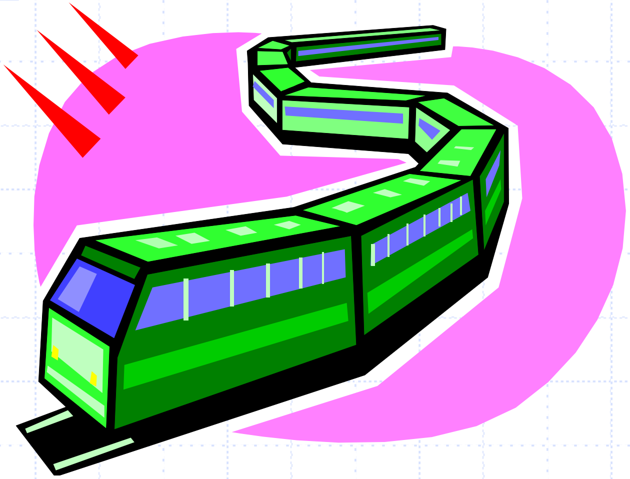
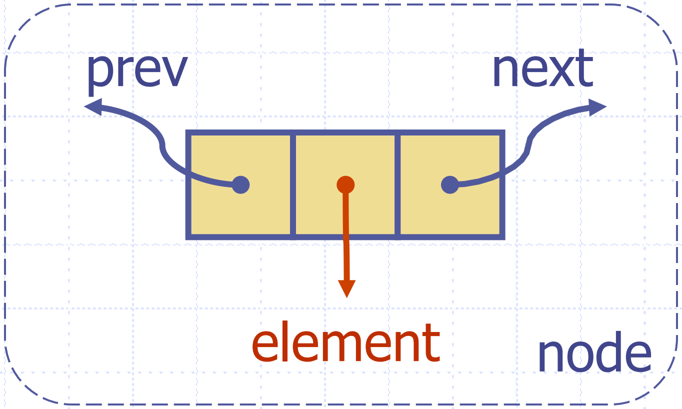
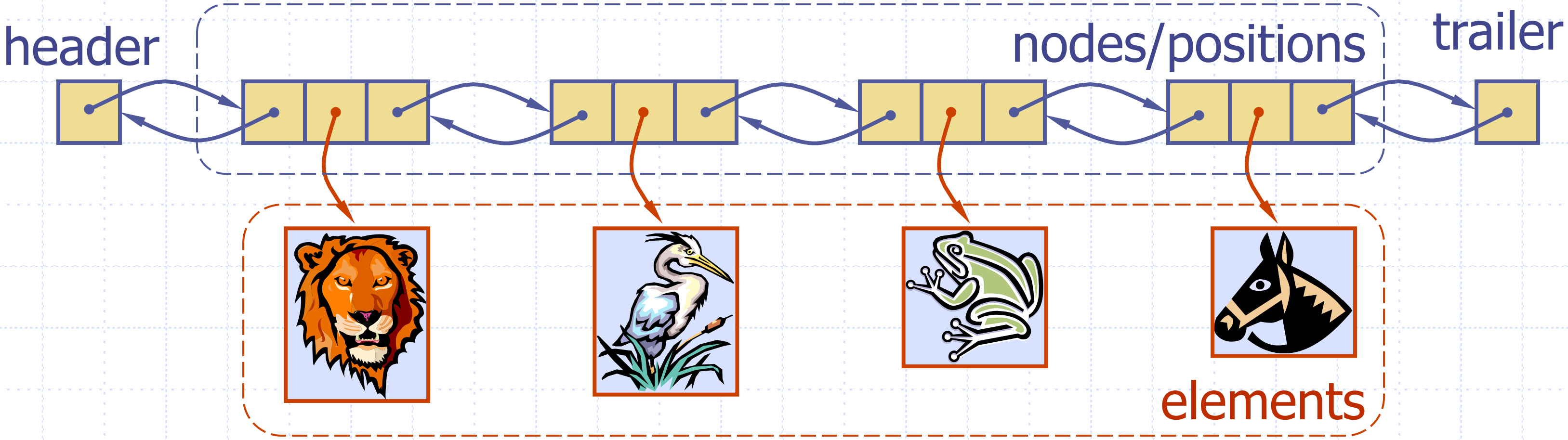
Doubly-Linked Lists



Doubly Linked List

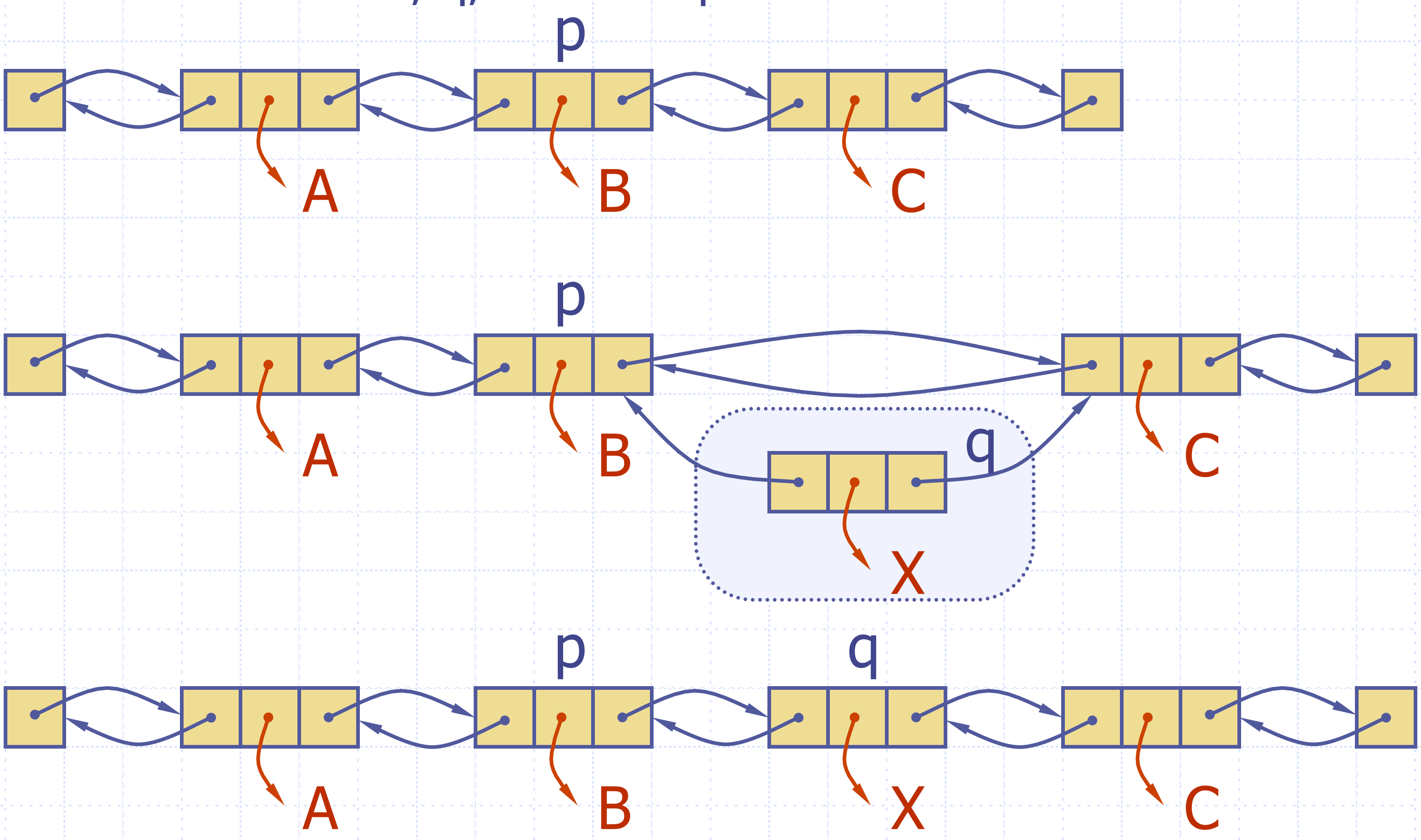
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





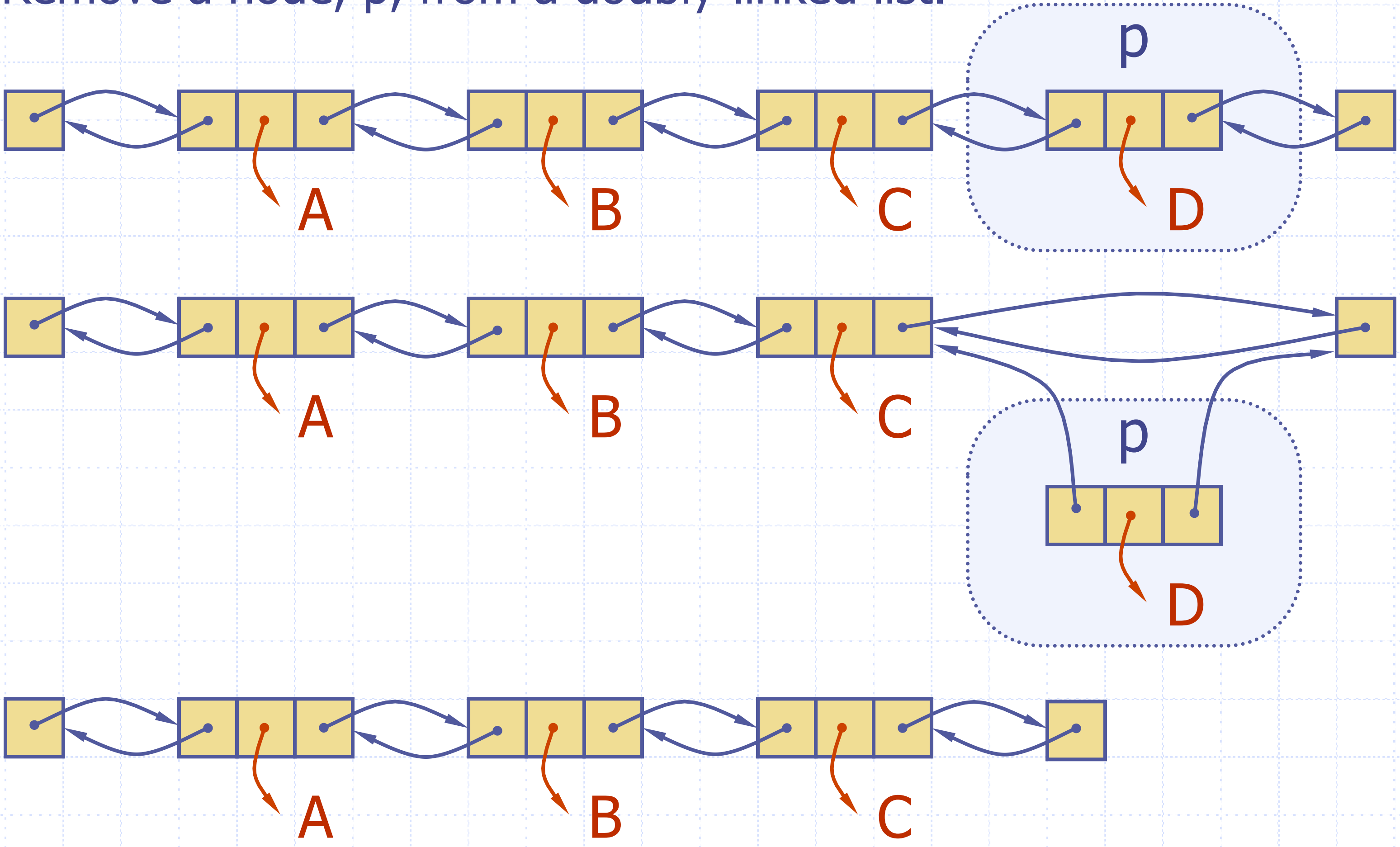
Insertion

- Insert a new node, q, between p and its successor.



Deletion

- Remove a node, p, from a doubly-linked list.



Doubly-Linked Node

```
class DoublyLinkedNode:

    def __init__( self, element, prev, next ):
        self.element = element
        self.prev = prev
        self.next = next
```

Doubly-Linked List in Python…

```python
from DoublyLinkedNode import DoublyLinkedNode

class DoublyLinkedList:

    #implements the ADT List (List.py)
    #uses the DoublyLinkedNode class (DoublyLinkedNode.py)

    def __init__( self ):
        self._head = DoublyLinkedNode( None, None, None )
        self._trail = DoublyLinkedNode( None, None, None )
        self._head.next = self._trail
        self._trail.prev = self._head
        self._size = 0
```

Doubly-Linked List in Python…

```
def __len__( self ):
     return self._size

def __str__( self ):
    if self.is_empty():
        return "[](size = 0)"
    else:
        pp = "["
        curr = self._head.next
        while curr.next != self._trail:
            pp += str( curr.element ) + ", "
         curr = curr.next
        pp += str( curr.element ) + "]"
        pp += "(size = " + str( self._size ) + ")"
    return pp
```

Doubly-Linked List in Python…

```
def is_empty( self ):
    return self._size == 0

def append( self, element ):
    newNode = DoublyLinkedNode( element, self._trail.prev, self._trail )
    self._trail.prev.next = newNode
    self._trail.prev = newNode
    self._size += 1

def insert( self, element ):
    newNode = DoublyLinkedNode( element, self._head, self._head.next )
    self._head.next.prev = newNode
    self._head.next = newNode
    self._size += 1
```

Doubly-Linked List in Python…

```
def remove( self, k ):
    if self.is_empty():
        return False
    elif k <= 0 or k > self._size:
        return False
    else:
        curr = self._head.next
        for i in range( k - 1 ):
            curr = curr.next
        curr.prev.next = curr.next
        curr.next.prev = curr.prev
        self._size -= 1
        return curr.element
```

Doubly-Linked List in Python…

```
def find( self, element ):
    if self.is_empty():
        return False
    else:
        curr = self._head.next
        for i in range( self._size ):
            if curr.element == element:
                return i + 1
            else:
               curr = curr.next
```

Doubly-Linked List in Python

```
def last( self ):
    if self.is_empty():
        return False
    else:
        return self._trail.prev.element

def first( self ):
    if self.is_empty():
        return False
    else:
        return self._head.next.element
```

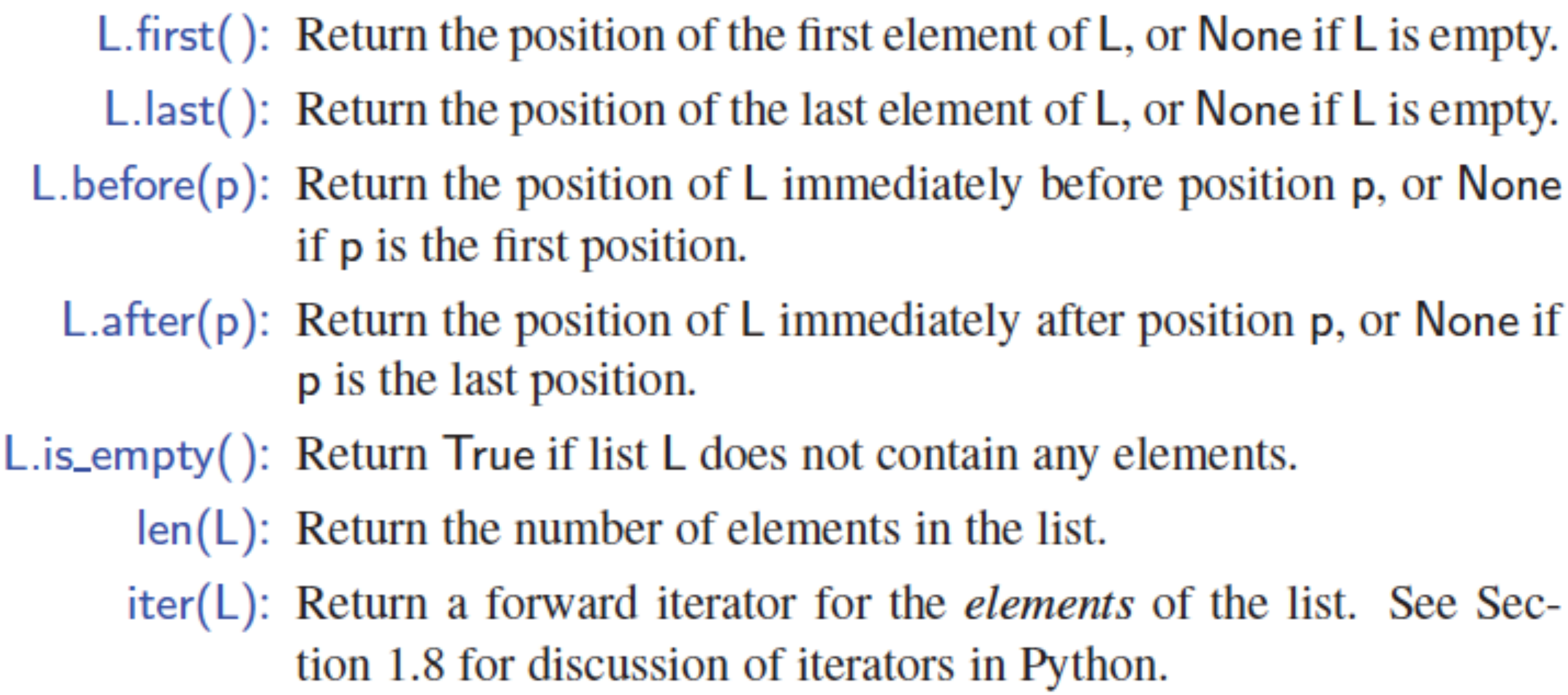
Performance

- In a doubly linked list
  - The space used by a list with $\boldsymbol{n}$ elements is $\boldsymbol{O}(\boldsymbol{n})$
  - The space used by each position of the list is $\boldsymbol{O}(1)$
  - insert and append runs in $\boldsymbol{O}(1)$ time
  - find and remove runs in $\boldsymbol{O}(\mathrm{n})$ time
  - last and first runs in $\boldsymbol{O}(1)$ time

Positional List

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
    - p.element( ): Return the element stored at position p.

Positional Accessor Operations



L.first( ): Return the position of the first element of L, or None if L is empty.

L.last( ): Return the position of the last element of L, or None if L is empty.

L.before(p): Return the position of L immediately before position p, or None if p is the first position.

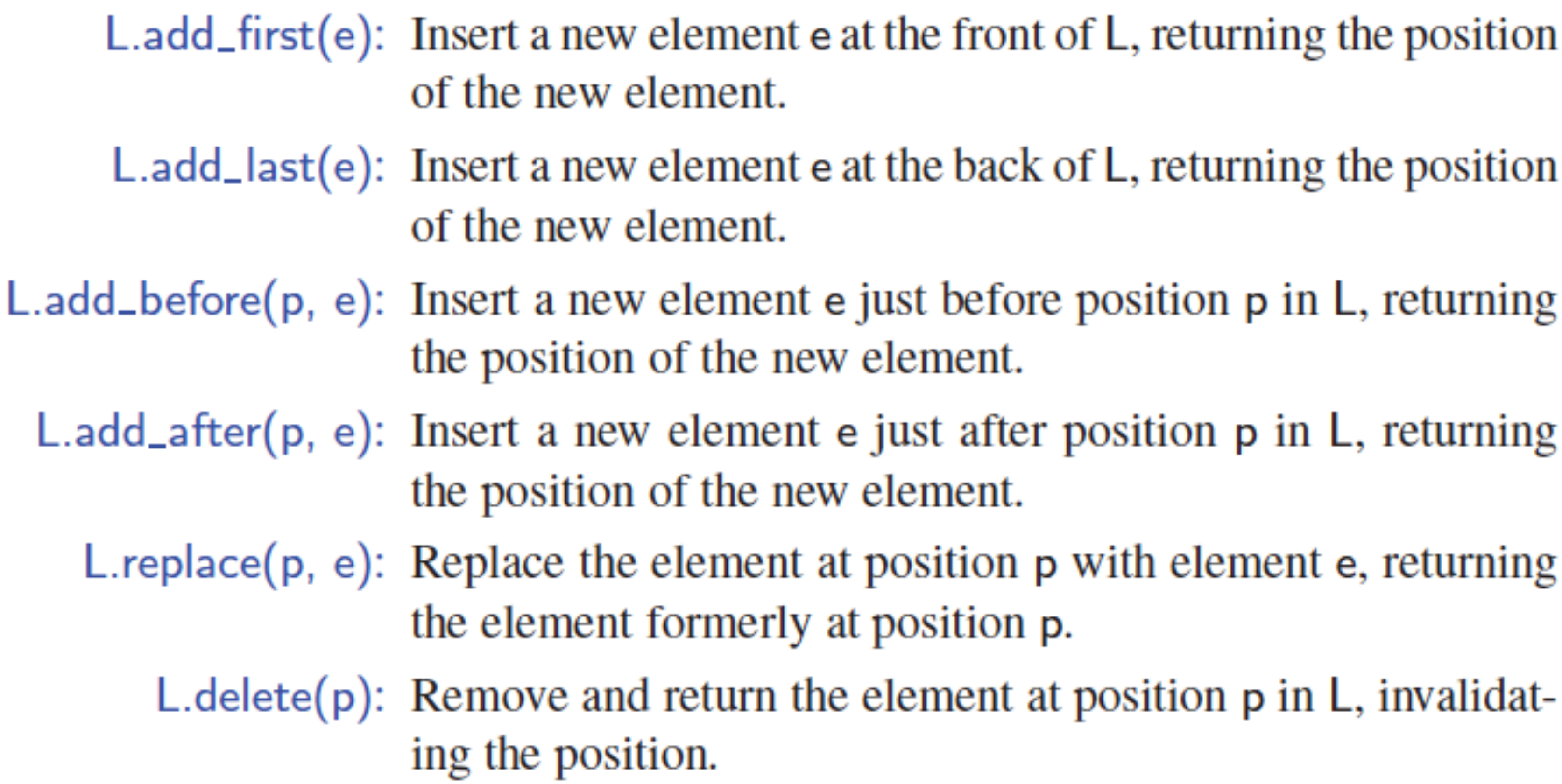
L.after(p): Return the position of L immediately after position p, or None if p is the last position.

L.is_empty( ): Return True if list L does not contain any elements.

len(L): Return the number of elements in the list.

iter(L): Return a forward iterator for the *elements* of the list. See Section 1.8 for discussion of iterators in Python.

Positional Update Operations



L.add_first(e): Insert a new element e at the front of L, returning the position of the new element.

L.add_last(e): Insert a new element e at the back of L, returning the position of the new element.

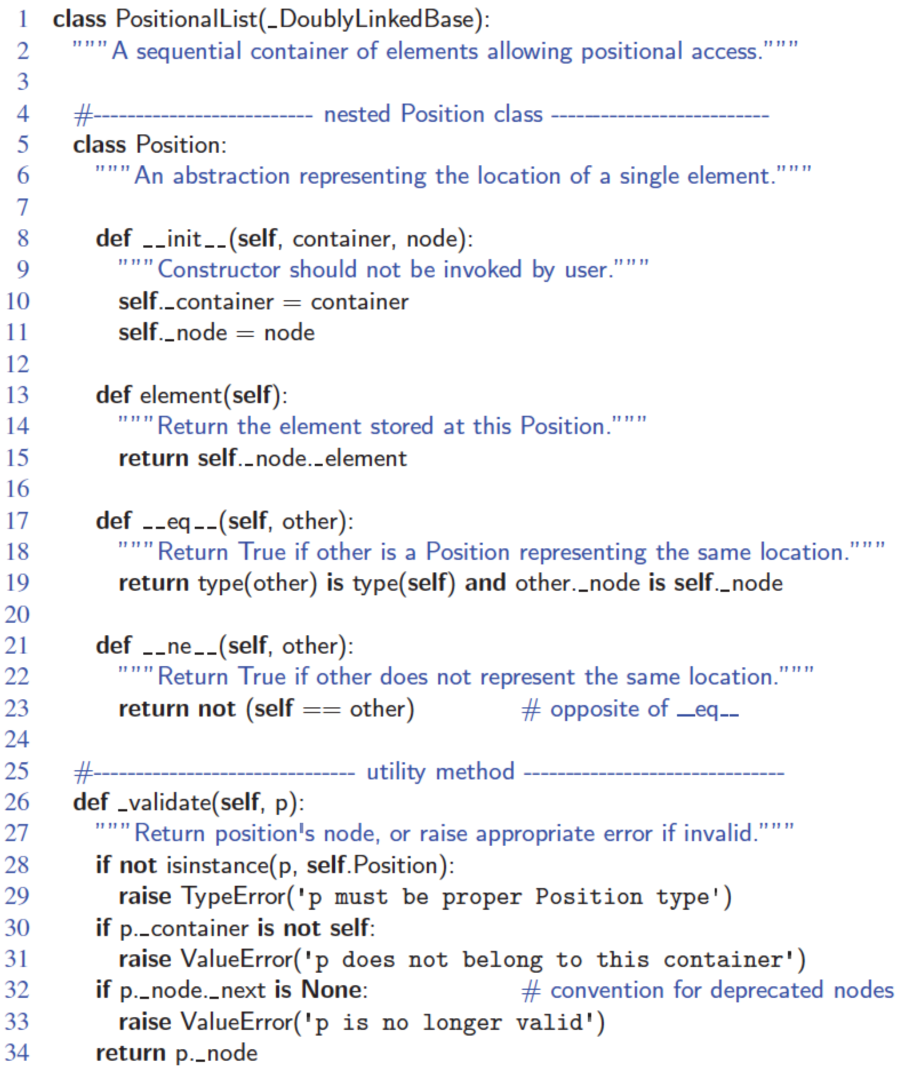
L.add_before(p, e): Insert a new element e just before position p in L, returning the position of the new element.

L.add_after(p, e): Insert a new element e just after position p in L, returning the position of the new element.

L.replace(p, e): Replace the element at position p with element e, returning the element formerly at position p.

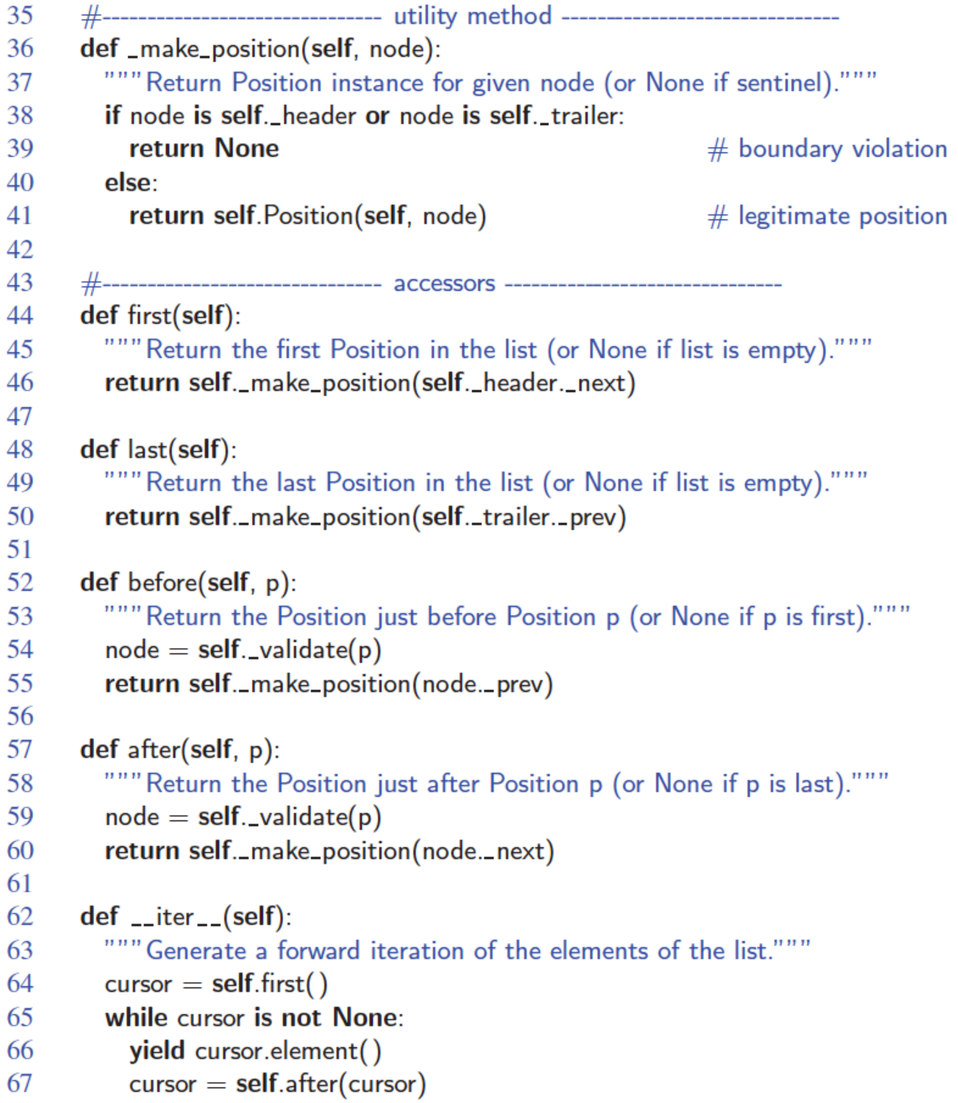
L.delete(p): Remove and return the element at position p in L, invalidating the position.

Positional List in Python



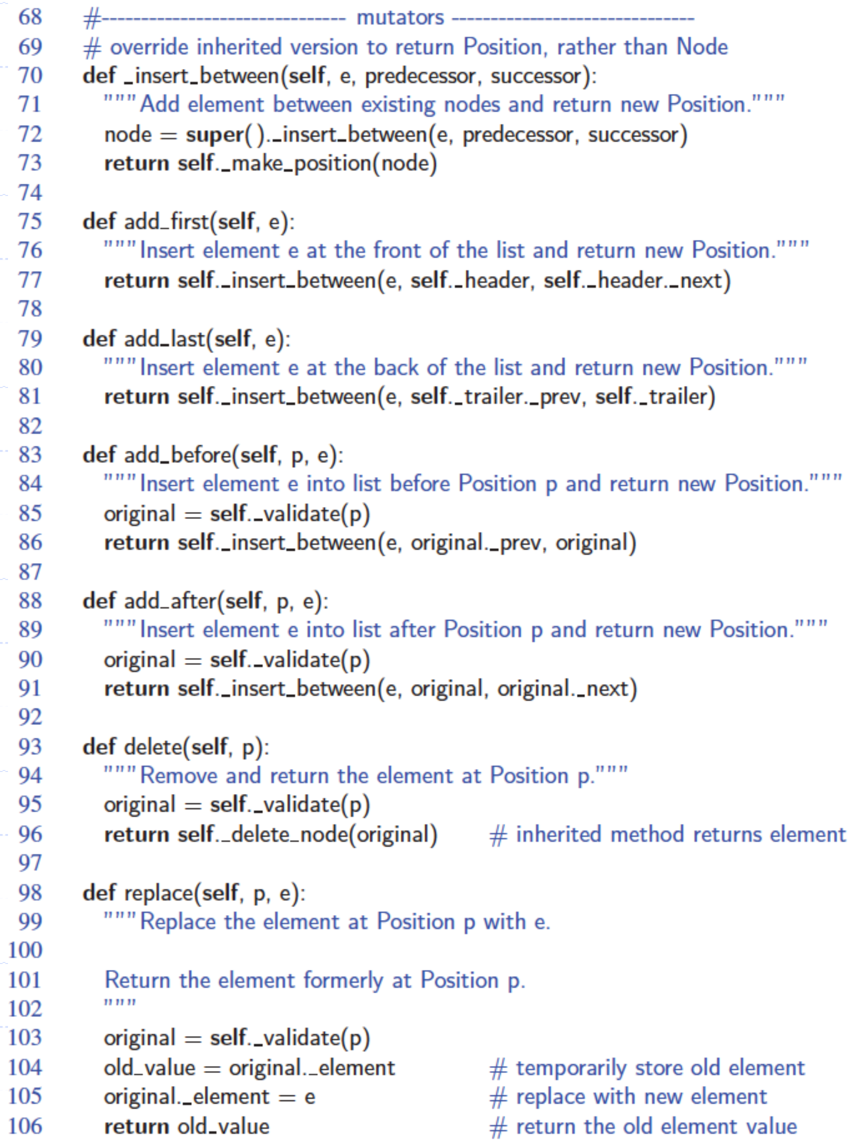
```
class PositionalList(_DoublyLinkedBase):
  """A sequential container of elements allowing positional access."""

  #------------------------- nested Position class -------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def __init__(self, container, node):
      """Constructor should not be invoked by user."""
      self._container = container
      self._node = node

    def element(self):
      """Return the element stored at this Position."""
      return self._node._element

    def __eq__(self, other):
      """Return True if other is a Position representing the same location."""
      return type(other) is type(self) and other._node is self._node

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)            # opposite of __eq__

  #------------------------------- utility method -------------------------------
  def _validate(self, p):
    """Return position's node, or raise appropriate error if invalid."""
    if not isinstance(p, self.Position):
      raise TypeError('p must be proper Position type')
    if p._container is not self:
      raise ValueError('p does not belong to this container')
    if p._node._next is None:               # convention for deprecated nodes
      raise ValueError('p is no longer valid')
    return p._node
```

Positional List in Python, Part 2



```
#------------------------------ utility method ------------------------------
def _make_position(self, node):
  """Return Position instance for given node (or None if sentinel)."""
  if node is self._header or node is self._trailer:
    return None                                   # boundary violation
  else:
    return self.Position(self, node)              # legitimate position

#------------------------------ accessors ------------------------------
def first(self):
  """Return the first Position in the list (or None if list is empty)."""
  return self._make_position(self._header._next)

def last(self):
  """Return the last Position in the list (or None if list is empty)."""
  return self._make_position(self._trailer._prev)

def before(self, p):
  """Return the Position just before Position p (or None if p is first)."""
  node = self._validate(p)
  return self._make_position(node._prev)

def after(self, p):
  """Return the Position just after Position p (or None if p is last)."""
  node = self._validate(p)
  return self._make_position(node._next)

def __iter__(self):
  """Generate a forward iteration of the elements of the list."""
  cursor = self.first()
  while cursor is not None:
    yield cursor.element()
    cursor = self.after(cursor)
```

Positional List in Python, Part 3



```
#------------------------------ mutators ------------------------------
# override inherited version to return Position, rather than Node
def _insert_between(self, e, predecessor, successor):
  """Add element between existing nodes and return new Position."""
  node = super()._insert_between(e, predecessor, successor)
  return self._make_position(node)

def add_first(self, e):
  """Insert element e at the front of the list and return new Position."""
  return self._insert_between(e, self._header, self._header._next)

def add_last(self, e):
  """Insert element e at the back of the list and return new Position."""
  return self._insert_between(e, self._trailer._prev, self._trailer)

def add_before(self, p, e):
  """Insert element e into list before Position p and return new Position."""
  original = self._validate(p)
  return self._insert_between(e, original._prev, original)

def add_after(self, p, e):
  """Insert element e into list after Position p and return new Position."""
  original = self._validate(p)
  return self._insert_between(e, original, original._next)

def delete(self, p):
  """Remove and return the element at Position p."""
  original = self._validate(p)
  return self._delete_node(original)      # inherited method returns element

def replace(self, p, e):
  """Replace the element at Position p with e.

  Return the element formerly at Position p.
  """
  original = self._validate(p)
  old_value = original._element           # temporarily store old element
  original._element = e                   # replace with new element
  return old_value                        # return the old element value
```