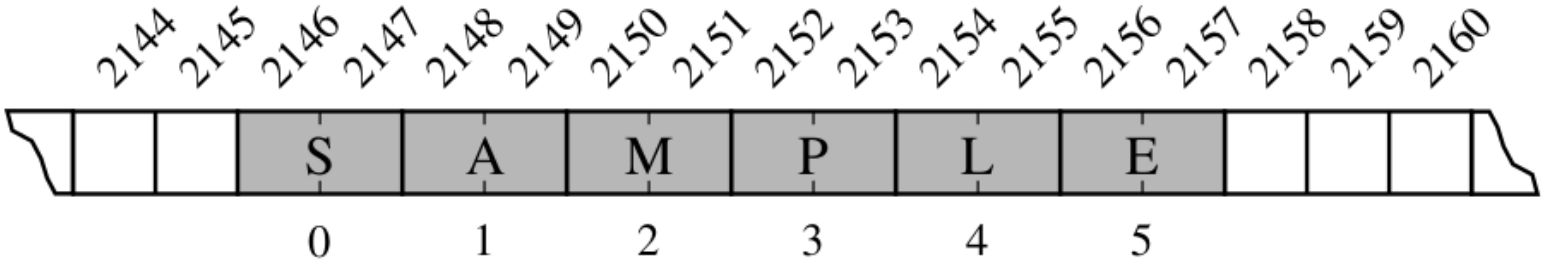
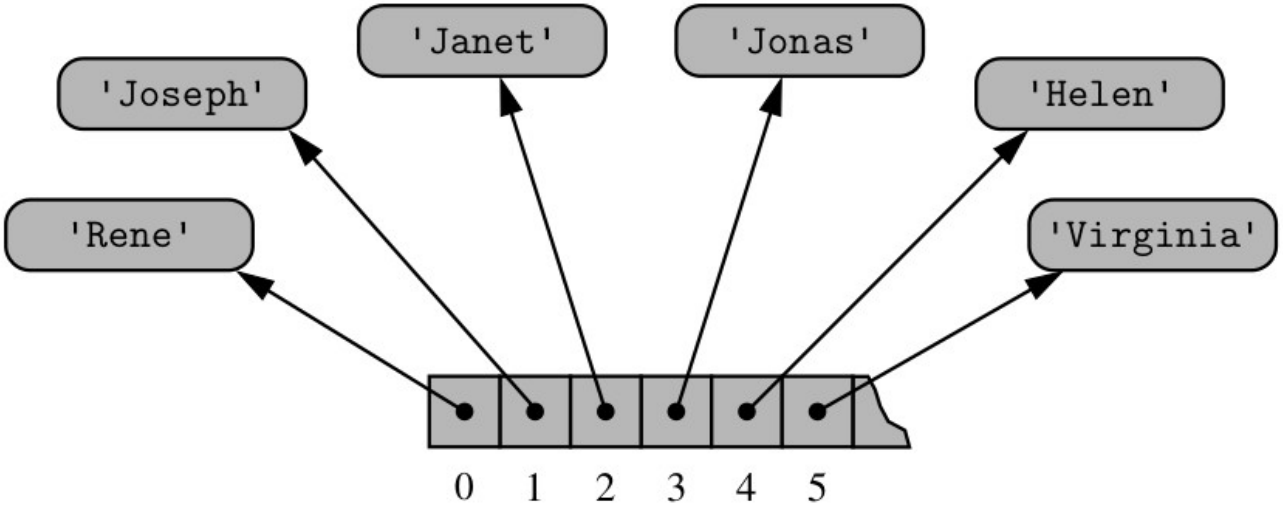
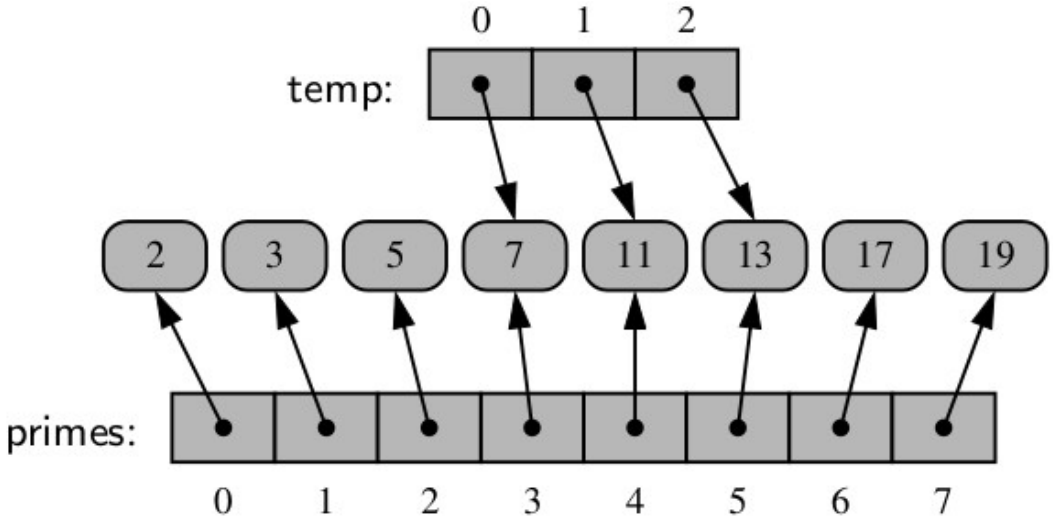
Array based Sequences

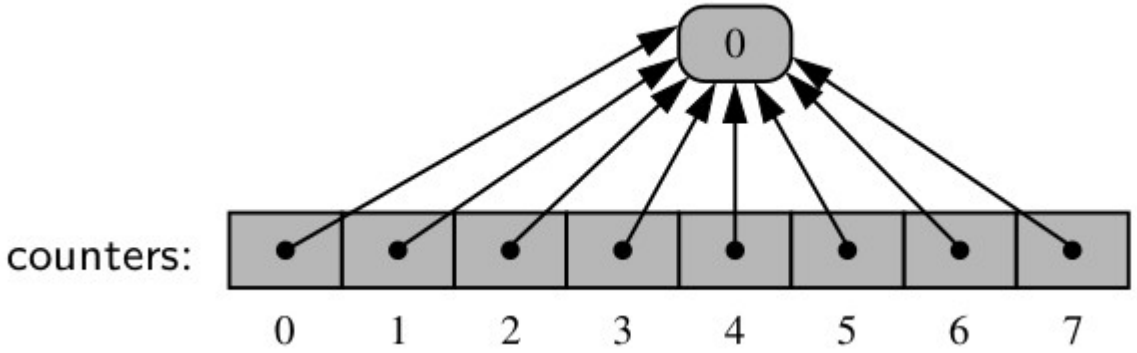
Compact and referential arrays





Referential arrays





Type Codes in the array Class

Python's array class has the following type codes:

Code	C Data Type	Typical Number of Bytes
`'b'`	signed char	1
`'B'`	unsigned char	1
`'u'`	Unicode char	2 or 4
`'h'`	signed short int	2
`'H'`	unsigned short int	2
`'i'`	signed int	2 or 4
`'I'`	unsigned int	2 or 4
`'l'`	signed long int	4
`'L'`	unsigned long int	4
`'f'`	float	4
`'d'`	float	8

Compact Arrays

```
# Array of floats:
farray = arr.array('f', [1.21, 3.054, 7.9])

# Array of characters:
c_array = arr.array('u', 'hello')  # 'u' for Unicode characters

# Array of integers:
i_array = arr.array('i', [100, 2, 230, 40])  # 'i' for signed
    integers
```

Numpy library can also be used for using compact arrays

Dynamic array implementation

In an add(elem) operation (without an index), we could always add at the end

When the array is full, we can replace the array with a larger one

But how large should the new array be?

Incremental strategy: increase the size by a constant c

Doubling strategy: double the size

Algorithm *add*(*elem*)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow elem$

Steps to implement dynamic arrays

1. Allocate a new array B with larger capacity.

2. Set B[i] = A[i], for i = 0, . . . , n − 1, where n denotes current number of items.

3. Set A = B, that is, we henceforth use B as the array supporting the list.

4. Insert the new element in the new array.

Incremental Strategy Analysis

Proved in the class

Time complexity for n insertions is $O(n^2)$

The amortized time of an add operation is $O(n)$

Doubling Strategy Analysis

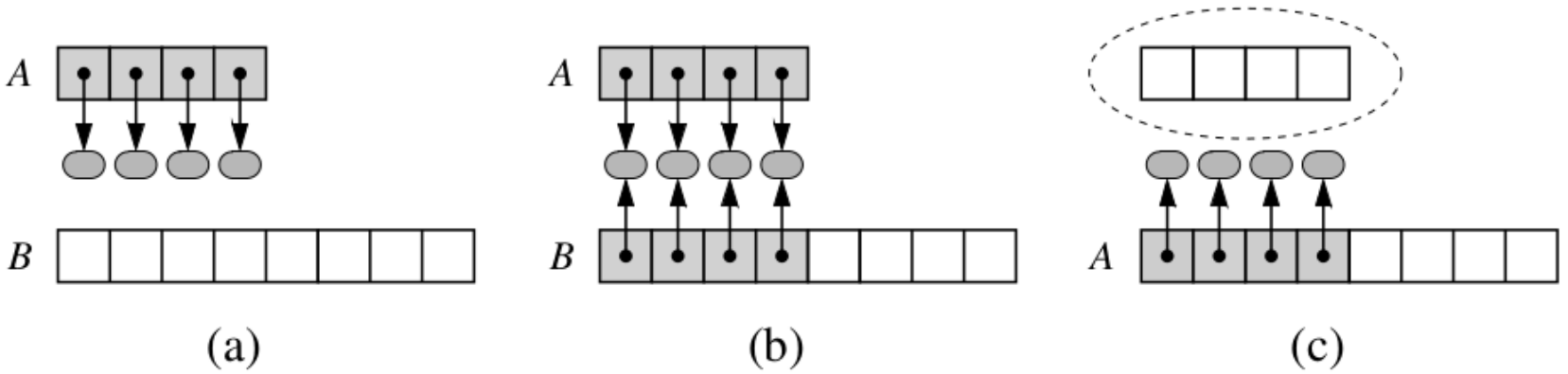
Proved in the class

The total time $T(n)$ of a series of n add operations is proportional to

$\sim 3n$

$T(n)$ is $O(n)$

The amortized time of an add operation is $O(1)$

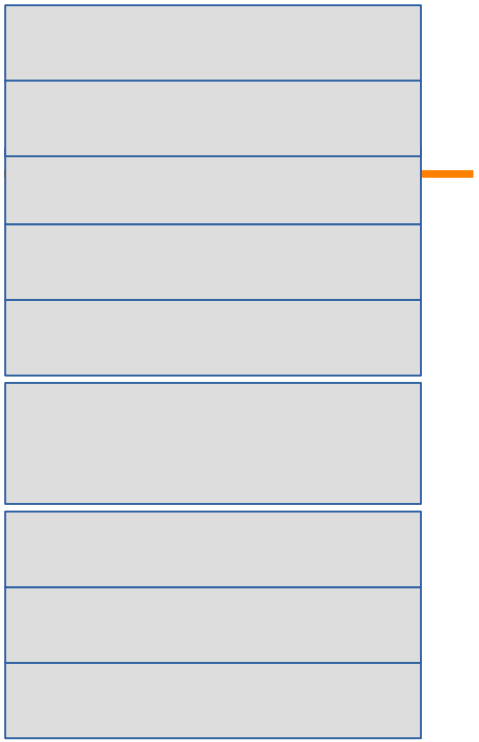

A
B
(a)
A
B
(b)
A
(c)

```
import sys
data = [ ]
for k in range(100):
 length = len(data)  # number of elements

 b = sys.getsizeof(data) # actual size in bytes

 print( f" array length is {length} Size in bytes: {b}" )
 data.append(None)
```

Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$



Asymptotic performance of the nonmutating behaviors of the list and tuple classes. Identifiers data, data1, and data2 designate instances of the list or tuple class, and n, n1 , and n2 their respective lengths. For the containment check and index method, k represents the index of the leftmost occurrence (with k = n if there is no occurrence). For comparisons between two sequences, we let k denote the leftmost index at which they disagree or else k = min(n1 , n2)

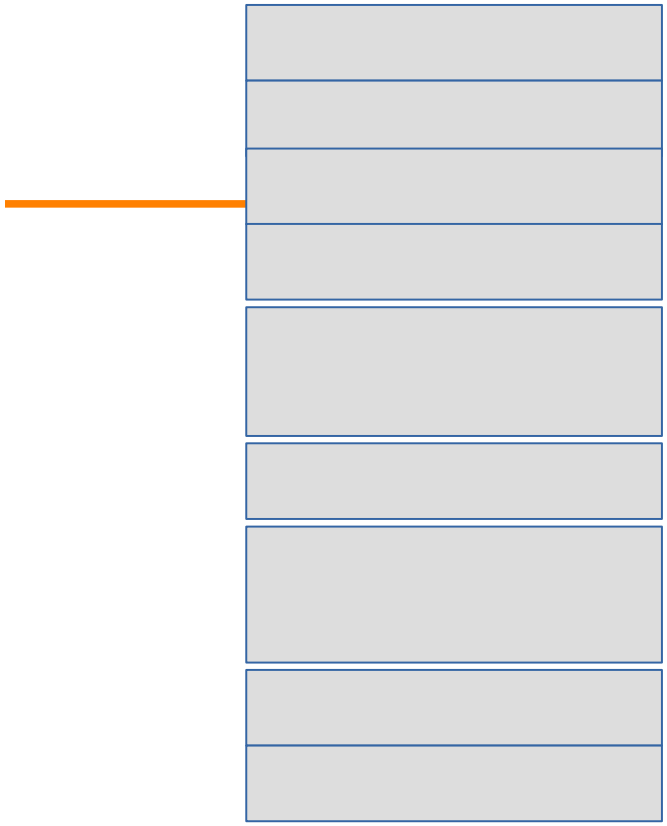
Average running time of append

n	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
μ**s**	0.219	0.158	0.164	0.151	0.147	0.147	0.149

Average running time of append, measured in microseconds, as observed over a sequence of n calls, starting with an empty list.

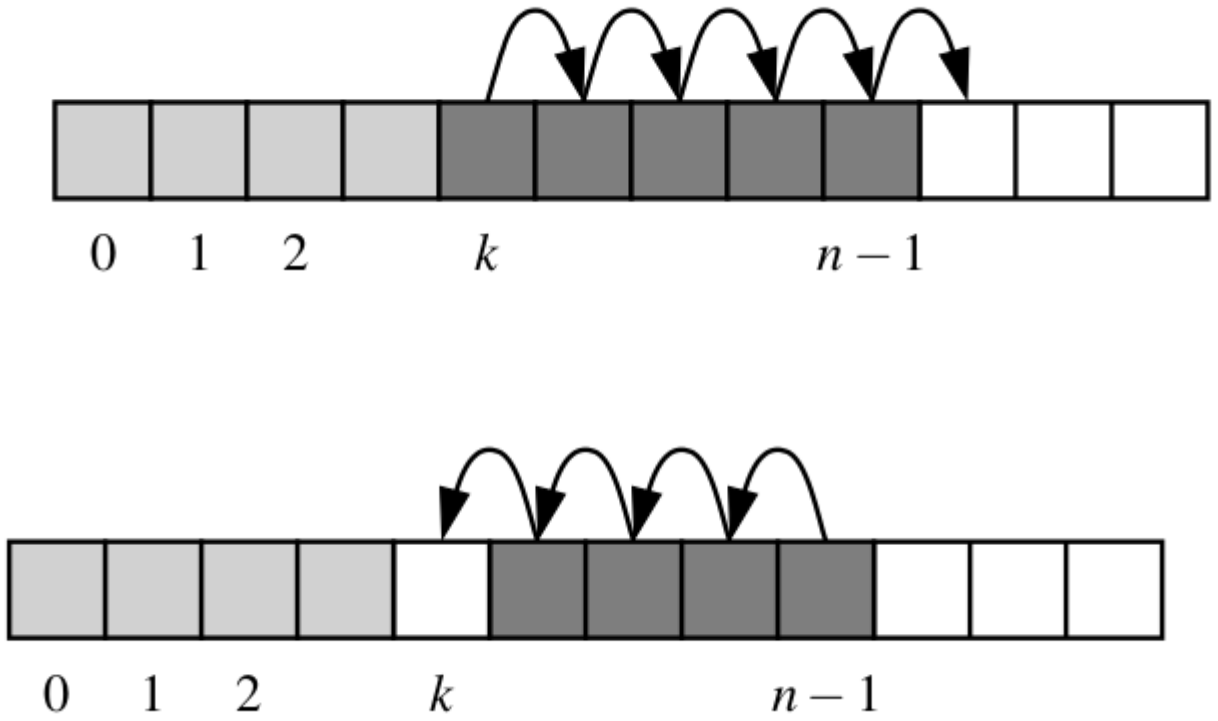
Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse( )	$O(n)$
data.sort( )	$O(n \log n)$

*amortized



Asymptotic performance of the mutating behaviors of the list class. Identifiers data, data1, and data2 designate instances of the list class, and n, n1 , and n2 their respective lengths.

Insertion & Removal



In the worst case ($i = 0$), both take $O(n)$ time

Average running time of insert(k, val) (measured in microseconds)

	N				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n$ // 2	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

Average running time of insert(k, val), measured in microseconds, as observed over a sequence of N calls, starting with an empty list. The variable n denotes the size of the current list (as opposed to the final list).

Performance

In an array based implementation of a dynamic list:

The space used by the data structure is $O(n)$

Indexing the element at I takes $O(1)$ time

add and ***remove*** run in $O(n)$ time in worst case

In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one…

Multidimensional Data

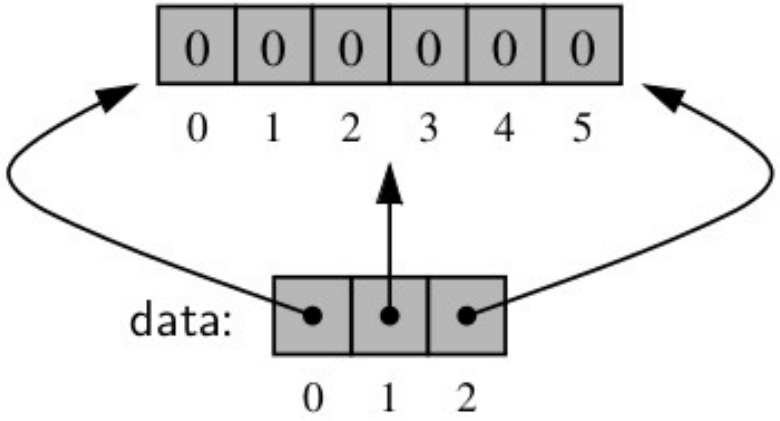
	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

data = [[22, 18, 709, 5, 33], [45, 32, 830, 120, 750], [4, 880, 45, 66, 61]]

How to initialise a matrix with zeros

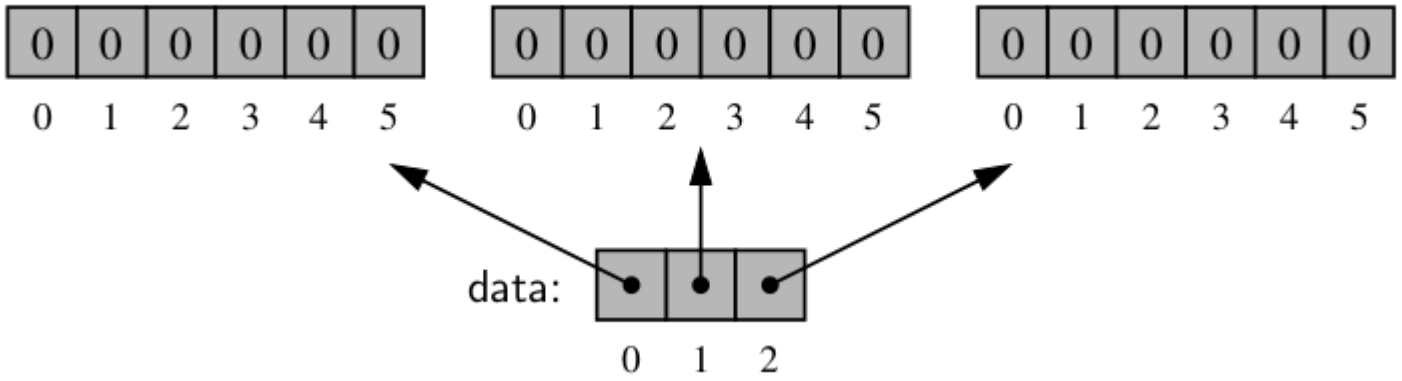
$$\text{data} = ([0] * c) * r$$

$$\text{data} = [\,[0] * c\,] * r$$



How to initialise a matrix with zeros

```
data = [ [0] * c for j in range(r) ]
```



This is the right procedure

Implementation of Dynamic Array

```
1  import ctypes                                    # provides low-level arrays
2
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
```

Implementation of Dynamic 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
```

References

Data Structures and Algorithms in Python
Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Introduction to Algorithms
Leiserson,Stein,Rivest,Cormen

Algorithms, 4th Edition
Robert Sedgewick and Kevin Wayne

Few Images from the internet