National University of Sciences & Technology (NUST)
School of Natural Sciences (SNS)
Department of Mathematics

CS250: Data Structures and Algorithm

Class: BS Mathematics - 2021

Lab 05:

Array Oriented Programming in Python

Array vs Lists in Python

Date: 11th October, 2023

Time: 10:00 am - 01:00 pm

Instructor: Fauzia Ehsan

National University of Sciences & Technology (NUST)
School of Natural Sciences (SNS)
Department of Mathematics

Lab 05: Introduction to Array-Oriented Programming in Python

Introduction

The purpose of this lab is to have to get familiar with array's data structures and its implementation in Python. We will also try to compare the performance of arrays with built-in lists.

.

Tools/Software Requirement
Python 3/ PyCharm IDE / VS Code

Contents

1. Low Level Arrays

A group of related variables can be stored one after another in a contiguous portion of the computer's memory. We will denote such a representation as an **array**. As a tangible example, a text string is stored as an ordered sequence of individual characters. In Python, each character is represented using the Unicode character set, and on most computing systems, Python internally represents each Unicode character with 16 bits (i.e., 2 bytes). Therefore, a six-character string, such as **SAMPLE** , would be stored in **12 consecutive bytes of memory,** as diagrammed in Figure 5.2.

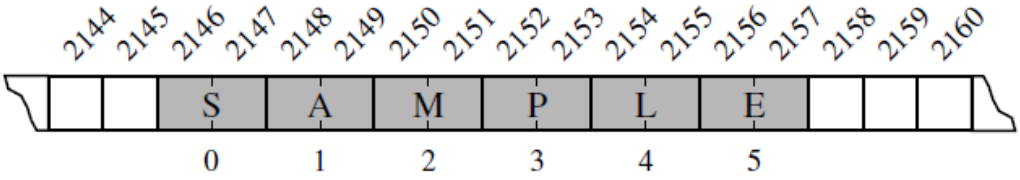

Figure 5.2: A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires two bytes of memory. The numbers below the entries are indices into the string.

We describe this as an array of six characters, even though it requires 12 bytes of memory. We will refer to each location within an array as a cell, and will use an integer index to describe its location within the array, with cells numbered starting with 0, 1, 2, and so on. For example, in Figure 5.2, the cell of the array with index 4 has contents L and is stored in bytes 2154 and 2155 of memory.

Each cell of an array must use the same number of bytes. This requirement is what allows an arbitrary cell of the array to be accessed in **constant time** based on its index. In particular, if one knows the memory address at which an array starts (e.g., 2146 in Figure 5.2), the number of bytes per element (e.g., 2 for a Unicode character), and a desired index within the array, the appropriate memory address can be computed using the calculation, **start + cellsize*index**. By this formula, the cell at index 0 begins precisely at the start of the array, the cell at index 1 begins precisely cellsize bytes beyond the start of the array, and so on. As an example, cell 4 of Figure 5.2 begins at memory location $2146+2 \cdot 4 = 2146+8 = 2154$.

2. Referential Arrays

As another motivating example, assume that we want a medical information system to keep track of the patients currently assigned to beds in a certain hospital. If we assume that the hospital has 200 beds, and conveniently that those beds are numbered from 0 to 199, we might consider using an array-based structure to maintain the names of the patients currently assigned to those beds. For example, in Python we might use a list of names, such as:

```
[ Rene , Joseph , Janet , Jonas , Helen , Virginia , ... ]
```

To represent such a list with an array, Python must adhere to the requirement that each cell of the array use the **same number of bytes**. Yet the elements are strings, and strings naturally have different lengths. Python could attempt to reserve enough space for each cell to hold the maximum length string (not just of currently stored strings, but of any string we might ever want to store), but that would be **wasteful.**

Instead, Python represents a list or tuple instance using an internal storage mechanism of **an array of object references.** At the lowest level, what is stored is a **consecutive sequence of memory addresses** at which the elements of the sequence reside. A high-level diagram of such a list is shown in Figure 5.4.

Although the relative size of the individual elements may vary, the number of bits used to store the memory address of each element is fixed (**e.g., 64-bits per address).** In this way, Python can support constant-time access to a list or tuple element based on its index.

In Figure 5.4, we characterize a list of strings that are the names of the patients in a hospital. It is more likely that a medical information system would manage more comprehensive information on each patient, perhaps represented as an instance of a Patient class. From the perspective of the list implementation, the same principle applies: **The list will simply keep a sequence of references to those objects.**

Note as well that a reference to the **None** object can be used as an element of the list to represent an empty bed in the hospital.

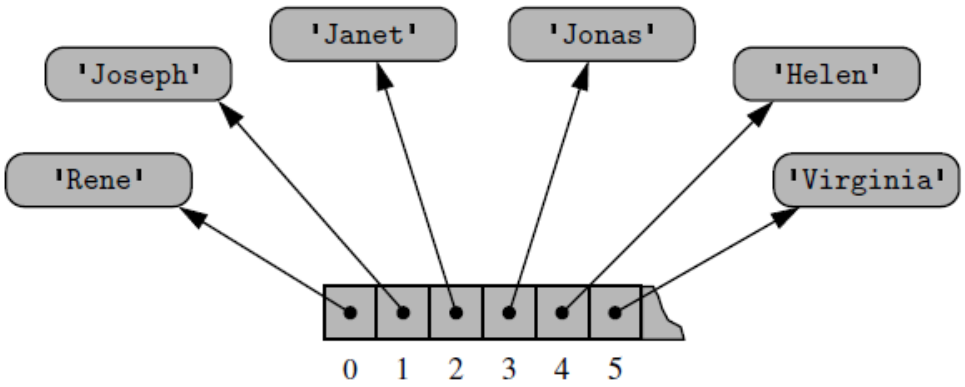

Figure 5.4: An array storing references to strings.

Primary support for compact arrays is in a module named **array**. That module defines a class, also named array, providing compact storage for arrays of primitive data types. A portrayal of such an array of integers is shown in Figure 5.10.

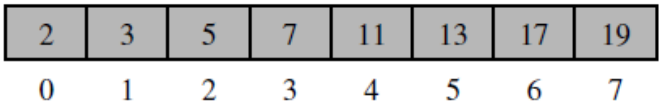

Figure 5.10: Integers stored compactly as elements of a Python array.

The public interface for the array class conforms mostly to that of a Python list. However, the constructor for the array class requires a **type code** as a first parameter, which is a character that designates the type of data that will be stored in the array.

As a tangible example, the type code, **`i`** , designates an array of (signed) integers, typically represented using at least 16-bits each. We can declare the array shown in Figure 5.10 as,

```
import array as arr
primes = arr.array('i' , [2, 3, 5, 7, 11, 13, 17, 19])
```

The type code allows the interpreter to determine precisely how many bits are needed per element of the array. The type codes supported by the array module, as shown in Table 5.1, are formally based upon the native data types used by the C programming language (the language in which the most widely used distribution of Python is implemented). You can also use **numpy** module to create arrays in Python.

See the attached comprehensive tutorial of NumPy

3. Dynamic Arrays Implementation:

The Python **list** class provides a highly optimized implementation of dynamic arrays but we can also create our own dynamic arrays class.

```
1 import ctypes # provides low-level arrays

3 class DynamicArray:
4 """A dynamic array class akin to a simplified Python list."""
5
6 def init (self):
7 """Create an empty array."""
8 self. n = 0 # count actual elements
9 self. capacity = 1 # default array capacity
10 self. A = self. make array(self. capacity) # low-level array
11
12 def len (self):
13 """Return number of elements stored in the array."""
14 return self. n
15
16 def getitem (self, k):
17 """Return element at index k."""
18 if not 0 <= k < self. n:
19 raise IndexError( invalid index )
20 return self. A[k] # retrieve from array
```

```
21
22 def append(self, obj):
23 """Add object to end of the array."""
24 if self. n == self. capacity: # not enough room
25 self. resize(2 self. capacity) # so double capacity
26 self. A[self. n] = obj
27 self. n += 1
28
29 def resize(self, c): # nonpublic utitity
30 """Resize internal array to capacity c."""
31 B = self. make array(c) # new (bigger) array
32 for k in range(self. n): # for each existing value
33 B[k] = self. A[k]
34 self. A = B # use the bigger array
35 self. capacity = c
36
37 def make array(self, c): # nonpublic utitity
38 """Return new array with capacity c."""
39   return (c ctypes.py object)( ) # see ctypes documentation
```

Task1a.

Consider the **`Array class`** in the **`Array.py`** program (Download from LMS), add a method called **`getMaxNum()`** that returns the value of the highest number in the array, or `None` if the array has no numbers. You can use the expression **`isinstance(x, (int, float))`** to test for numbers.

- Add some code to **`ArrayClient.py`** to exercise this method.
- You should try it on arrays containing a variety of data types and some that contain zeros and some that contain no numbers.

Task1b.

Modify the method in Task1a so that the item with the highest numeric value is not only returned by the method but also removed from the array. Call the method **`deleteMaxNum().`**

Task 2.

Write a **`removeDupes()`** method for the **`Array.py`** program (Listing 2-3) that removes any duplicate entries in the array. That is, if three items with the value **`'bar'`** appear in the array, **`removeDupes()`** should remove two of them.

Don’t worry about maintaining the order of the items.

One approach is to make a new, empty list, move items one at a time into it after first checking that they are not already in the new list, and then set the array to be the new list. Of course, the array size will be reduced if any duplicate entries exist.

Write some **tests** to show it works on arrays with and without duplicate values.

Task03.

Repeat the above tasks with Python built-in Lists and numpy arrays.
and compare the performance of all three techniques (i-e. by using the above Array class methods, built-in lists and numpy arrays) using python `timeit` module.

Observe your results against different array and lists lengths.
Plot your results using matplotlib module. (a toy example is given at the end of this manual.)
To install, matplotlib, write in terminal the following command:

py -m pip install matplotlib
or just
pip install matplotlib

Task04. List vs Arrays Performance

Use `timeit` to compare the execution time of the following two statements.

The first uses a list comprehension to create a list of the integers from 0 to 9,999,999, then totals them with the built-in `sum` function.

The second statement does the same thing using an array and its sum method.

```
sum([x for x in range(10_000_000)])
np.arange(10_000_000).sum()
```

Task05. (Median and Mode of an array)

NumPy arrays offer a mean method, but not median or mode. Write functions median and mode that use existing NumPy capabilities to determine the **median** (middle) and **mode** (most frequent) of the values in an array. Your functions should determine the median and mode regardless of the array's shape. Test your function on three arrays of different shapes.

Matplotlib

Matplotlib is a Python library that provides MATLAB-like plotting functions in Python. Simple plots such as bar graphs and line graphs are very easy to create using matplotlib. Using the **`.plot()`** and **`.bar()`** methods of **`matplotlib.pyplot`**, we can quickly show line and bar graphs. **`.plot()`** and **`.bar()`** take a dataset for x and a dataset for y, with many more optional parameters such as align.

Let's plot the following x and y coordinates:

x	0	2	4	6	8	10	12	14	16	18
y	0	4	8	12	16	20	24	28	32	36

Table 1Values for Figure 01

```
import matplotlib.pylab as plt
 x = range (0 , 20 , 2) y
= range (0 , 40 , 4)
plt.plot (x, y, '.')
plt.xlabel ('X axis')
plt.ylabel ("Y axis")
plt.title ('Random plot of X vs Y') plt.show()
```

Determine the output?
Figure 01

The **`pl.plot()`** function takes as arguments a sequence for x, a sequence for y, and optionally a formatting specifier. The important ones are "-" for a solid line and "." for point markers. You can add more options, such as **colors** or **labels**. You can find more docs on these specifiers on the Matploblib website:

https://matplotlib.org/stable/

You have to call **`pl.show()`** for the graph to actually show. There are ways to print the graphs to image files, but we are not covering those for now. You can look them up in the documentation if you want. Bar plots work in a similar way.

The **`pl.bar()`** takes a sequence of numbers to label the left side of the bars with and a sequence of heights of the bars.

```
import matplotlib.pylab as pl

x = range(1 , 11 , 2)

y = range(1 , 21 , 4)

pl.bar(x, y)
 pl.show ()
```

Determine the output?
Figure 02
These are the two main functions you will need for plotting in this lab. If you want to customize more, please see the matplotlibpylab documentation at

https://matplotlib.org/stable/api/pyplot_summary.html

Grade Criteria

This lab is graded. Min marks: 0. Max marks: 30

Tasks Shown during Lab (At least 2 Tasks must be shown to LE during lab working) 10 marks	Modern Tool Usage 5 marks	Lab Ethics 5 Marks	Lab Report and Tasks Final Submission 5 marks	Lab Viva/Quiz 5 Marks
Total Marks: 30	**Obtained Marks: _____**			

Happy Coding!

Deliverables

Comment your program heavily. Intelligent comments and a clean, readable formatting of your code account for 20% of your grade.
You should submit your codes and report as a compressed zip file. It should contain all files used in the exercises for this lab.
The submitted file should be named cs250_firstname_lastname_lab05.zip