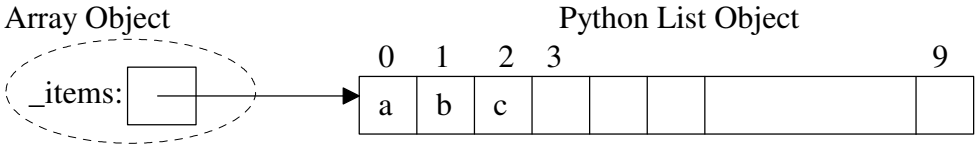
1. The textbook's Array class (section 13.2) is built upon a Python list representation. (Recall a Python list a contiguous block of memory similiar to an "array" in most programming language)



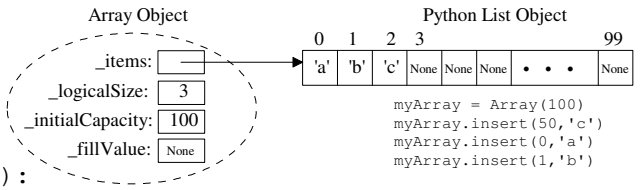
Not surprisingly, Array class methods call the corresponding list operations. However, the method names to overload operators are of the form `__xyz__`. The following table shows the relationship between operator usage and corresponding method names.

Array object user operation	Method called in Array class
`myArray = Array(100, 0)`	`__init__(self, capacity, fillValue = None)`
`len(myArray)`	`__len__(self)`
`str(myArray)`	`__str__(self)`
`for item in myArray:`	`__iter__(self)`
`item = myArray[i]`	`__getitem__(self, index)`
`myArray[i] = newValue`	`__setitem__(self, index, newItem)`

a) In the below code for the Array class, add appropriate precondition comments and `raise` statements.

```
""" File: arrays.py
     An Array is a restricted list whose clients can use only [], len, iter, and str.
     To instantiate, use:        <variable> = Array(<capacity>, <optional fill value>)
     The fill value is None by default.
"""
class Array(object):
    """Represents an array."""

    def __init__(self, capacity, fillValue = None):
        """Capacity is the static size of the array.
        fillValue is placed at each position."""
        self._items = list()
        for count in xrange(capacity):
            self._items.append(fillValue)

    def __len__(self):
        """-> The capacity of the array."""
        return len(self._items)

    def __str__(self):
        """-> The string representation of the array."""
        return str(self._items)

    def __iter__(self):
        """Supports traversal with a for loop."""
        return iter(self._items)

    def __getitem__(self, index):
        """Subscript operator for access at index.
           Precondition:                                                """



       return self._items[index]

    def __setitem__(self, index, newItem):
        """Subscript operator for replacement at index.
           Precondition:                                                """



        self._items[index] = newItem
```

b) The Array implementation above assumes that the length of the array cannot change after construction. If we want to improve the Array class so that it can dynamically change size, we need to keep track of the logical size as well as the physical size (e.g., `len(self._items)`). See HW #2 description for specifications, and complete code for `__init__`, `size`, `grow`, and `insert`.




```
class Array(object):
    """Represents a dynamic array."""

    def __init__(self, capacity = 0, fillValue = None):
        """Capacity is the initial size of the array.
        fillValue is placed at each position."""
        self._items = list()  # constructs an empty list
        for count in xrange(capacity):
            self._items.append(fillValue)

        if fillValue == None:
            self._logicalSize = 0
        else:
            self._logicalSize = capacity
        self._initialCapacity = capacity
        self._fillValue = fillValue

    def size(self):
        """Returns the logical capacity of the array."""

        return self._logicalSize



    def grow(self):
        """Doubles the physical capacity of the array"""
        currentSize = len(self._items)
        if currentSize == 0:
            currentSize = 1
        for count in xrange(currentSize):
            self._items.append(self._fillValue)



   def insert(self, index, newItem):
        """Adds newItem at spot index of array or at the logical end if index > size().
           Precondition:  0 <= index"""
        if index < 0:
            raise IndexError, "Index out of range"

        if self._logicalSize == len(self._items):        # array full
            self.grow()

        if index >= self._logicalSize:
            self._items[self._logicalSize] = newItem
        else:
            # make room for newItem at index
            for i in xrange(self._logicalSize, index, -1):
                self._items[i] = self._items[i-1]
            self._items[index] = newItem
        self._logicalSize += 1
```

Data Structures (CS 1520) HW #2 Due: Feb. 4 (Sat.) at 11:59 PM

To start the homework: Download and extract the file hw2.zip from

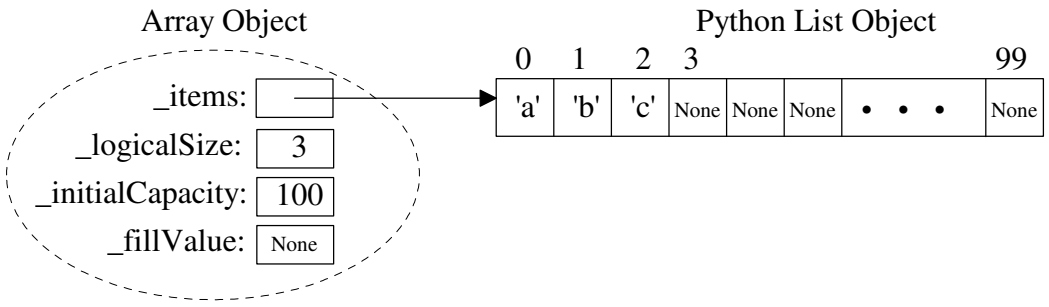
http://www.cs.uni.edu/~fienup/cs1520s12/homework/

This file contains the Array class with the following extensions (sensibly modified Projects 1-6 of Ch. 13):

- a `_logicalSize` data attribute to keep track of the number of items in the array. If no `fillValue` is provided, the `_logicalSize` is set to 0. Otherwise, is it set to the `capacity` since we generally use the `fillValue` to initialize actual items in the array.
- a `_initialCapacity` data attribute to remember the `capacity` parameter of `__init__` . If the physical size of the array grows and then later shrinks, we don't want it to shrink below the `_initialCapacity`.
- a `_fillValue` data attribute to remember the `fillValue` parameter of `__init__` . If the physical size of the array grows, we will use it to initialize the extra slots. (this seems pointless since the Array user should be accessing only the items counted in the logical size)
- a `size` method which returns the array's logical size
- a `grow` method to double the physical size of the Array using the `_fillValue` to initialize the extra slots.
- an `insert` method which expects a position and an item as arguments and inserts the item at the given position. If the position is greater than or equal to the logical size, the item is inserted after the last item in the array before `insert` was called. If the position is less than 0, an error should be raised. If the Array is full when `insert` is called, the `grow` method is called.

The following code generates the shown Array object, myArray.

```
myArray = Array(100)
myArray.insert(50,'c')
myArray.insert(0,'a')
myArray.insert(1,'b')
```



Part A: Add the following Array methods:

- a `shrink` method to halve the physical size of the Array when it is only a quarter full. However, the physical size of the Array should not be allowed to drop below the `_initialCapacity`.
- a `remove` method which expects a position as an argument. The `remove` method should remove and return the item at that position. The "hole" left by removing the item at position should be filled by sliding the items at larger positions down with the spot vacated by the last item filled by the `_fillValue`. If the Array's logical size drops to a quarter full, the `shrink` method should be called to avoid wasting memory space. Be sure to include a precondition of 0 <= position index < size() and code to check the precondition and raise an appropriate exception
- an `__eq__` method to allow two Array objects to be compared using the == operator. This method should return `True` if the right-hand-side argument is also an Array with the same logical size and corresponding pairs of items from each logical position in the two arrays are equal. Otherwise, this method should return `False`.

Part B: Update the interactive menu-driven testArray.py program to allow testing of your Part A methods. For testing the `__eq__` method use the compareArray with myArray. You get to build myArray so you can check all possible outcomes.

Part C: Answer the following questions raised in Project 6.

- Why should we remove the current implementation of the `__iter__` method? (NOTE: we don't know how to fix this problem yet since it is not addressed until section 16.6.2)
- Explain how the `__str__` method should be modified at this point.

(additional requirements on reverse side)

Data Structures (CS 1520) HW #2 Due: Feb. 4 (Sat.) at 11:59 PM

Implementation Requirements:

- Use meaningful variable names with good style (i.e., useCamelCase or use_underscores)
- Use docstring comments at the module, class, method, and function levels. Include preconditions and postconditions for all functions and class methods. For each function or method with a precondition, include code to check the precondition and raise an appropriate exception.

Implementation Suggestions:

- Don't procrastinate! Start the project early so if you have problems you can get help.
- Test your program thoroughly using your updated testArray.py program

Submit your homework electronically at https://www.cs.uni.edu/~schafer/submit/which_course.cgi
The steps for the homework submission system are:

1. Write, debug, and test your program in the hw2 folder. When you are ready to submit your homework, zip the whole folder by right-clicking on it and selecting `Send to | Compressed (zipped) folder`. This will create a new file called `hw2.zip` which you will submit electronically.
2. Log on to the submission system at: **https://www.cs.uni.edu/~schafer/submit/which_course.cgi**

   (It is very likely that you will get some security certificate warnings when trying to use this. You may add an exception and accept the existing security certificate.) Use the same AD-ITS User name and password you use to log on the lab computers.
3. Select the course and section number of "810:052, Data Structures, Fienup". Click the "Continue".
4. Select the homework that you wish to submit: "HW 2: Array class". Click the "Continue" button.
5. Specify how many extra files you want to submit. Just leave it at 0. Click the "Continue" button.
6. Upload your program by Browsing and selecting your `hw2.zip` file. Click the "Continue" button.
7. The next page reports on the status of the upload(s). You can always continue to upload a better version of the program until the deadline. The newer file will replace an older file of the same name.