Coding Example: Modifying the list

This coding example will help you go through a simple and straightforward solution to add, delete, and set items in a list.

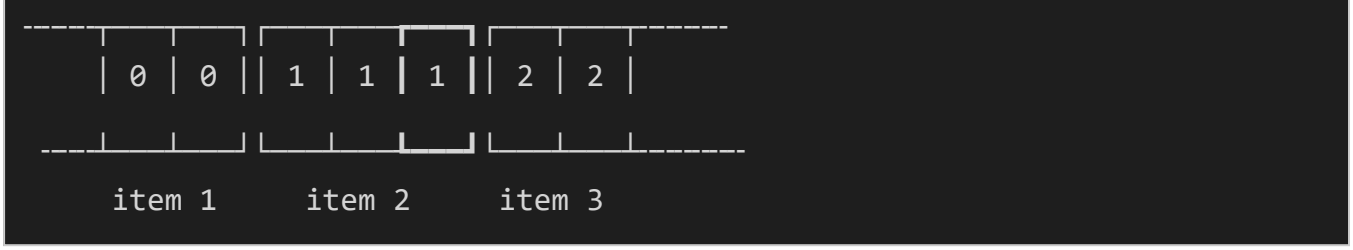
WE'LL COVER THE FOLLOWING

Modification of the list is a little complicated, because it requires managing the memory properly. The solution is pretty straightforward as it poses no real difficulty. You can have a look at the code below (the comments are added at each step to help you understand what's happening).

> **Note:** Always be careful with negative steps, key range and array expansion. When the underlying array needs to be expanded, it's better to expand it more than necessary in order to avoid future expansion.

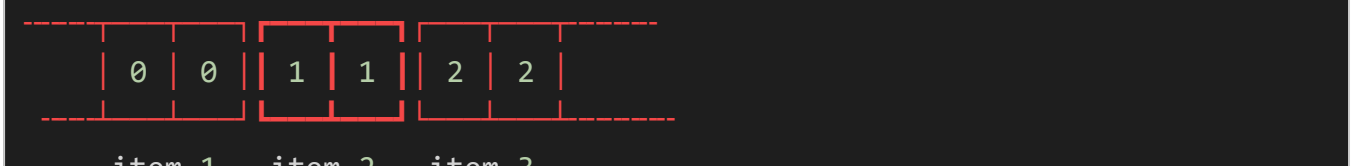
`setitem`

```
L[1] = 1,1,1

 ____ ____ ____ ____ ____ ____ ______
|    |    |    |    |    |    |      |
|  0 |  0 ||  1 |  1 ||  2 |  2 |      |
 ---- ---- ---- ---- ---- ---- ------

     item 1   item 2   item 3


OUTPUT
```



delitem

```
L = TypedList([[0,0], [1,1], [0,0]])
del L[1]
```

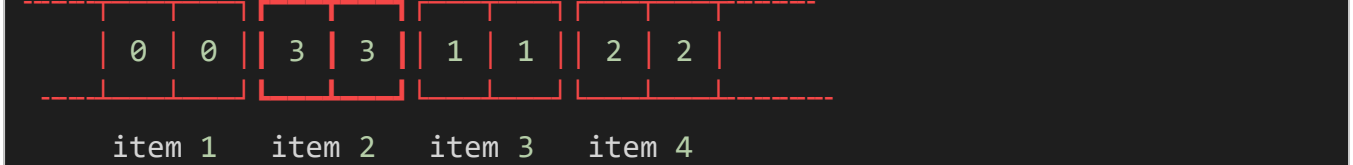



```
OUTPUT
```

insert

```
L = TypedList([[0,0], [1,1], [0,0]])
L.insert(1, [3,3])
```


```
OUTPUT
```



Complete Solution:

Here's the complete implementation of the list with all the methods added in the code:

```
# -*- coding: utf-8 -*-
# -----------------------------------------------------------------------------
# Copyright (c) 2014, Nicolas P. Rougier. All rights reserved.
# Distributed under the terms of the new BSD License.
# -----------------------------------------------------------------------------

import numpy as np

class ArrayList(object):
    """
    An ArrayList is a strongly typed list whose type can be anything that can be
    interpreted as a numpy data type.
    """

    def __init__(self, data=None, sizes=None, dtype=float, sizeable=True, writeable=True):
        """ Create a new buffer using given data and sizes or dtype

        Parameters
        ----------

        data : array_like
            An array, any object exposing the array interface, an object
            whose __array__ method returns an array, or any (nested) sequence.

        sizes:  int or 1-D array
            If `itemsize is an integer, N, the array will be divided
            into elements of size N. If such partition is not possible,
            an error is raised.

            If `itemsize` is 1-D array, the array will be divided into
            elements whose successive sizes will be picked from itemsize.
            If the sum of itemsize values is different from array size,
            an error is raised.

        dtype: np.dtype
            Any object that can be interpreted as a numpy data type.

        sizeable : boolean
            Indicate whether item can be appended/inserted/deleted

        writeable : boolean
            Indicate whether content can be changed
        """

        self._sizeable = sizeable
        self._writeable = writeable

        if data is not None:
            if type(data) in [list,tuple]:
                if type(data[0]) in [list,tuple]:
                    sizes = [len(l) for l in data]
                    data = [item for sublist in data for item in sublist]
            self._data = np.array(data, copy=False)
```

```
        self._size = self._data.size

        # Default is one group with all data inside

        _sizes = np.ones(1)*self._data.size

        # Check item sizes and get items count
        if sizes is not None:
            if type(sizes) is int:
                if (self._size % sizes) != 0:
                    raise ValueError("Cannot partition data as requested")
                self._count = self._size//sizes
                _sizes = np.ones(self._count,dtype=int)*(self._size//self._count)
            else:
                _sizes = np.array(sizes, copy=False)
                self._count = len(sizes)
                if _sizes.sum() != self._size:
                    raise ValueError("Cannot partition data as requested")
        else:
            self._count = 1

        # Store items
        self._items = np.zeros((self._count,2),int)
        C = _sizes.cumsum()
        self._items[1:,0] += C[:-1]
        self._items[0:,1] += C

    else:
        self._data = np.zeros(512, dtype=dtype)
        self._items = np.zeros((64,2), dtype=int)
        self._size = 0
        self._count = 0


@property
def data(self):
    """ The array's elements, in memory. """
    return self._data[:self._size]



@property
def itemsize(self):
    """ Individual item sizes """
    return self._items[:self._count,1] - self._items[:self._count,0]



@property
def size(self):
    """ Number of base elements, in memory. """
    return self._size



@property
def dtype(self):
    """ Describes the format of the elements in the buffer. """
    return self._data.dtype



def __len__(self):
```

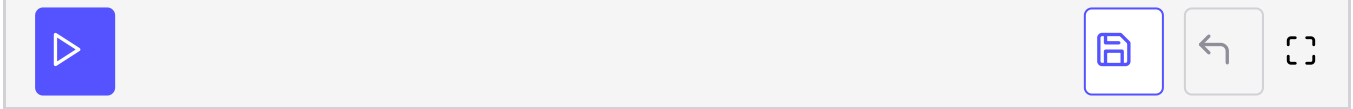
```
    """ x.__len__() <==> len(x) """
    return self._count



def __str__(self):
    s = '[ '
    for item in self: s += str(item) + ' '
    s += ']'
    return s



def __getitem__(self, key):
    """ x.__getitem__(y) <==> x[y] """

    if type(key) is int:
        if key < 0:
            key += len(self)
        if key < 0 or key >= len(self):
            raise IndexError("Tuple index out of range")
        dstart = self._items[key][0]
        dstop  = self._items[key][1]
        return self._data[dstart:dstop]

    elif type(key) is slice:
        istart, istop, step = key.indices(len(self))
        if istart > istop:
            istart,istop = istop,istart
        dstart = self._items[istart][0]
        if istart == istop:
            dstop = dstart
        else:
            dstop  = self._items[istop-1][1]
        return self._data[dstart:dstop]

    elif isinstance(key,str):
        return self._data[key][:self._size]

    elif key is Ellipsis:
        return self.data

    else:
        raise TypeError("List indices must be integers")


def __setitem__(self, key, data):
    """ x.__setitem__(i, y) <==> x[i]=y """

    if not self._writeable:
        raise AttributeError("List is not sizeable")

    if type(key) is int:
        if key < 0:
            key += len(self)
        if key < 0 or key > len(self):
            raise IndexError("List assignment index out of range")
        dstart = self._items[key][0]
        dstop  = self._items[key][1]
        self._data[dstart:dstop] = data

    elif type(key) is slice:
```

```
        istart, istop, step = key.indices(len(self))
        if istart > istop:
            istart,istop = istop,istart
        if istart == istop:
            dstart = self._items[key][0]
            dstop  = self._items[key][1]
            self._data[dstart:dstop] = data
        else:
            if istart > len(self) or istop > len(self):
                raise IndexError("Can only assign iterable")
            dstart = self._items[istart][0]
            if istart == istop:
                dstop = dstart
            else:
                dstop  = self._items[istop-1][1]
            self._data[dstart:dstop] = data

    elif key is Ellipsis:
        self.data[...] = data

    elif type(key) is str:
        self._data[key][:self._size] = data

    else:
        raise TypeError("List assignment indices must be integers")



def __delitem__(self, key):
    """ x.__delitem__(y) <==> del x[y] """

    if not self._sizeable:
        raise AttributeError("List is not sizeable")

    # Deleting a single item
    if type(key) is int:
        if key < 0:
            key += len(self)
        if key < 0 or key > len(self):
            raise IndexError("List deletion index out of range")
        istart, istop = key, key+1
        dstart,dstop = self._items[key]

    # Deleting several items
    elif type(key) is slice:
        istart, istop, step = key.indices(len(self))
        if istart > istop:
            istart,istop = istop,istart
        if istart == istop:
            return
        dstart = self._items[istart][0]
        dstop  = self._items[istop-1][1]

    # Ellipsis
    elif key is Ellipsis:
        istart,istop = 0, len(self)
        dstart, dstop = 0, self.size
    # Error
    else:
        raise TypeError("List deletion indices must be integers")

    # Remove data
```

```
        size = self._size - (dstop-dstart)
        self._data[dstart:dstart+size] = self._data[dstop:dstop+size]
        self._size -= dstop-dstart

        # Remove corresponding items
        size = self._count - istop
        self._items[istart:istart+size] = self._items[istop:istop+size]

        # Update other items
        size = dstop-dstart
        self._items[istart:istop+size+1] -= size, size
        self._count -= istop-istart



    def insert(self, index, data, sizes=None):
        """ Insert data before index

        Parameters
        ----------

        index : int
            Index before which data will be inserted.

        data : array_like
            An array, any object exposing the array interface, an object
            whose __array__ method returns an array, or any (nested) sequence.

        sizes:  int or 1-D array
            If `itemsize is an integer, N, the array will be divided
            into elements of size N. If such partition is not possible,
            an error is raised.

            If `itemsize` is 1-D array, the array will be divided into
            elements whose successive sizes will be picked from itemsize.
            If the sum of itemsize values is different from array size,
            an error is raised.
        """

        if not self._sizeable:
            raise RuntimeError("List is not sizeable")

        if type(data) in [list,tuple] and type(data[0]) in [list,tuple]:
            sizes = [len(l) for l in data]
            data = [item for sublist in data for item in sublist]

        data = np.array(data,copy=False).ravel()
        size = data.size

        # Check item size and get item number
        if sizes is not None:
            if type(sizes) is int:
                if (size % sizes) != 0:
                    raise ValueError("Cannot partition data as requested")
                _count = size//sizes
                _sizes = np.ones(_count,dtype=int)*(size//_count)
            else:
                _sizes = np.array(sizes,copy=False)
                _count = len(sizes)
                if _sizes.sum() != size:
                    raise ValueError("Cannot partition data as requested")
        else:
```

```
        _count = 1

    # Check if data array is big enough and resize it if necessary
    if self._size + size  >= self._data.size:
        capacity = int(2**np.ceil(np.log2(self._size + size)))
        self._data = np.resize(self._data, capacity)

    # Check if item array is big enough and resize it if necessary
    if self._count + _count  >= len(self._items):
        capacity = int(2**np.ceil(np.log2(self._count + _count)))
        self._items = np.resize(self._items, (capacity, 2))

    # Check index
    if index < 0:
        index += len(self)
    if index < 0 or index > len(self):
        raise IndexError("List insertion index out of range")

    # Inserting
    if index < self._count:
        istart = index
        dstart = self._items[istart][0]
        dstop  = self._items[istart][1]
        # Move data
        self._data[dstart+size:self._size+size] = self._data[dstart:self._size]
        # Update moved items
        I = self._items[istart:self._count]+size
        self._items[istart+_count:self._count+_count] = I

    # Appending
    else:
        dstart = self._size
        istart = self._count

    # Only one item (faster)
    if _count == 1:
        # Store data
        self._data[dstart:dstart+size] = data
        self._size += size
        # Store data location (= item)
        self._items[istart][0] = dstart
        self._items[istart][1] = dstart+size
        self._count += 1

    # Several items
    else:
        # Store data
        dstop = dstart + size
        self._data[dstart:dstop] = data
        self._size += size

        # Store items
        items = np.ones((_count,2),int)*dstart
        C = _sizes.cumsum()
        items[1:,0] += C[:-1]
        items[0:,1] += C
        istop = istart + _count
        self._items[istart:istop] = items
        self._count += _count


def append(self, data, sizes=None):
```

```
        """
        Append data to the end.

        Parameters
        ----------

        data : array_like
            An array, any object exposing the array interface, an object
            whose __array__ method returns an array, or any (nested) sequence.

        sizes:  int or 1-D array
            If `itemsize is an integer, N, the array will be divided
            into elements of size N. If such partition is not possible,
            an error is raised.

            If `itemsize` is 1-D array, the array will be divided into
            elements whose successive sizes will be picked from itemsize.
            If the sum of itemsize values is different from array size,
            an error is raised.
        """

        self.insert(len(self), data, sizes)

L = ArrayList( np.arange(10), [3,3,4])
print("L")
print (L)
#[[0 1 2] [3 4 5] [6 7 8 9]]
print("L.data")
print (L.data)
#[0 1 2 3 4 5 6 7 8 9]

# Set an item
print("L[0]=3,3,3")
L[0] = 3,3,3
print (L)
#[3 3 3] [3 4 5] [6 7 8 9]]

# Delete an item
print("del L[0]")
del L[0]
print (L)
#[[3 4 5][6 7 8 9]]

# Insert an item
print("L.insert(1, [3,3])")
L.insert(1, [3,3])
print (L)
#[[3 4 5] [3 3] [6 7 8 9]]
```



> **Note:** You can also add several items at once by specifying common or individual size: a single scalar means all items are the same size while a list of sizes is used to specify individual item sizes.

In the next lesson, we will discuss another interesting type called the *Memory-aware* array!