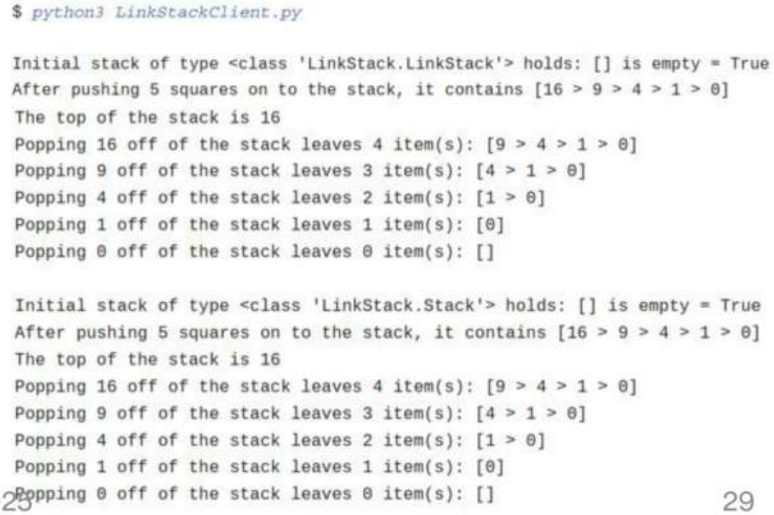


```python
#### Option 1
class LinkStack(object):
    def __init__(self):
        self.__sList = LinkedList()

    def push(self, item):
        self.__sList.insert(item)

    def pop(self):
        return self.__sList.deleteFirst() # Return first and delete it

    def peek(self):
        if not self.__sList.isEmpty():
            return self.__sList.first() # Return top item

    def isEmpty(self):
        return self.__sList.isEmpty()

    def __len__(self):
        return len(self.__sList)

    def __str__(self):
        return str(self.__sList)

#### Option 2: Defining a Stack by Renaming
class Stack(LinkedList):
    push = LinkedList.insert
    pop = LinkedList.deleteFirst
    peek = LinkedList.first
```

Stack Client



```python
from LinkStack import *

for stack in (LinkStack(), Stack()):
    print('\nInitial stack of type', type(stack),
            'holds:', stack, 'is empty =', stack.isEmpty())
    for i in range(5):
        stack.push(i ** 2)
    print('After pushing', len(stack), 'squares on to the stack, it contains', stack)
    print('The top of the stack is', stack.peek())

    while not stack.isEmpty():
        print('Popping', stack.pop(), 'off of the stack leaves', len(stack), 'item(s):', stack)
```

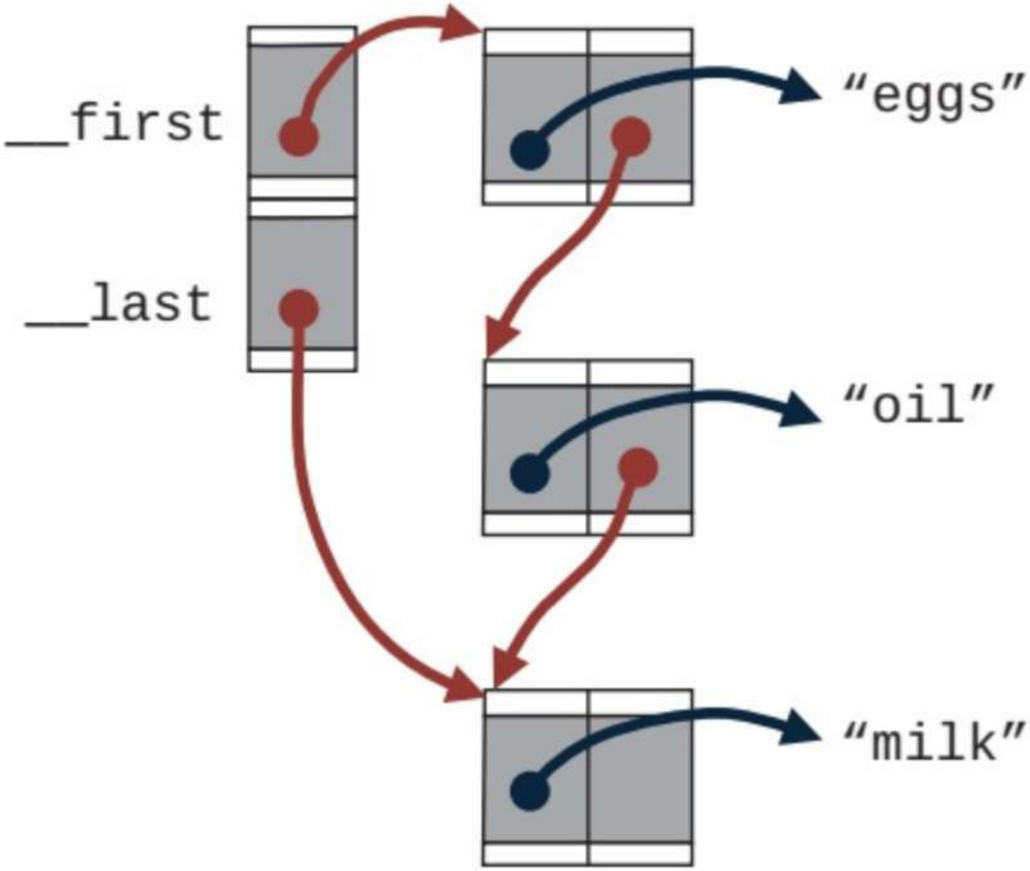


```
$ python3 LinkStackClient.py

Initial stack of type <class 'LinkStack.LinkStack'> holds: [] is empty = True
After pushing 5 squares on to the stack, it contains [16 > 9 > 4 > 1 > 0]
The top of the stack is 16
Popping 16 off of the stack leaves 4 item(s): [9 > 4 > 1 > 0]
Popping 9 off of the stack leaves 3 item(s): [4 > 1 > 0]
Popping 4 off of the stack leaves 2 item(s): [1 > 0]
Popping 1 off of the stack leaves 1 item(s): [0]
Popping 0 off of the stack leaves 0 item(s): []

Initial stack of type <class 'LinkStack.Stack'> holds: [] is empty = True
After pushing 5 squares on to the stack, it contains [16 > 9 > 4 > 1 > 0]
The top of the stack is 16
Popping 16 off of the stack leaves 4 item(s): [9 > 4 > 1 > 0]
Popping 9 off of the stack leaves 3 item(s): [4 > 1 > 0]
Popping 4 off of the stack leaves 2 item(s): [1 > 0]
Popping 1 off of the stack leaves 1 item(s): [0]
Popping 0 off of the stack leaves 0 item(s): []
```

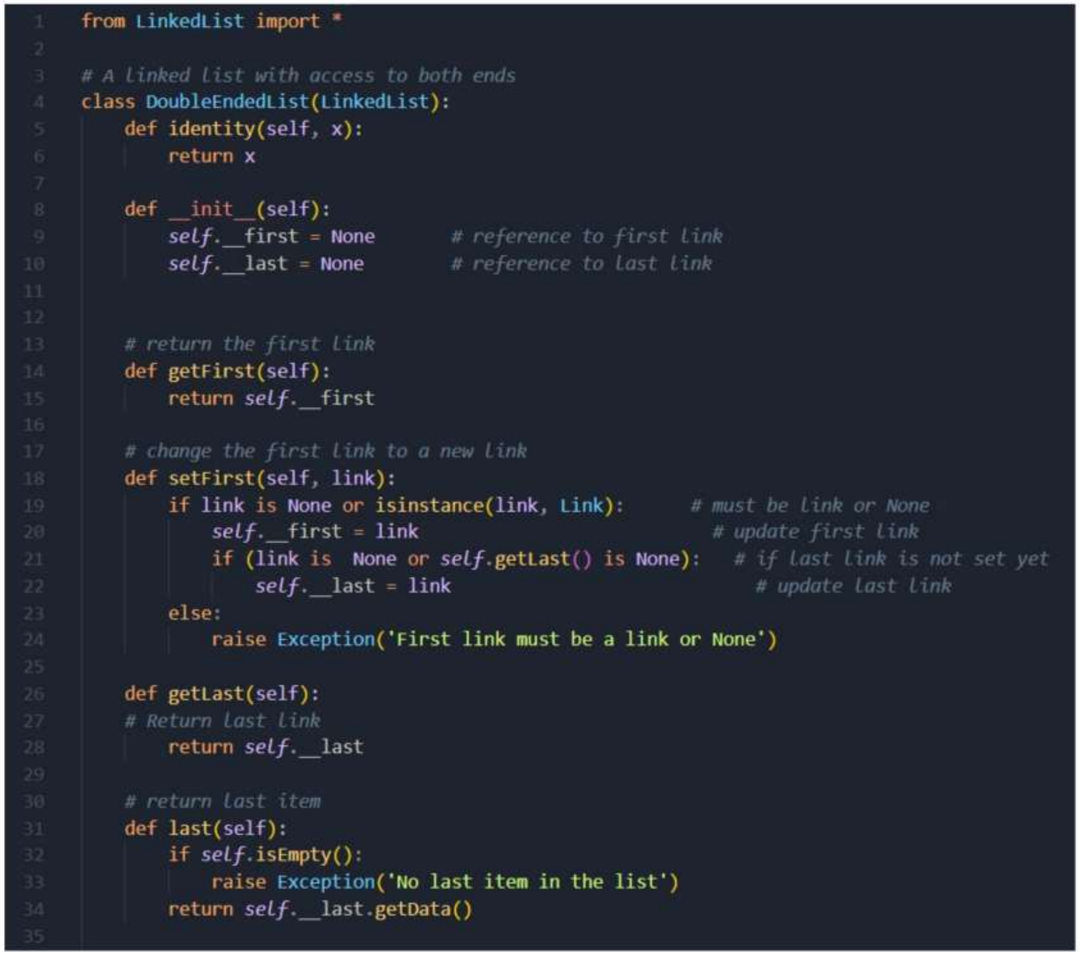
Double Ended List

Double Ended List

- Similar to a regular linked list, but it has an extra feature: a reference to both the start and last links.
- Allows for the straight insertion of new links both at the end and at the beginning of the list.

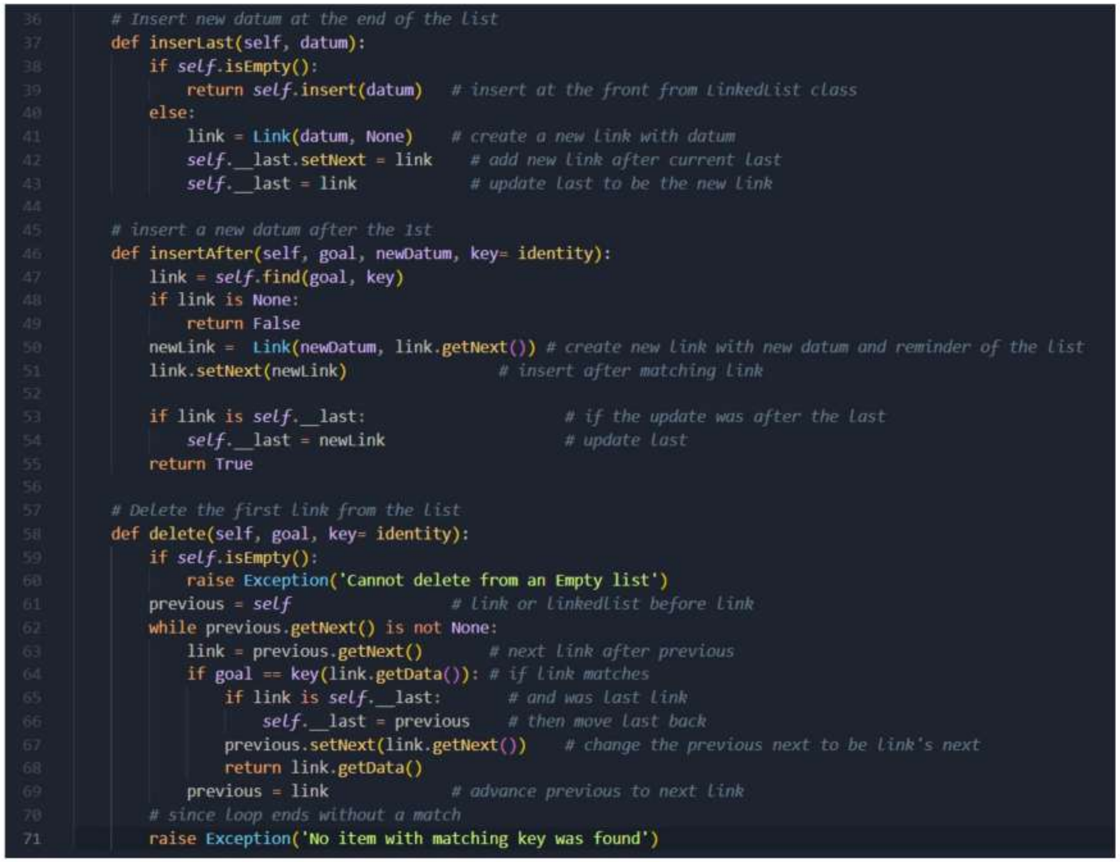


Double Ended List Implementation



```python
from LinkedList import *

# A linked list with access to both ends
class DoubleEndedList(LinkedList):
    def identity(self, x):
        return x

    def __init__(self):
        self.__first = None       # reference to first link
        self.__last = None        # reference to last link


    # return the first link
    def getFirst(self):
        return self.__first

    # change the first link to a new link
    def setFirst(self, link):
        if link is None or isinstance(link, Link):      # must be link or None
            self.__first = link                           # update first link
            if (link is  None or self.getLast() is None):   # if last link is not set yet
                self.__last = link                            # update last link
        else:
            raise Exception('First link must be a link or None')

    def getLast(self):
    # Return last link
        return self.__last

    # return last item
    def last(self):
        if self.isEmpty():
            raise Exception('No last item in the list')
        return self.__last.getData()

```

Double Ended List Implementation

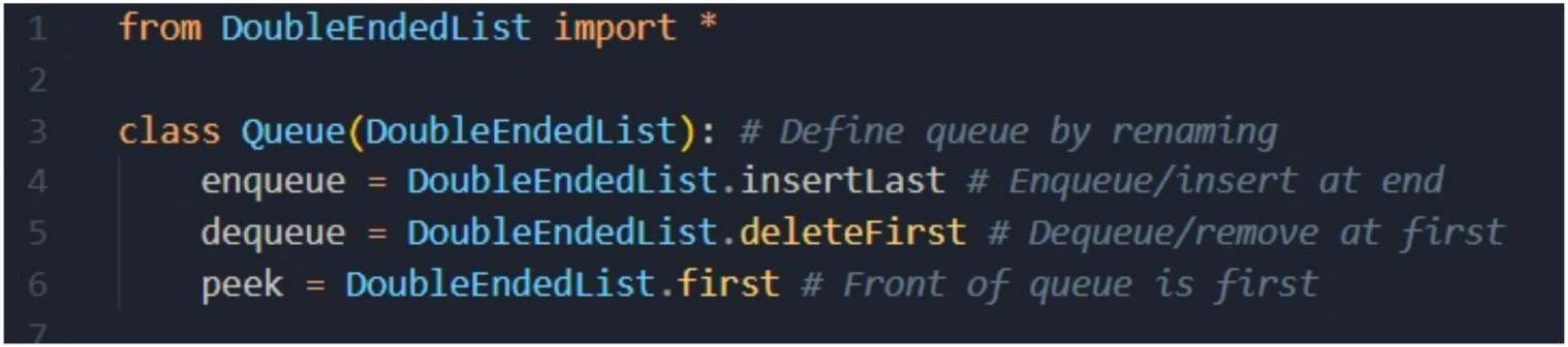


```python
    # Insert new datum at the end of the list
    def inserLast(self, datum):
        if self.isEmpty():
            return self.insert(datum)   # insert at the front from LinkedList class
        else:
            link = Link(datum, None)    # create a new link with datum
            self.__last.setNext = link    # add new link after current last
            self.__last = link            # update last to be the new link

    # insert a new datum after the 1st
    def insertAfter(self, goal, newDatum, key= identity):
        link = self.find(goal, key)
        if link is None:
            return False
        newLink =  Link(newDatum, link.getNext()) # create new link with new datum and reminder of the list
        link.setNext(newLink)                # insert after matching link

        if link is self.__last:                     # if the update was after the last
            self.__last = newLink                   # update last
        return True

    # Delete the first link from the list
    def delete(self, goal, key= identity):
        if self.isEmpty():
            raise Exception('Cannot delete from an Empty list')
        previous = self                 # link or linkedlist before link
        while previous.getNext() is not None:
            link = previous.getNext()       # next link after previous
            if goal == key(link.getData()): # if link matches
                if link is self.__last:       # and was last link
                    self.__last = previous    # then move last back
                previous.setNext(link.getNext())    # change the previous next to be link's next
                return link.getData()
            previous = link             # advance previous to next link
        # since loop ends without a match
        raise Exception('No item with matching key was found')
```

Queue Implemented as LinkedList

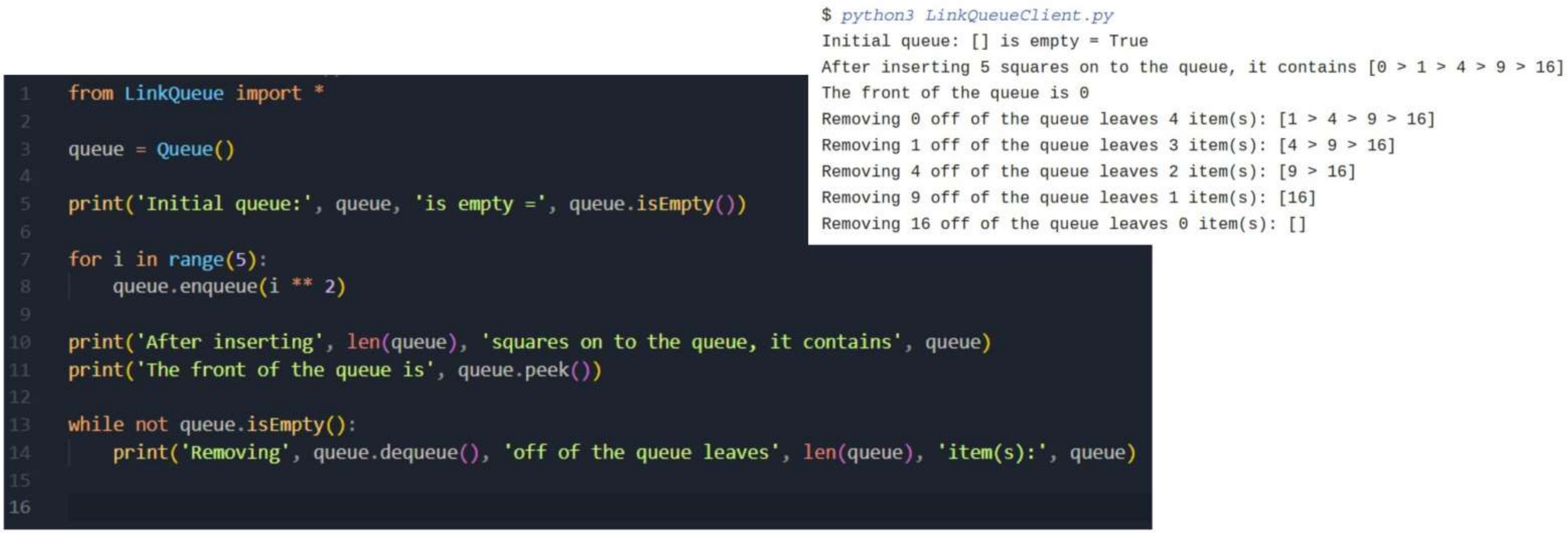
Queue Implemented as LinkedList

- Utilize a Double-Ended List.
- Then you don’t need access points to the double ends of the queue.



```python
from DoubleEndedList import *

class Queue(DoubleEndedList): # Define queue by renaming
    enqueue = DoubleEndedList.insertLast # Enqueue/insert at end
    dequeue = DoubleEndedList.deleteFirst # Dequeue/remove at first
    peek = DoubleEndedList.first # Front of queue is first
```

Queue Client



```
$ python3 LinkQueueClient.py
Initial queue: [] is empty = True
After inserting 5 squares on to the queue, it contains [0 > 1 > 4 > 9 > 16]
The front of the queue is 0
Removing 0 off of the queue leaves 4 item(s): [1 > 4 > 9 > 16]
Removing 1 off of the queue leaves 3 item(s): [4 > 9 > 16]
Removing 4 off of the queue leaves 2 item(s): [9 > 16]
Removing 9 off of the queue leaves 1 item(s): [16]
Removing 16 off of the queue leaves 0 item(s): []
```

```python
from LinkQueue import *

queue = Queue()

print('Initial queue:', queue, 'is empty =', queue.isEmpty())

for i in range(5):
    queue.enqueue(i ** 2)

print('After inserting', len(queue), 'squares on to the queue, it contains', queue)
print('The front of the queue is', queue.peek())

while not queue.isEmpty():
    print('Removing', queue.dequeue(), 'off of the queue leaves', len(queue), 'item(s):', queue)

```

Thanks