Lecture 17: Linked Lists

CS 61A - Summer 2024
Raymond Tan

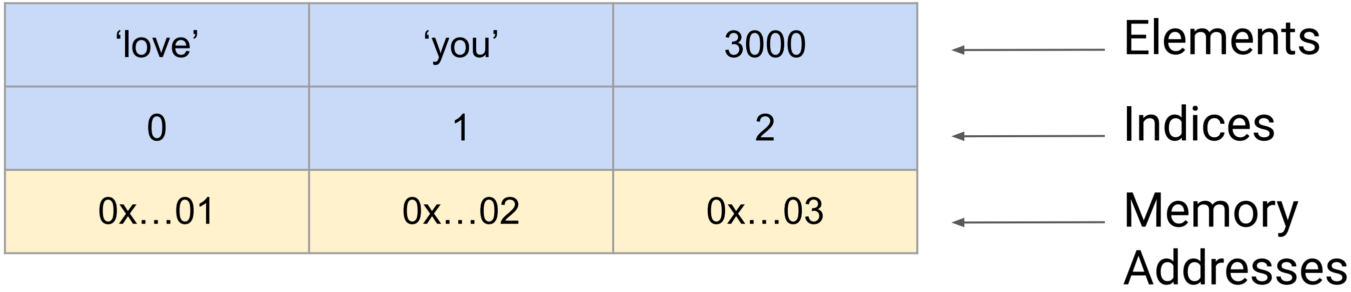
Linked Lists

Python Lists

- We’ve seen how we can use Python lists to store an **ordered sequence of elements**
- Each element stored in the list has an associated index and a particular *memory address* where that element is stored
    - When a list is created, a chunk of memory is allocated for that list
- Elements of the list are stored at consecutive *memory addresses*

Insertion into Python lists

- Consider the following list:
  - `lst = [‘love’, ‘you’, 3000]`
- The diagram for this list would look like:



‘love’	‘you’	3000	Elements
0	1	2	Indices
0x…01	0x…02	0x…03	Memory Addresses

Insertion into Python lists

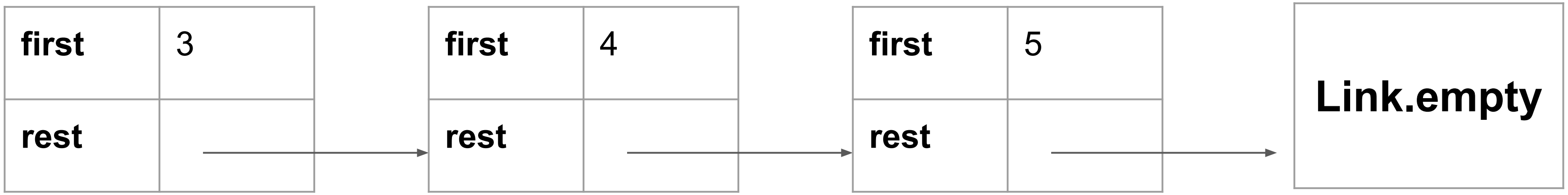
- What if we wanted to insert `’i’` at index 0 of our list?
  - `lst.insert(0, ‘i’)`
- What would our list diagram now look like?

‘i’	‘love’	‘you’	3000
0	1	2	3
0x…01	0x…02	0x…03	0x…04

All existing elements of our list had to shift over
Linear/O(N) runtime for inserting at the beginning of the list, where N is the number of elements in the list

Linked Lists

- A Linked List is either:
  - Empty
  - A `first` value and the `rest` of the Linked List



first 3
rest

first 4
rest

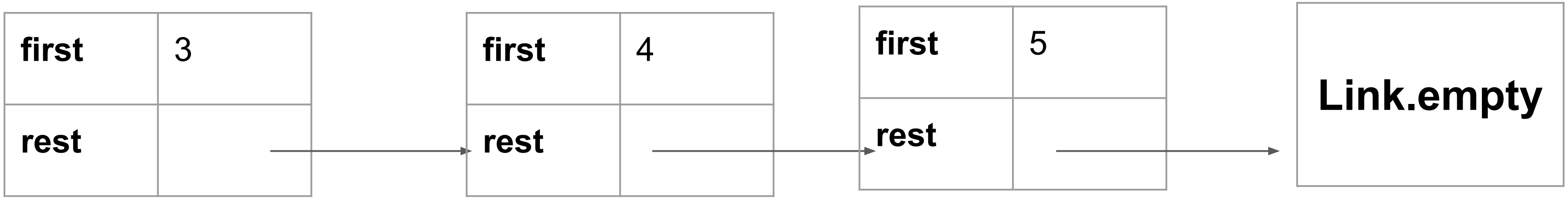
first 5
rest

Link.empty

A Linked List is made up of ***Link*** instances (objects) chained together, all the way until we hit Link.empty (denoting the end of the linked list)
Linked Lists are a recursive data structure

Linked List Insertion

- Insertion into linked lists is much more efficient
  - Just need to mutate rest pointer to add an element to a linked list
  - Each link object is stored in a different place in memory, so no need to shift all values that come after a link object



first 3
rest

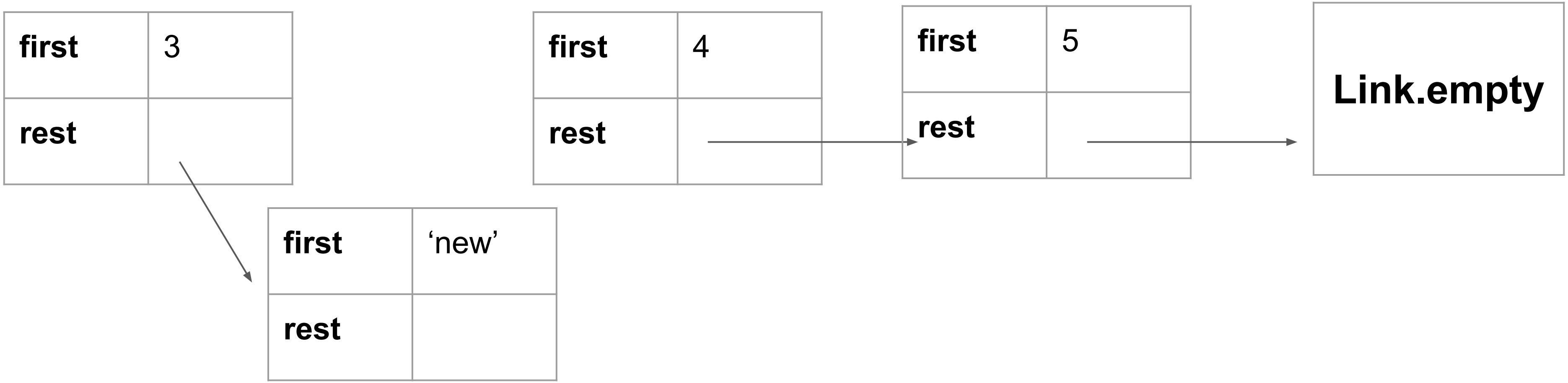
first 4
rest

first 5
rest

Link.empty

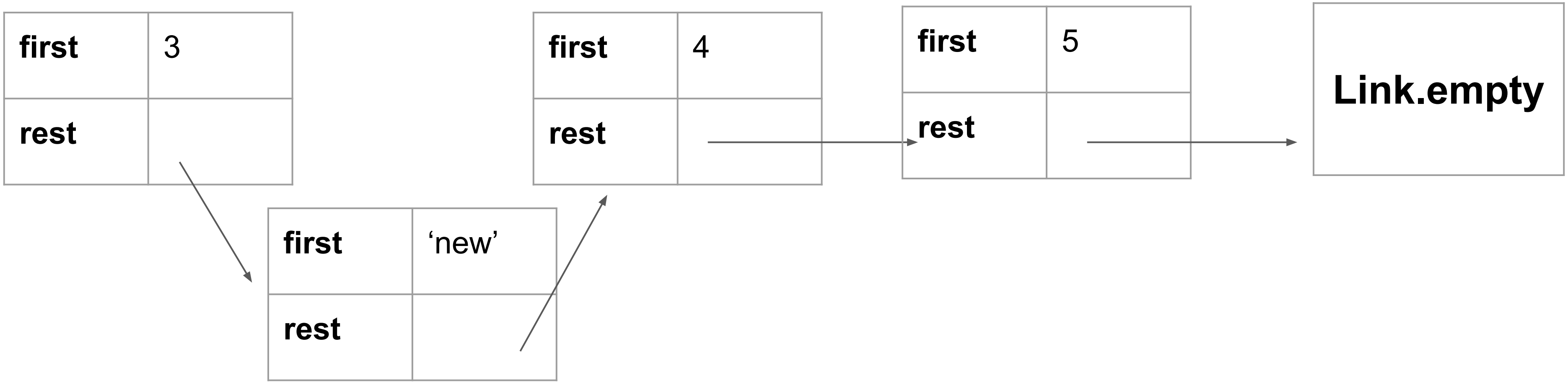
Linked List Insertion

- Insertion into linked lists is much more efficient
  - Just need to mutate rest pointer to add an element to a linked list
  - Each link object is stored in a different place in memory, so no need to shift all values that come after a link object

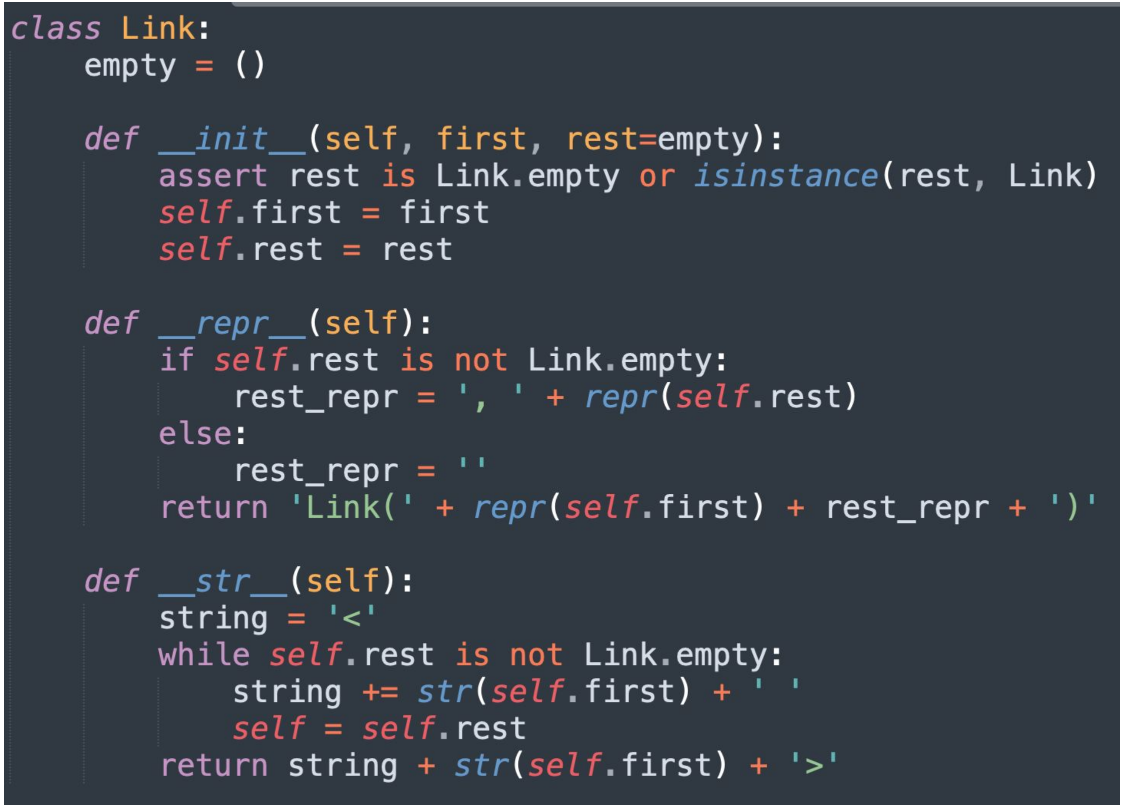


Linked List Insertion

- Insertion into linked lists is much more efficient
  - Just need to mutate rest pointer to add an element to a linked list
  - Each link object is stored in a different place in memory, so no need to shift all values that come after a link object



Linked List Python Class



```python
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

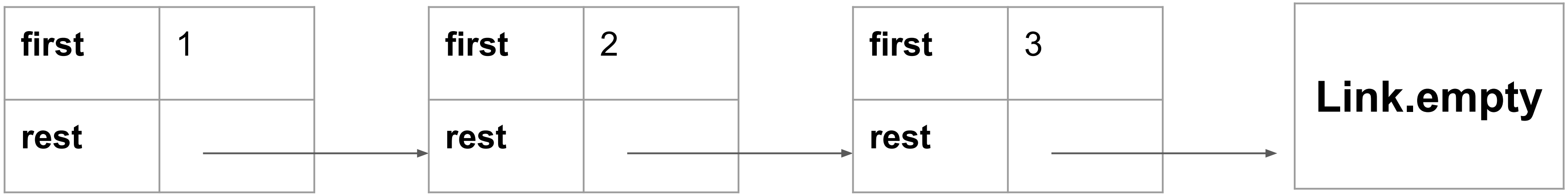
- Objects created from `Link` class
- The `init` method takes in two parameters – the `first`, representing the value stored at a node, and `rest`, representing a pointer to another Link object
  - ○ `rest` is set to `empty` if only one parameter is passed in to constructor
- `empty` is a class attribute of the Link class, which represents an empty Link object

Creating Linked Lists in Python

```
lnk = Link(1, Link(2, Link(3, Link.empty)))
```

or

```
lnk = Link(1, Link(2, Link(3)))
```



first 1

rest

first 2

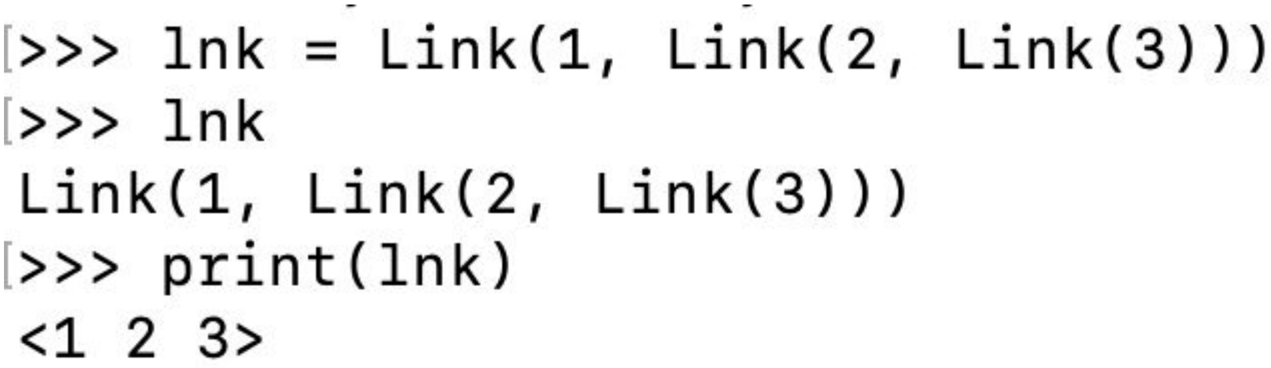
rest

first 3

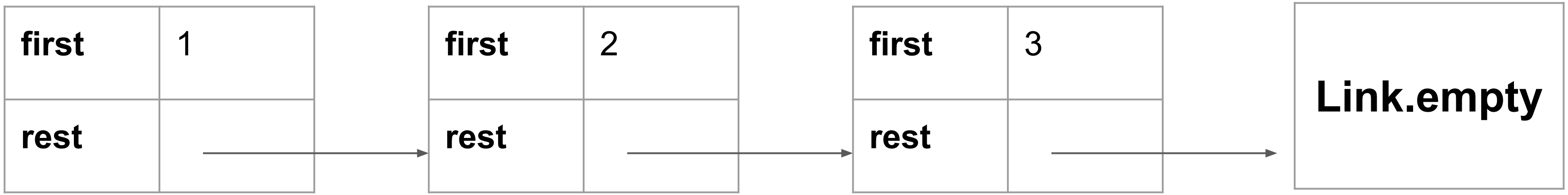
rest

Link.empty

Linked List String Representation

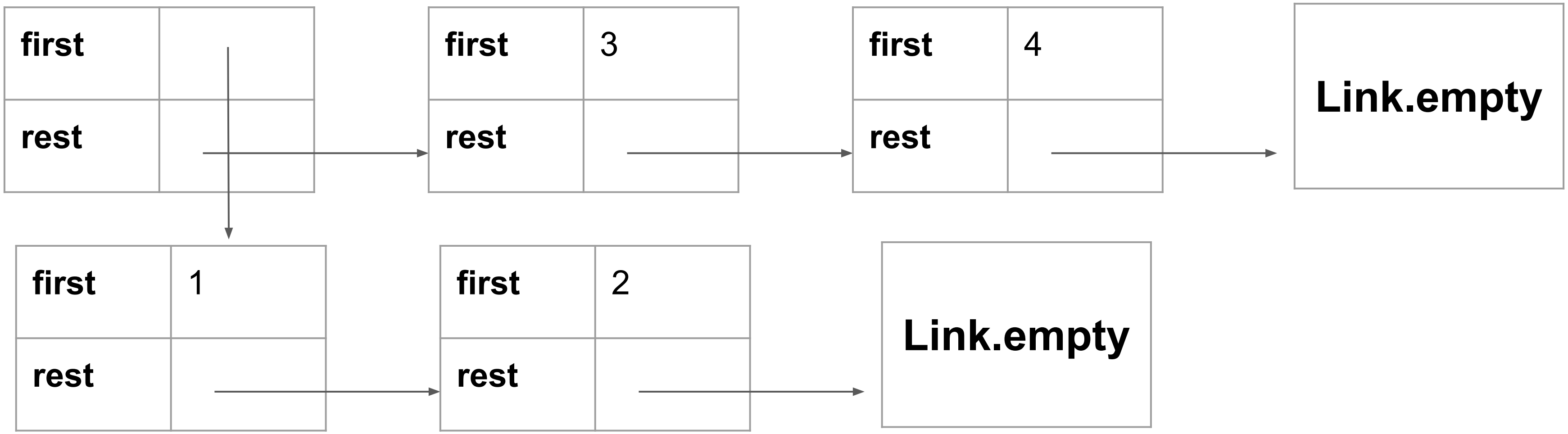


```
>>> lnk = Link(1, Link(2, Link(3)))
>>> lnk
Link(1, Link(2, Link(3)))
>>> print(lnk)
<1 2 3>
```



Deep/nested Linked Lists

- Similarly to how we can make deep/nested Python lists, we can do the same thing with Linked Lists
- Example list:
  - `lnk = Link(Link(1, Link(2)), Link(3, Link(4)))`



Comparing and Contrasting - Trees & Linked Lists

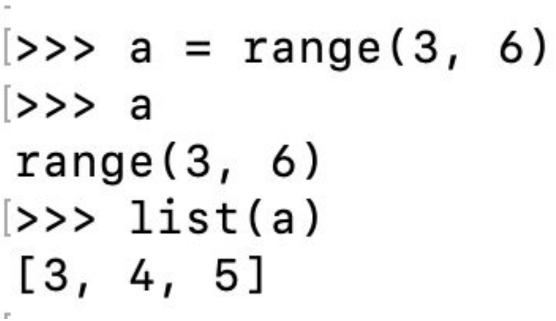
- Linked Lists are similar to Trees in many different aspects
  - Both are **recursive data structures** (defined in terms of itself)
  - Recursive solutions are common
- Differences:
  - Trees represent a hierarchical structure, while linked lists have a more ordered structure
  - Iterative solutions for linked lists are common
  - Trees have branches that point to multiple trees, while linked lists have a rest attribute which point to a singular linked list

Linked List Processing

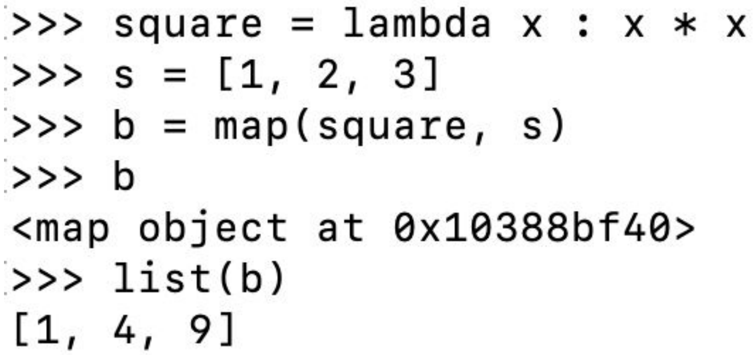
Demo: print_link

Review: range, map, filter

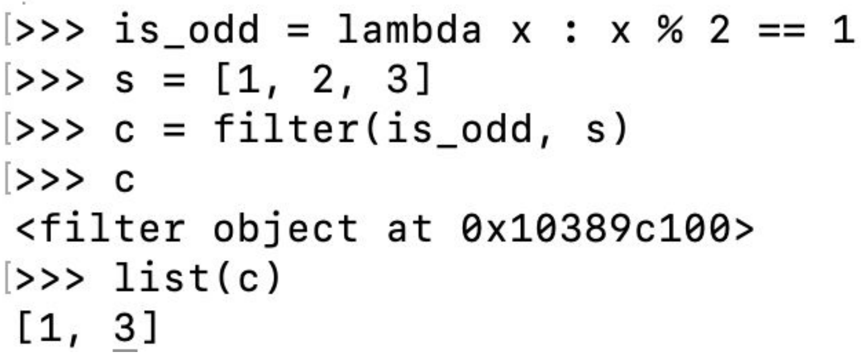
- `range`, `map`, and `filter` are built in functions that return iterables



```
>>> a = range(3, 6)
>>> a
range(3, 6)
>>> list(a)
[3, 4, 5]
```



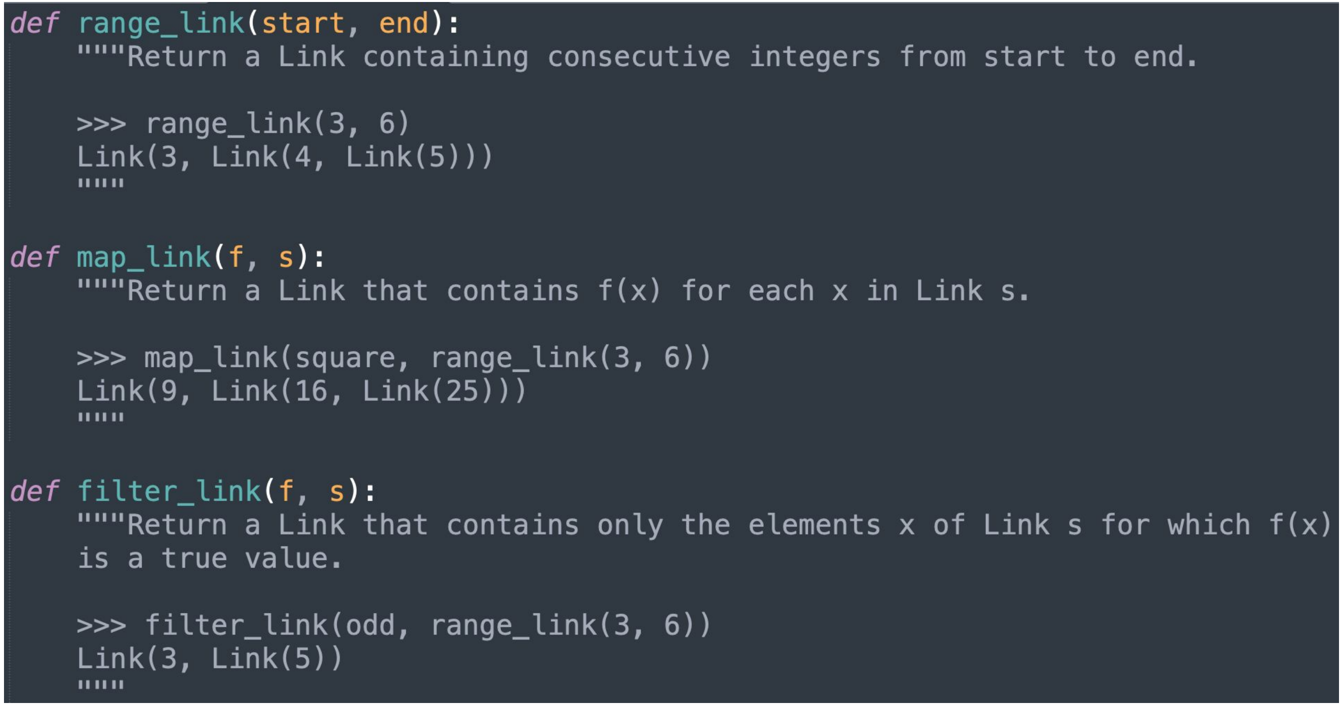
```
>>> square = lambda x : x * x
>>> s = [1, 2, 3]
>>> b = map(square, s)
>>> b
<map object at 0x10388bf40>
>>> list(b)
[1, 4, 9]
```



```
>>> is_odd = lambda x : x % 2 == 1
>>> s = [1, 2, 3]
>>> c = filter(is_odd, s)
>>> c
<filter object at 0x10389c100>
>>> list(c)
[1, 3]
```

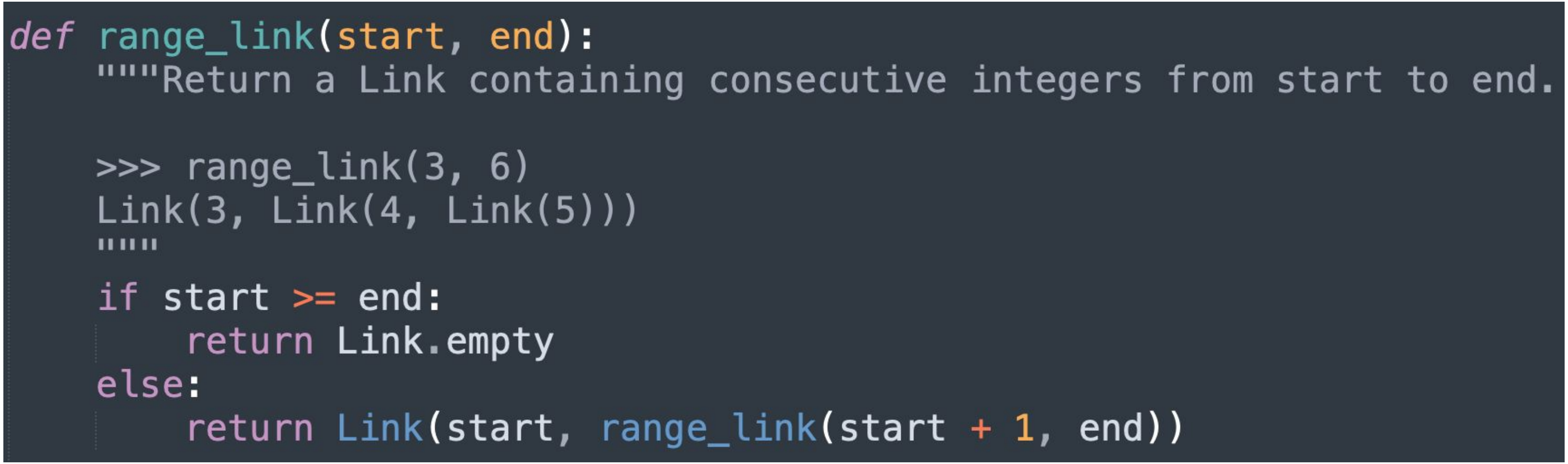
range, map, filter for Linked Lists

- Let’s now create similar functions that operate with/on Linked Lists



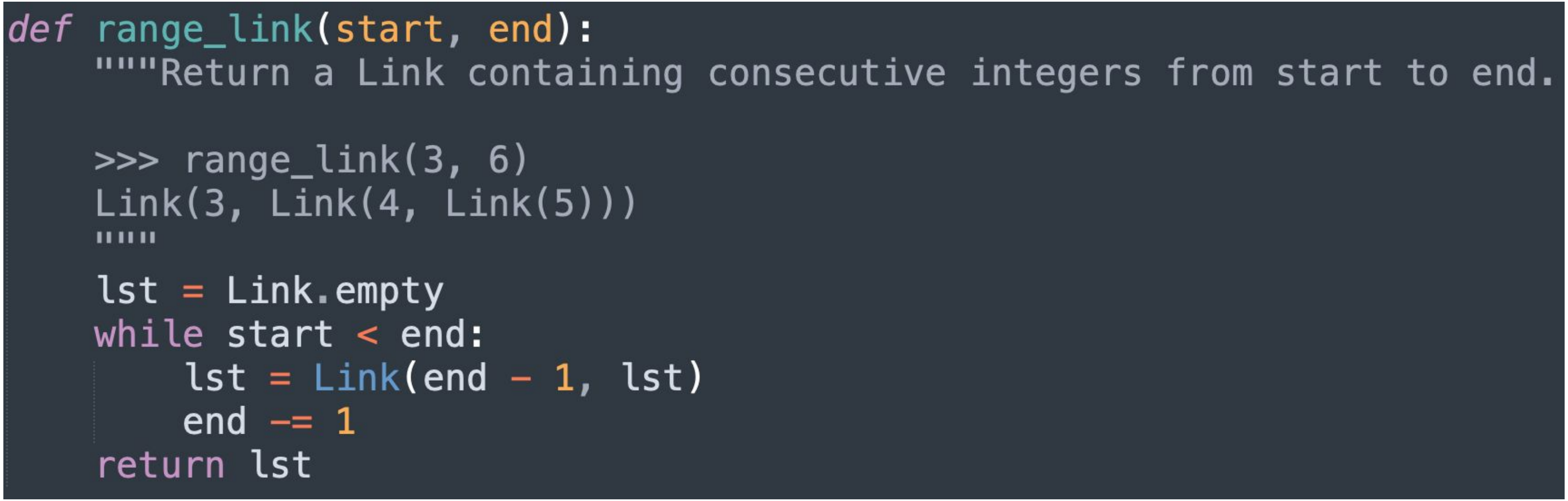
```
def range_link(start, end):
    """Return a Link containing consecutive integers from start to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """

def map_link(f, s):
    """Return a Link that contains f(x) for each x in Link s.

    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
    """

def filter_link(f, s):
    """Return a Link that contains only the elements x of Link s for which f(x)
    is a true value.

    >>> filter_link(odd, range_link(3, 6))
    Link(3, Link(5))
    """
```

range_link



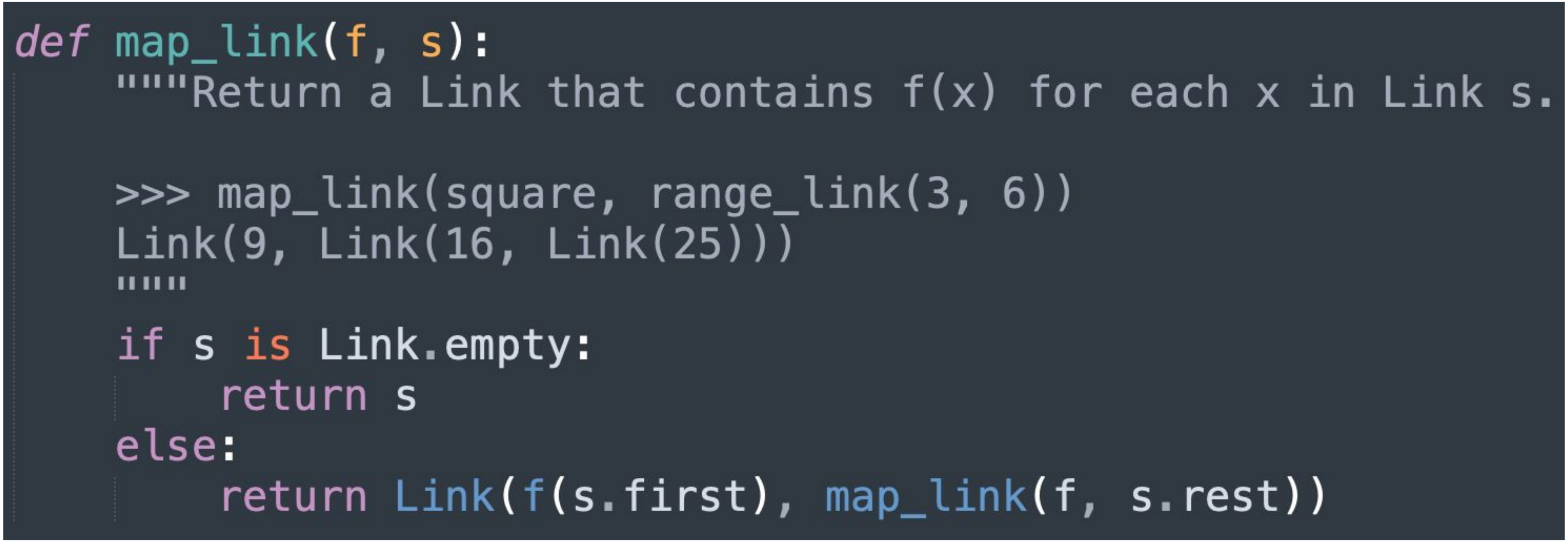
```python
def range_link(start, end):
    """Return a Link containing consecutive integers from start to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    if start >= end:
        return Link.empty
    else:
        return Link(start, range_link(start + 1, end))
```

range_link (iterative)



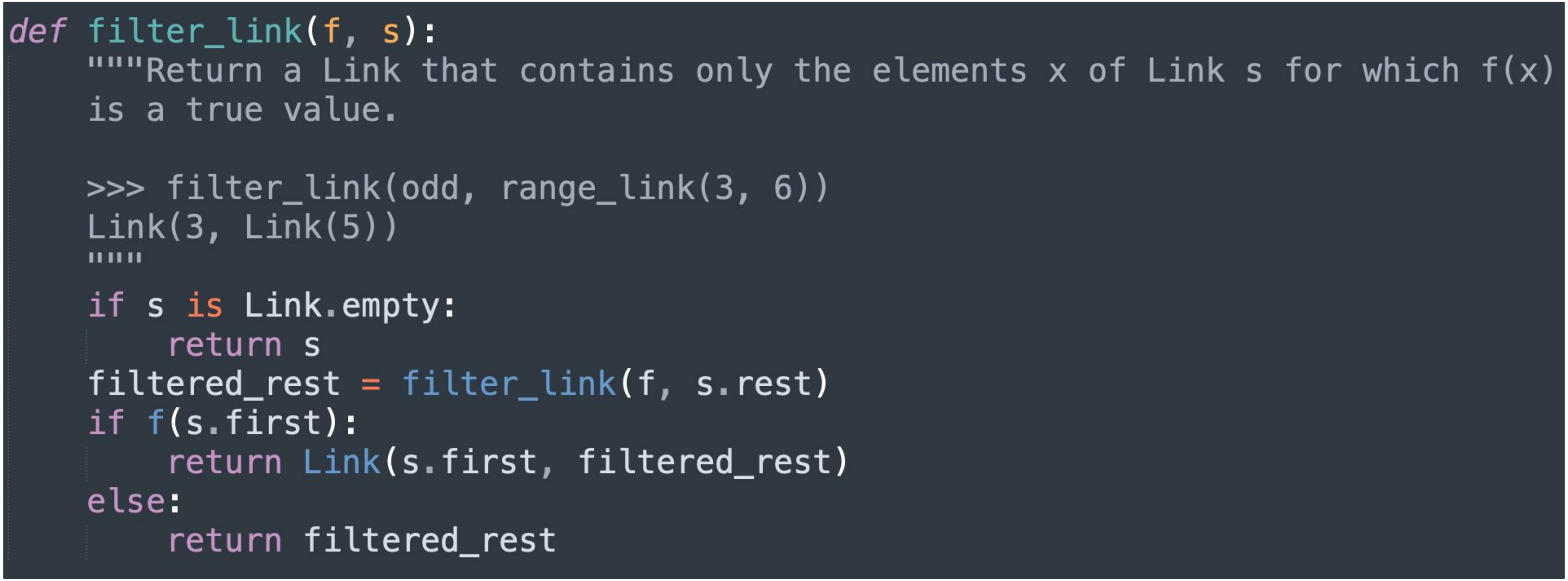
```python
def range_link(start, end):
    """Return a Link containing consecutive integers from start to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    lst = Link.empty
    while start < end:
        lst = Link(end - 1, lst)
        end -= 1
    return lst
```

map_link



```
def map_link(f, s):
    """Return a Link that contains f(x) for each x in Link s.

    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
    """
    if s is Link.empty:
        return s
    else:
        return Link(f(s.first), map_link(f, s.rest))
```

filter_link



```
def filter_link(f, s):
    """Return a Link that contains only the elements x of Link s for which f(x)
    is a true value.

    >>> filter_link(odd, range_link(3, 6))
    Link(3, Link(5))
    """
    if s is Link.empty:
        return s
    filtered_rest = filter_link(f, s.rest)
    if f(s.first):
        return Link(s.first, filtered_rest)
    else:
        return filtered_rest
```

Break

Linked List Mutation

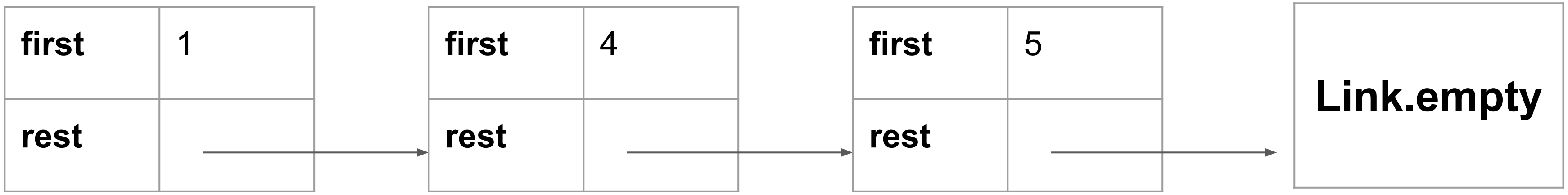
Linked List Mutation

- Linked Lists are instances of the Link class
    - Similar to instances of the Tree class, these objects are mutable!
- Let’s see how we can use mutation to insert into an existing linked list

Example: add_to_ordered_link

- Problem: Given an ***ordered*** linked list s, and a value x, mutate s to now have x, while **maintaining the ordered property**.

add_to_ordered_link: Visualization



first 1

rest

first 4

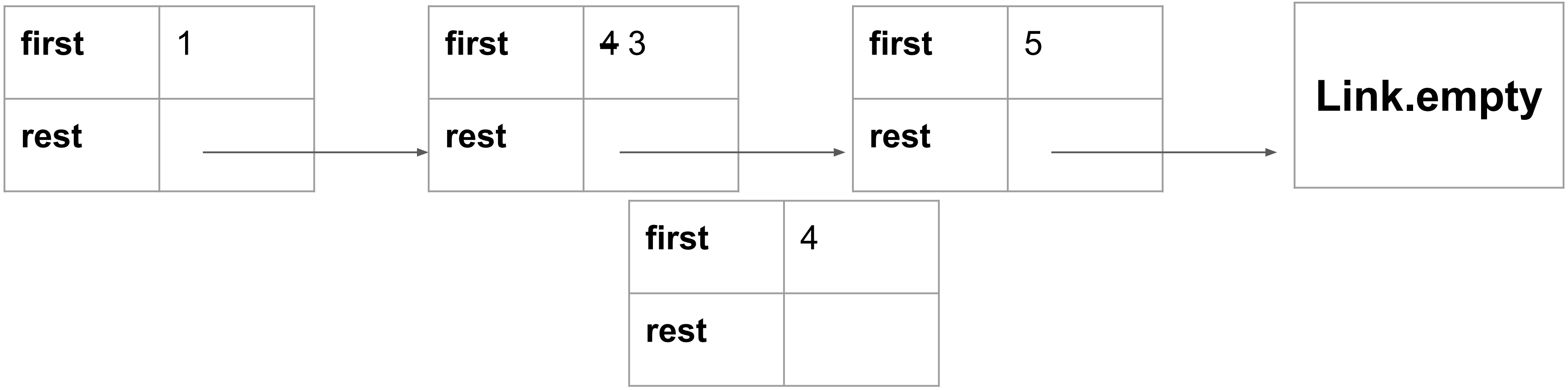
rest

first 5

rest

Link.empty

```
add_to_ordered_link(s, 3)
```



first 1

rest

first ~~4~~ 3

rest

first 5

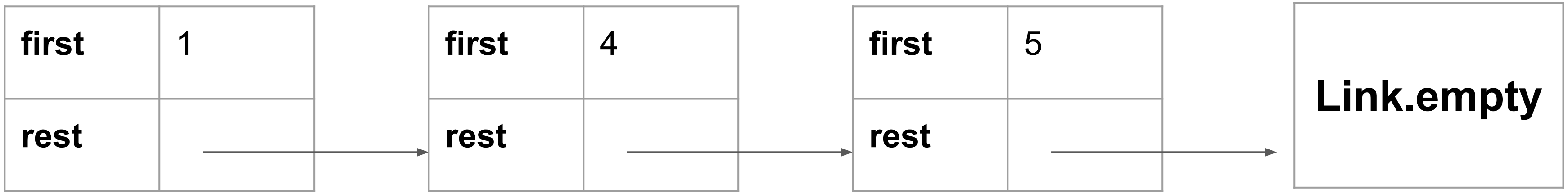
rest

Link.empty

first 4

rest

add_to_ordered_link: Visualization



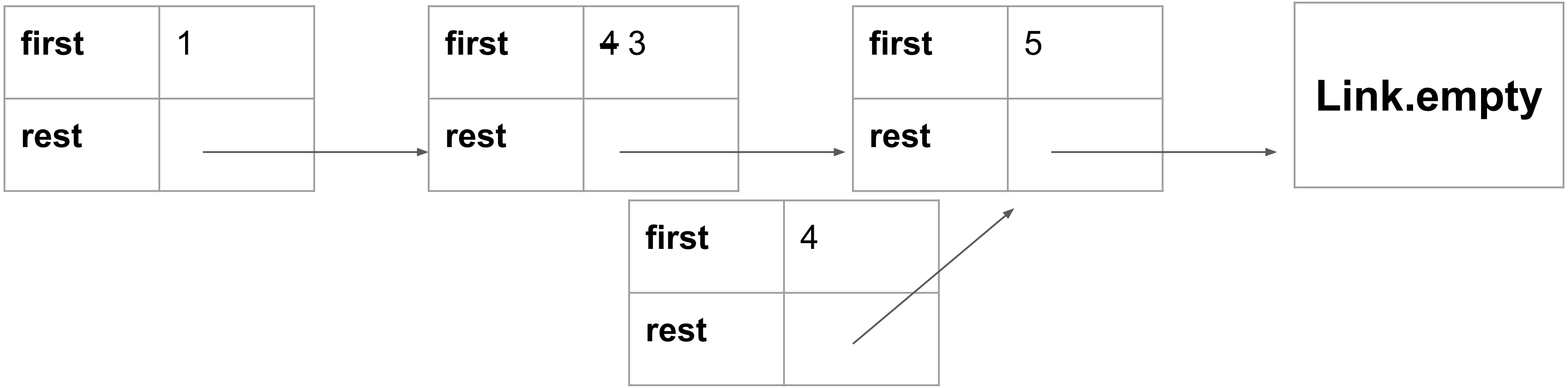
first 1
rest

first 4
rest

first 5
rest

Link.empty

```
add_to_ordered_link(s, 3)
```



first 1
rest

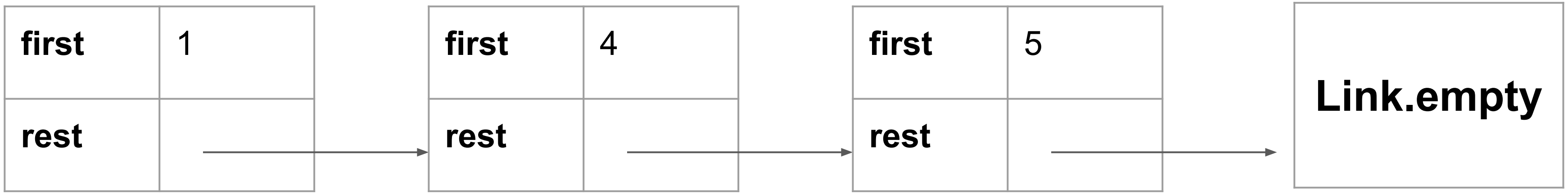
first ~~4~~ 3
rest

first 5
rest

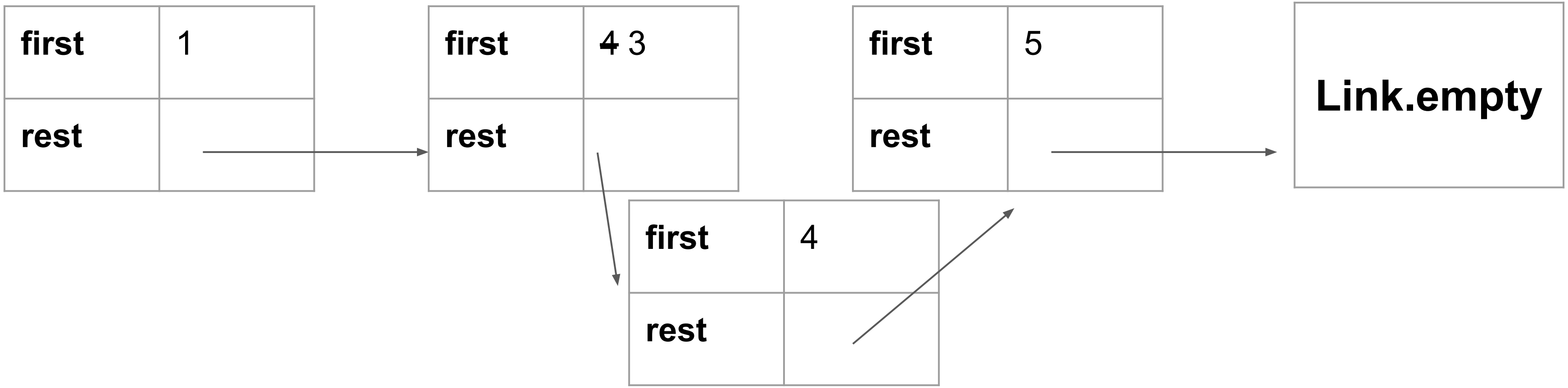
Link.empty

first 4
rest

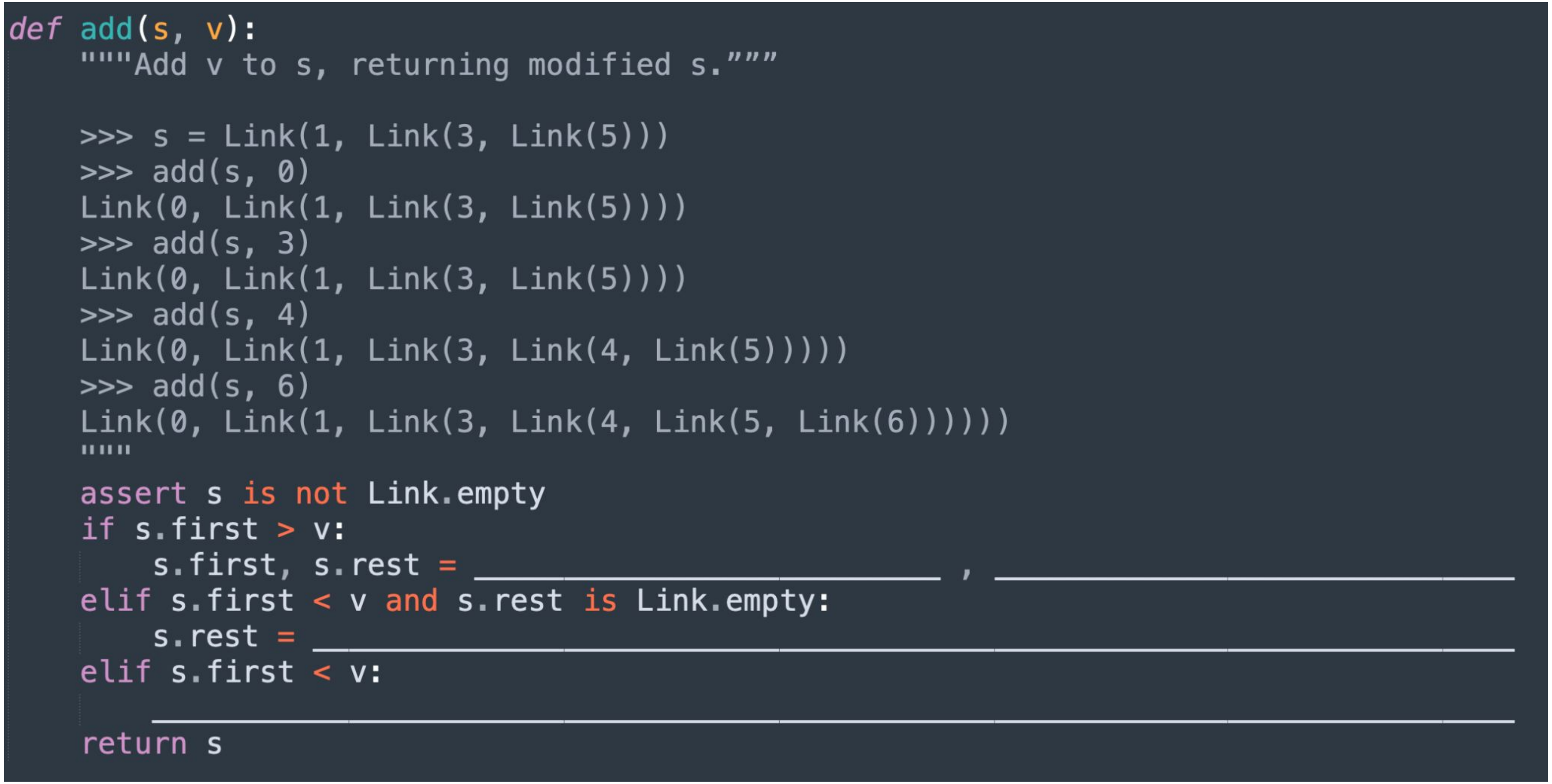
add_to_ordered_link: Visualization



`add_to_ordered_link(s, 3)`

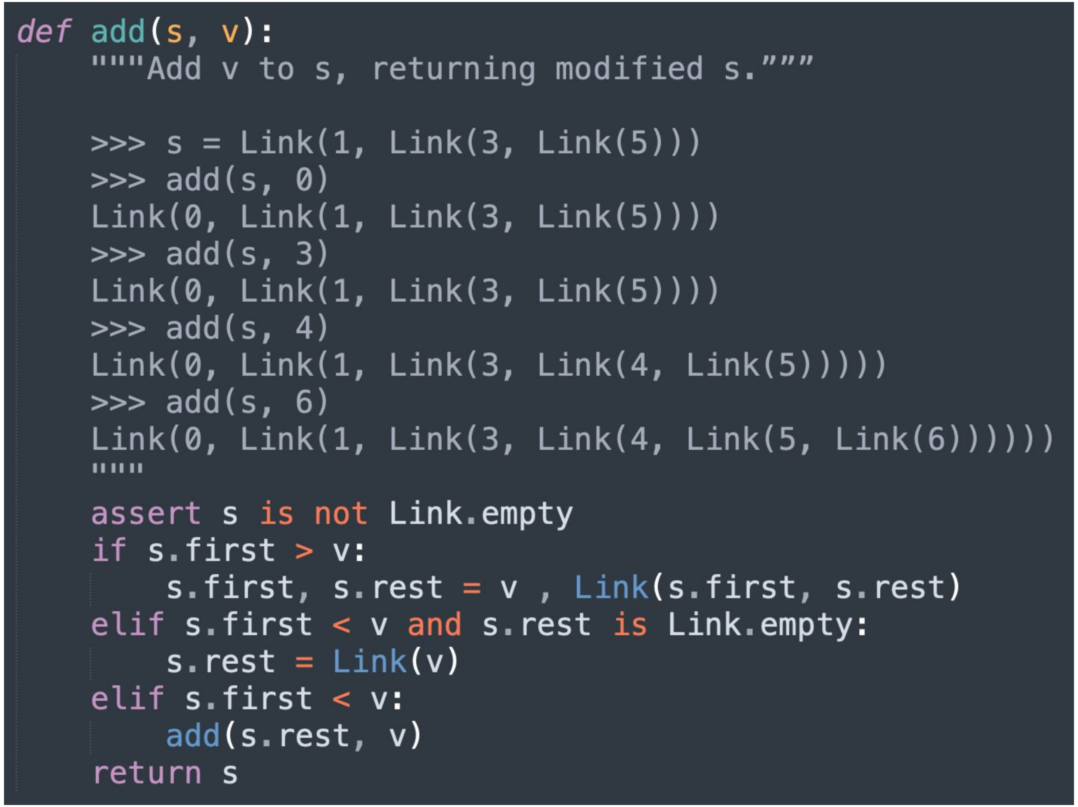


add_to_ordered_link: Code



```
def add(s, v):
    """Add v to s, returning modified s.”””

    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    assert s is not Link.empty
    if s.first > v:
        s.first, s.rest = __________________________ , ______________________________
    elif s.first < v and s.rest is Link.empty:
        s.rest = ___________________________________________________________________
    elif s.first < v:
        ____________________________________________________________________________
    return s
```

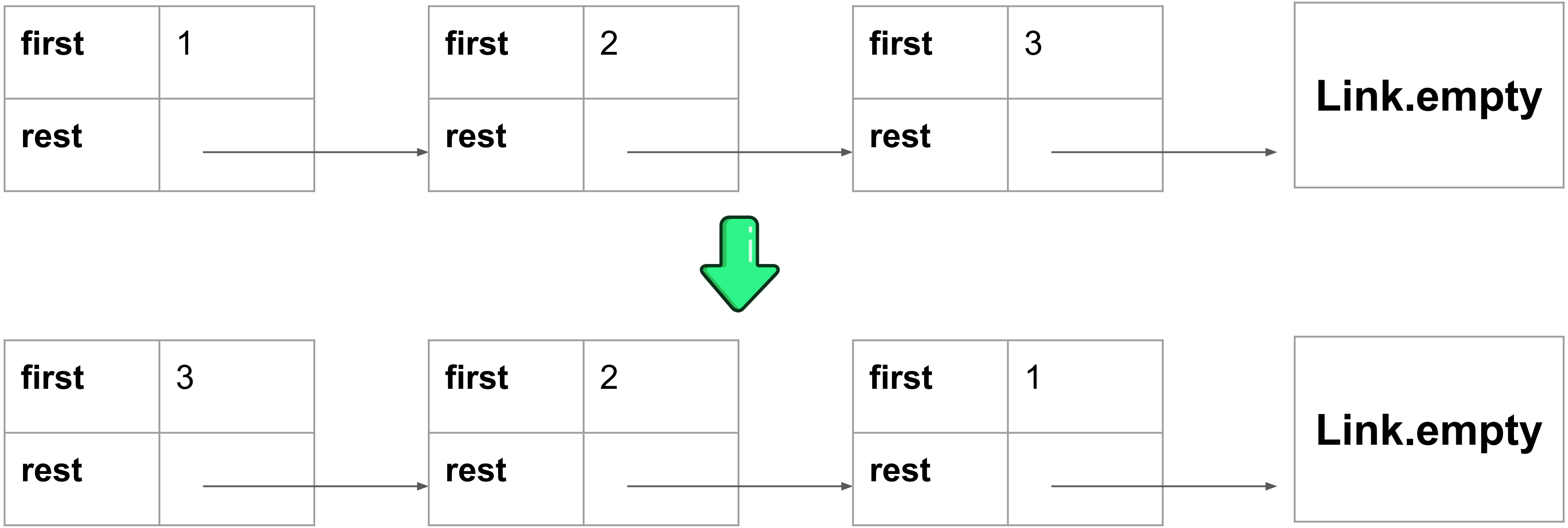
add_to_ordered_link: Solution



```
def add(s, v):
    """Add v to s, returning modified s.”””

    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    assert s is not Link.empty
    if s.first > v:
        s.first, s.rest = v , Link(s.first, s.rest)
    elif s.first < v and s.rest is Link.empty:
        s.rest = Link(v)
    elif s.first < v:
        add(s.rest, v)
    return s
```

Example: reverse_link

- Problem: Given a linked list s, return a reversed version of s.



first 1
rest

first 2
rest

first 3
rest

Link.empty

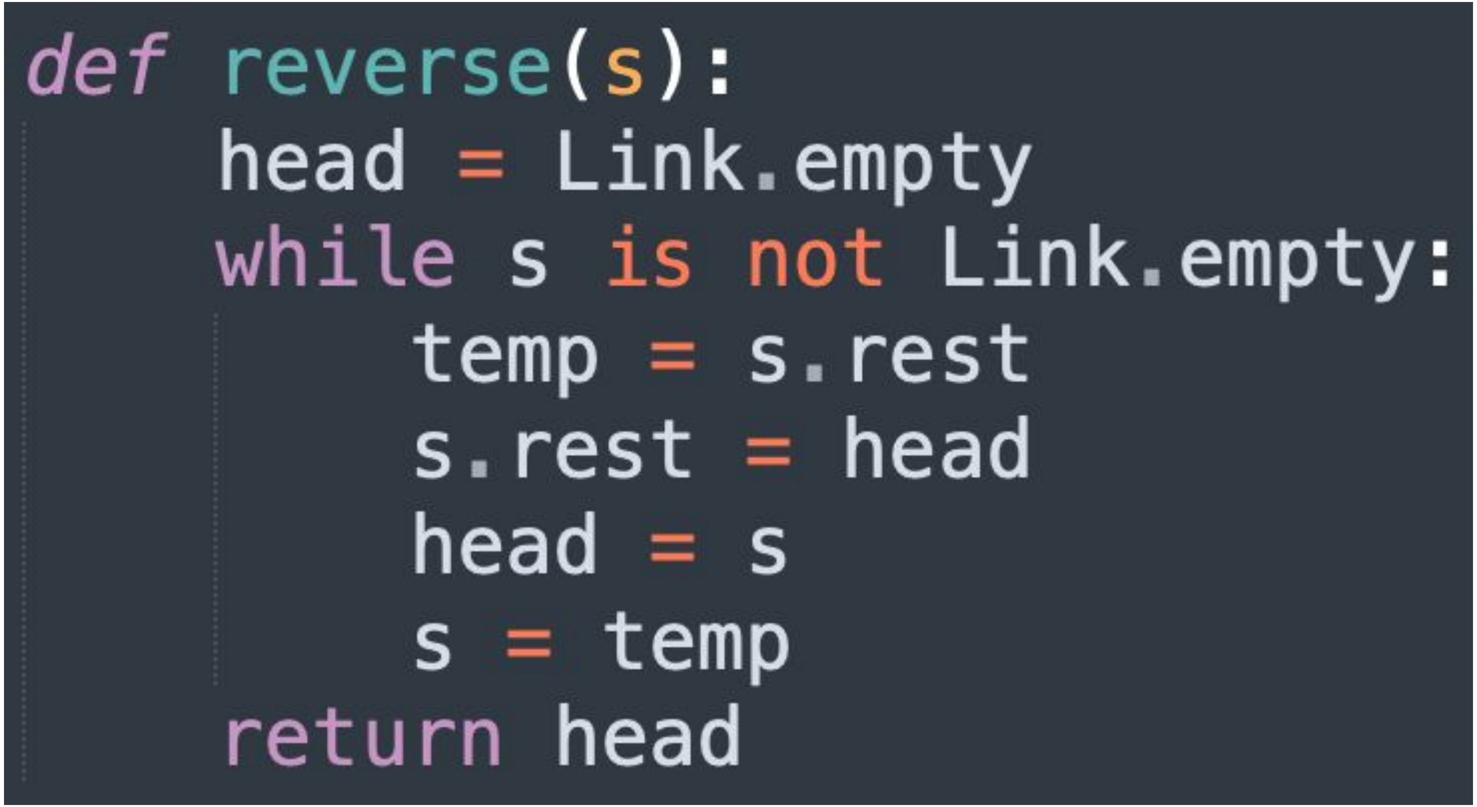
first 3
rest

first 2
rest

first 1
rest

Link.empty

reverse_link: Solution



```
def reverse(s):
    head = Link.empty
    while s is not Link.empty:
        temp = s.rest
        s.rest = head
        head = s
        s = temp
    return head
```

Summary

- Linked Lists are a new recursive data structure that can store an ordered sequence
    - Much more efficient at insertion than a Python list
- Mutable, since they are instances of the Link class
- Solvable using both recursion and iteration