

lot of memory  
↳ import array  
↳ define array  
↳ not pointer based = less memory  
↳ only int, float, char

tuple, list, array  
↳ hold addresses  
↳ point to actual object

# Array-Based Sequences

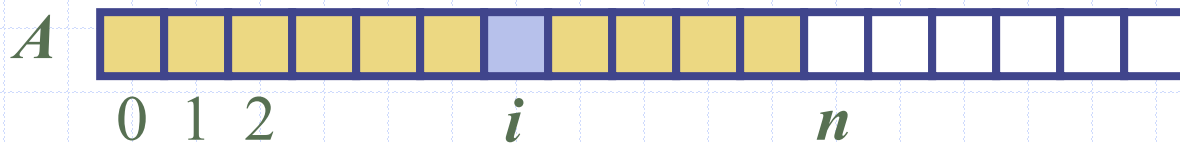


Sareh Taebi

COP3410 – Florida Atlantic University

# Python Sequence Classes

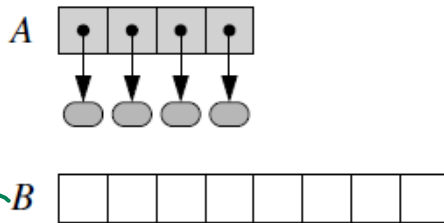
- Python has built-in types, **list**, **tuple**, and **str**.
- Each of these **sequence** types supports indexing to access an individual element of a sequence, using a syntax such as  $A[i]$
- Each of these types uses an **array** to represent the sequence.
  - An array is a set of memory locations that can be addressed using consecutive indices, which, in Python, start with index 0.



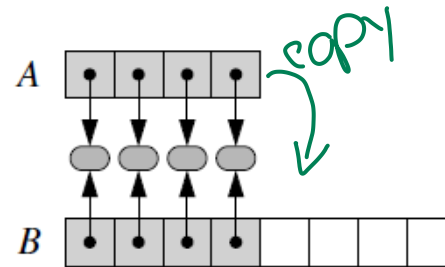
# Dynamic Array <sup>List & tuple</sup>

- ❑ A Python **list** can grow.
- ❑ Python list class uses an efficient dynamic array implementation

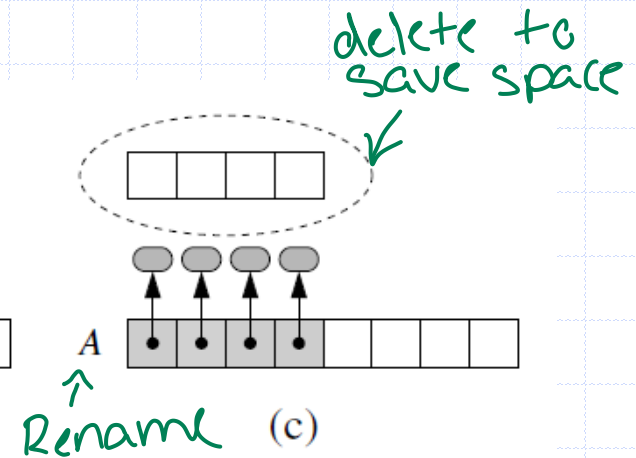
create more memory than needed



(a)



(b)



(a) create new array *B* (b) store elements of *A* in *B* (c) reassign reference *A* to the new array.

# Running times of append operations on dynamic arrays

list  
↳ list.append(value)

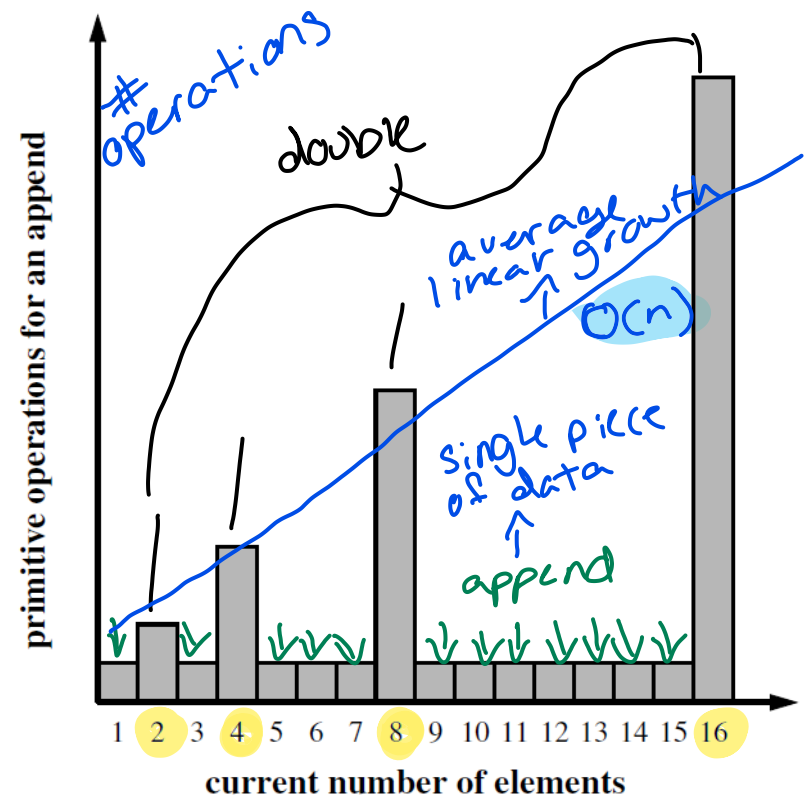
- After every  $n$  append operations, the list capacity **doubles**.
  - This is efficient.
- The total time to perform a series of  $n$  append operations is  **$O(n)$** .

size = 1

size = size \* 2 # 2

size = size \* 2 # 4

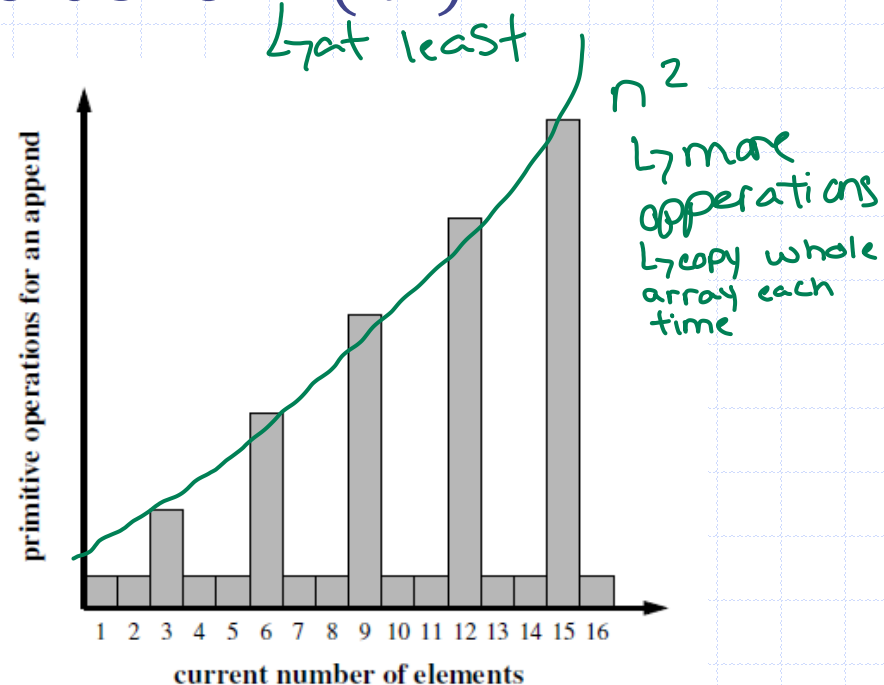
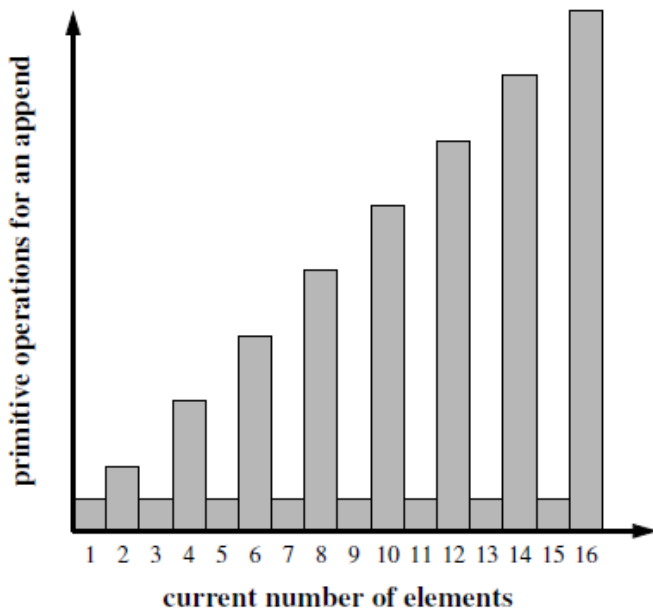
size = size \* 2 # 8



geometric progression  
↳  $O(n)$  ↳ multiplying 4

# Beware of Arithmetic Progression

- *adding # each time* **Arithmetically** increasing the array size by adding 2 or 3 may seem to save memory space.
- Run time will be in the order of  $\Omega(n^2)$



Non-mutating behaviors

↳ worst case complexity of  $O(n)$

# Efficiency of list and tuple

↳ not dynamic  
↳ cannot expand

## □ Non-mutating behaviors of tuples and lists

1 2 4 2  
data.count(2)  
↳ comparing each

Operation	Running Time
len(data)	$O(1)$ → creates list & saves size
reading value ← data[j]	$O(1)$
data.count(value)	$O(n)$ # times value in seq. ↳ repeats
data.index(value)	$O(k+1)$ → 1st
value in data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
slicing data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1 + n_2)$
c * data	$O(cn)$

data, data1, and data2 designate instances of the list or tuple class, and  $n$ ,  $n_1$ , and  $n_2$  their respective lengths.

# Mutating Behaviors of `list` class

↓  
can mutate, dynamic  
↳ no tuples

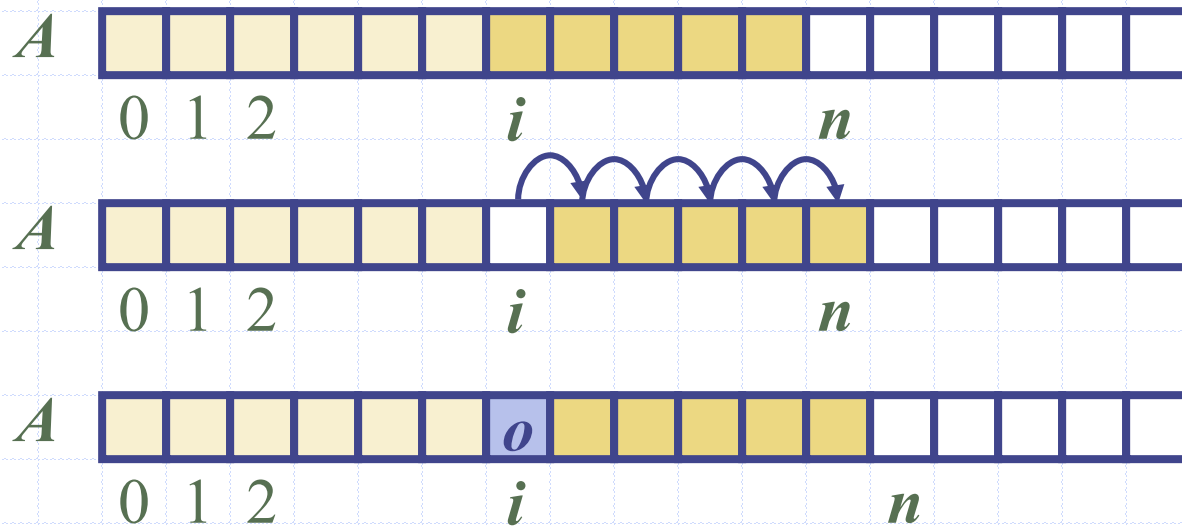
Operation	Running Time
<code>data[j] = val</code>	$O(1)$
<code>data.append(value)</code>	$O(1)^*$
<code>data.insert(k, value)</code>	$O(n - k + 1)^*$
<code>data.pop()</code>	$O(1)^*$
<code>data.pop(k)</code> <code>del data[k]</code>	$O(n - k)^*$
<code>data.remove(value)</code>	$O(n)^*$
<code>data1.extend(data2)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

\*amortized → best case

add  $\rightarrow$  algorithmic

# Insertion: `data.insert(i, o)`

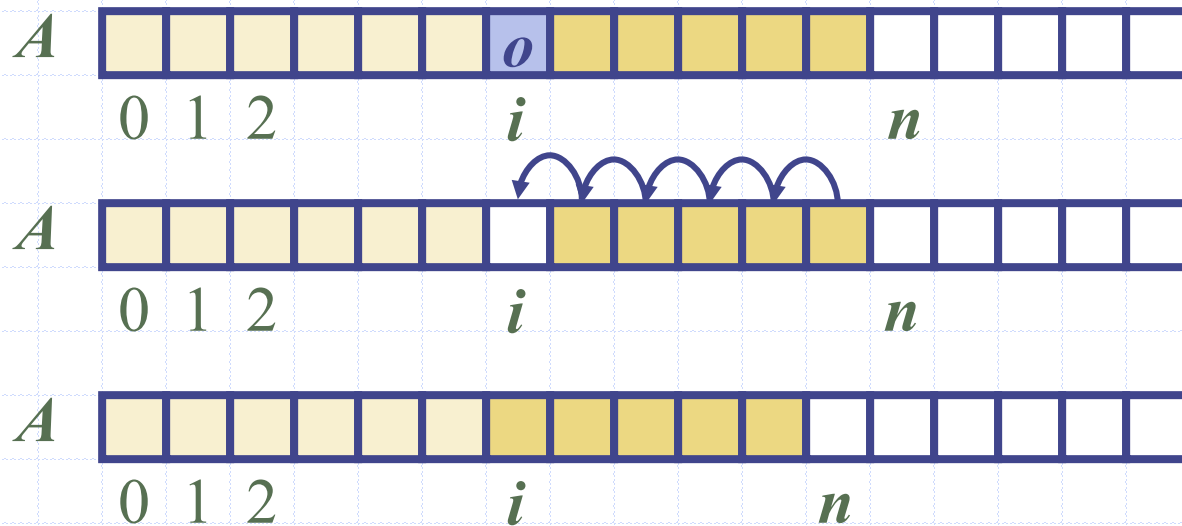
- In an operation **add**( $i, o$ ), we need to make room for the new element by shifting forward the  $n - i$  elements  $A[i], \dots, A[n - 1]$
- In the worst case ( $i = 0$ ), this takes  $O(n)$  time





# Element Removal: `data.pop(i)`

- In an operation `remove(i)`, we need to fill the hole left by the removed element by shifting backward the  $n - i - 1$  elements  $A[i + 1], \dots, A[n - 1]$
- In the worst case ( $i = 0$ ), this takes  $O(n)$  time



# Extending a List

- Python provides a method named `extend` that is used to add all elements of one list to the end of a second list.

`for element in other:`

`data.append(element)`

equivalent to :

`data.extend(other)`

`data += other`

Preferred

↳ don't use `append` in loop → call every time

↳ write code eliminating need to call many functions

[1, 4, 9, 16]

# Constructing New list

## □ List comprehension:

```
squares = [ k*k for k in range(1, n+1) ]
```

## □ Equivalent to :

```
squares = [ ] empty  
for k in range(1, n+1):  
    squares.append(k*k)  
    ↳ add each
```

→ Faster, more efficient

## □ [0] \* n : list of size n

→ Very efficient

# Performance Conclusion

- 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...

# Growable Array-based Array List

- ❑ In an **add(o)** operation (without an index), we could always add at the end
- ❑ When the array is full, we replace the array with a larger one
- ❑ How large should the new array be?
  - **Incremental strategy**: increase the size by a constant  $c$
  - **Doubling strategy**: double the size

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

# Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time  $T(n)$  needed to perform a series of  $n$   $\text{add}(o)$  operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e.,  $T(n)/n$

# Incremental Strategy Analysis

- We replace the array  $k = n/c$  times
- The total time  $T(n)$  of a series of  $n$  add operations is proportional to

$$n + c + 2c + 3c + 4c + \dots + kc =$$

$$n + c(1 + 2 + 3 + \dots + k) =$$

$$n + ck(k + 1)/2$$

- Since  $c$  is a constant,  $T(n)$  is  $O(n + k^2)$ , i.e.,  $O(n^2)$
- The amortized time of an add operation is  $O(n)$

# Doubling Strategy Analysis

- We replace the array  $k = \log_2 n$  times
- The total time  $T(n)$  of a series of  $n$  add operations is proportional to

$$\begin{aligned} n + 1 + 2 + 4 + 8 + \dots + 2^k &= \\ n + 2^{k+1} - 1 &= \\ 3n - 1 \end{aligned}$$

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

geometric series

