Data Structures for Data Science

Grading Policy

Mid-Term:	40
Final:	40
coding Exam:	20

(Part 1)
Books: Data Structures and Algorithms in Python
Goodrich, Tamassia, Goldwasser

(Part 2): Handwritten notes based on other Course Notes

Data Structures for "Computer Science" (DS 4 CS)

$\underbrace{\text{Array, Linked List, Stack, Queue}}_{\text{Linear DS}}$, Tree, Graph

Store : Input Data ⟶ Data Structure ≡ Store / arrange input data intelligently
Phase

Retrieve : Query ⟶ Data Structure ⟶ Retrieve the stored input data (as it is)
Phase

Or

Compute values related to input data

Key Question: How to arrange input data ?

Most of Machine Learning converts data into Model

Store : Training Data ⟶ Model
Phase

Retrieve : Test ⟶ Model ⟶ Output values related to unseen data
Phase New / unseen

Key Question: How to convert data into Model ; i.e., how to "learn" models?

Data structures for "Data Science" (DS4DS)

NOT MUCH CHANGED : $\underbrace{\text{Array, Linked List, Stack, Queue}}_{\text{Linear DS}}$, Tree, Graph

DS4DS

- No need to learn models
- No need to retrieve input data as it is
  $\equiv$ Can throw away input data
- No need to provide answers for unseen data
- Need to provide $\underbrace{\text{approximate statistical summary}}_{\text{Count, Frequency, Quantile}}$ of input data

Tools : Probability

DS4CS ——— DS4DS ——— ML

Data Structures

Manipulation one does to arrange stuff

Data $\xrightarrow{\text{Input}}$ $\boxed{\text{Arrange Data}}$ $\xrightarrow{\text{Query}}$ Retrieval of what we stored

Data Science $\equiv$ Machine Learning

SVM, Perceptron, Deep Neural Net

Training Data $\xrightarrow{\text{Input}}$ $\boxed{\text{Model}}$ $\xrightarrow{\text{Query}}$ Information on new / unseen data

This Course

DS4DS

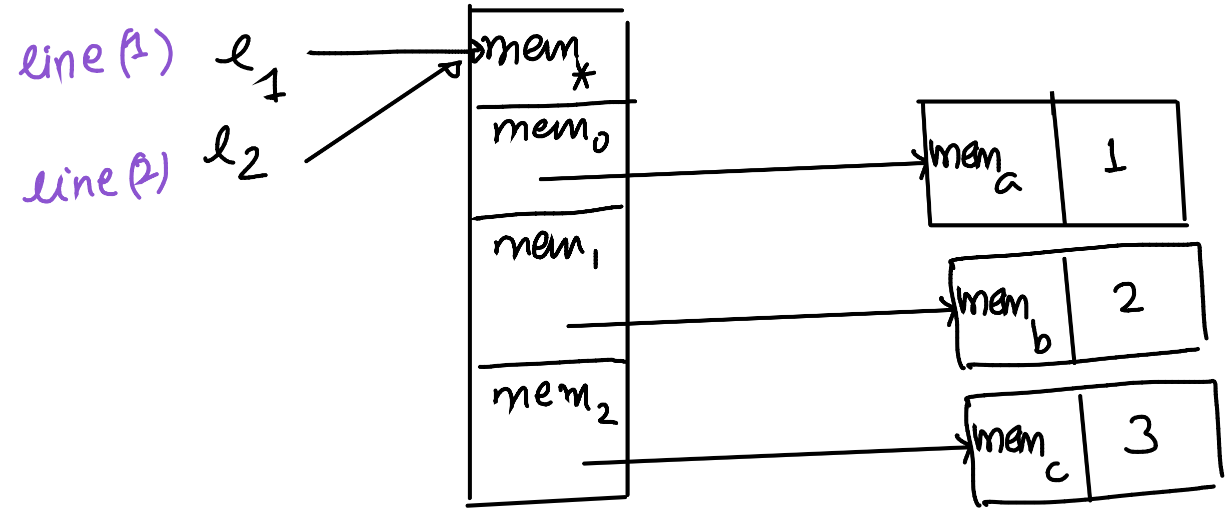
Data Structures — Machine Learning

- We dont need to retrieve data as it is
- We want some approximately correct statistical summaries.

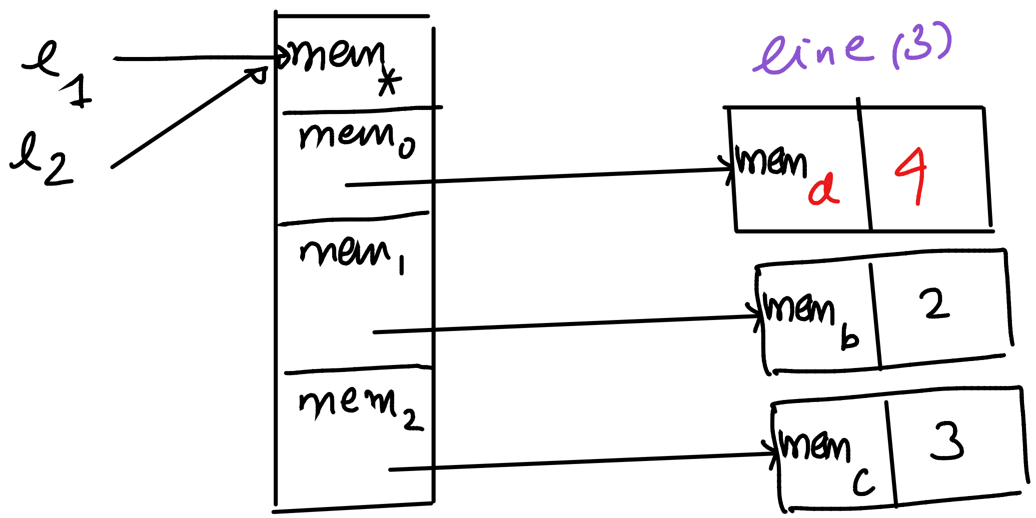
Memory Allocation In Python

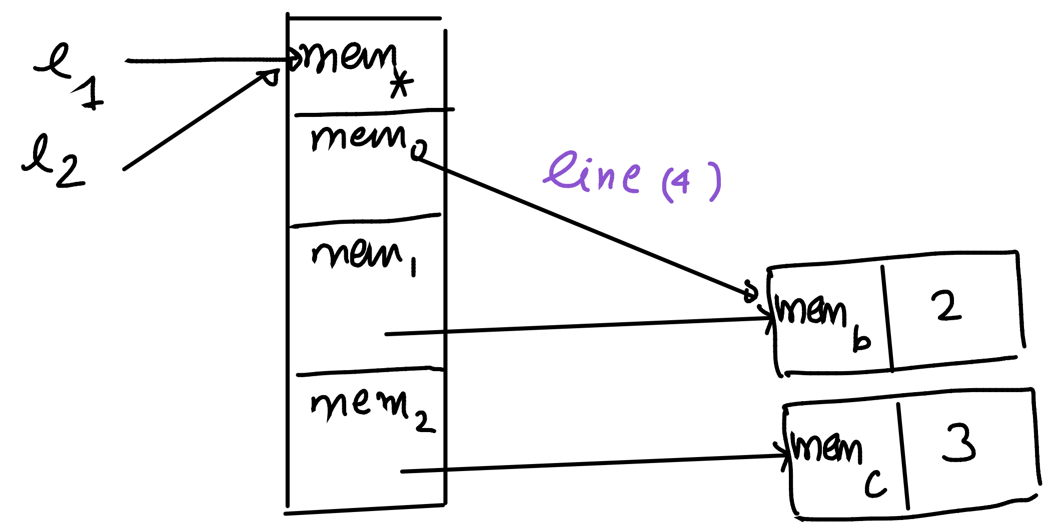
Alias

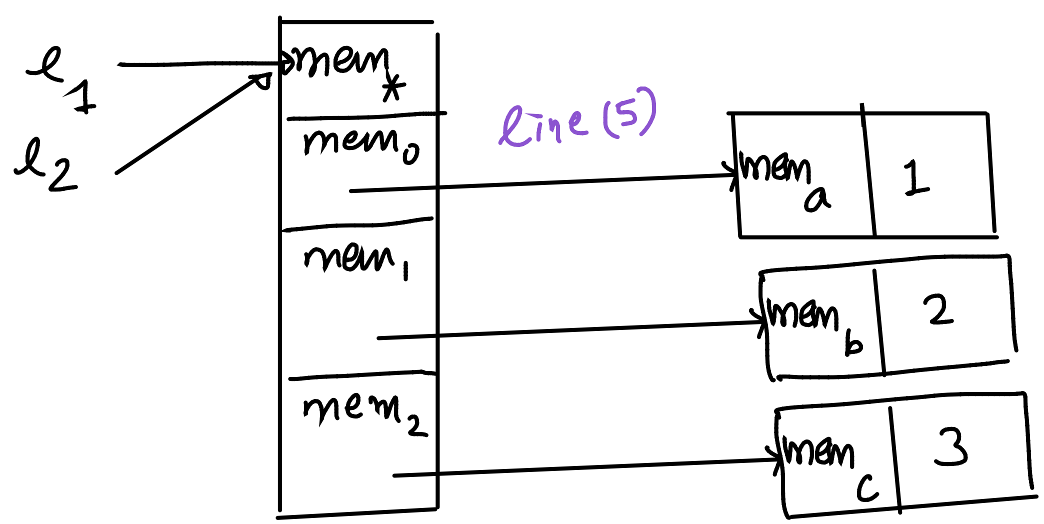
line	
(1)	l1 = [1, 2, 3]
(2)	l2 = l1
(3)	l2[0] = 4
(4)	l1[0] = 2
(5)	l1[0] = 1



mem_0 stores pointer to mem_a







Note: in line 4) and line 5) python is reusing the memory

Shallow Copy

line

(1) $l_1 = [1, 2, 3]$

(2) $l_2 = list(l_1)$

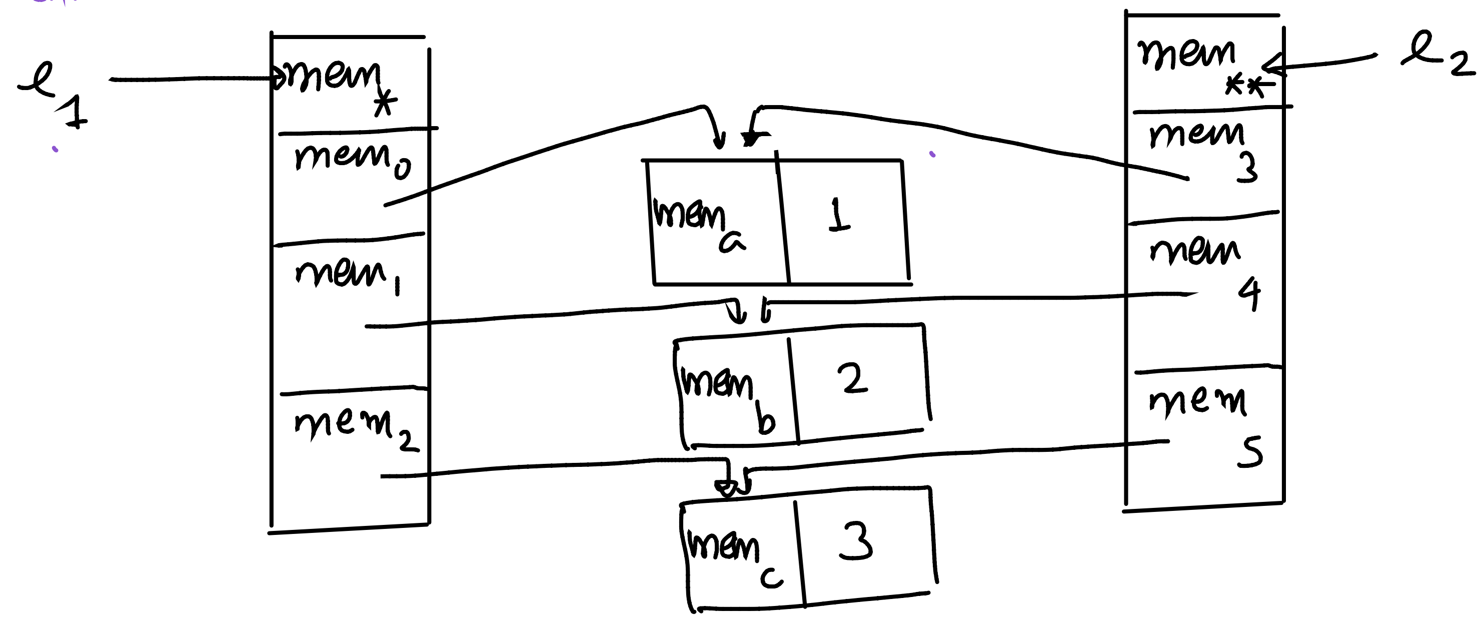
(3) $l_2[0] = 4$

(4) $l_1[0] = 2$

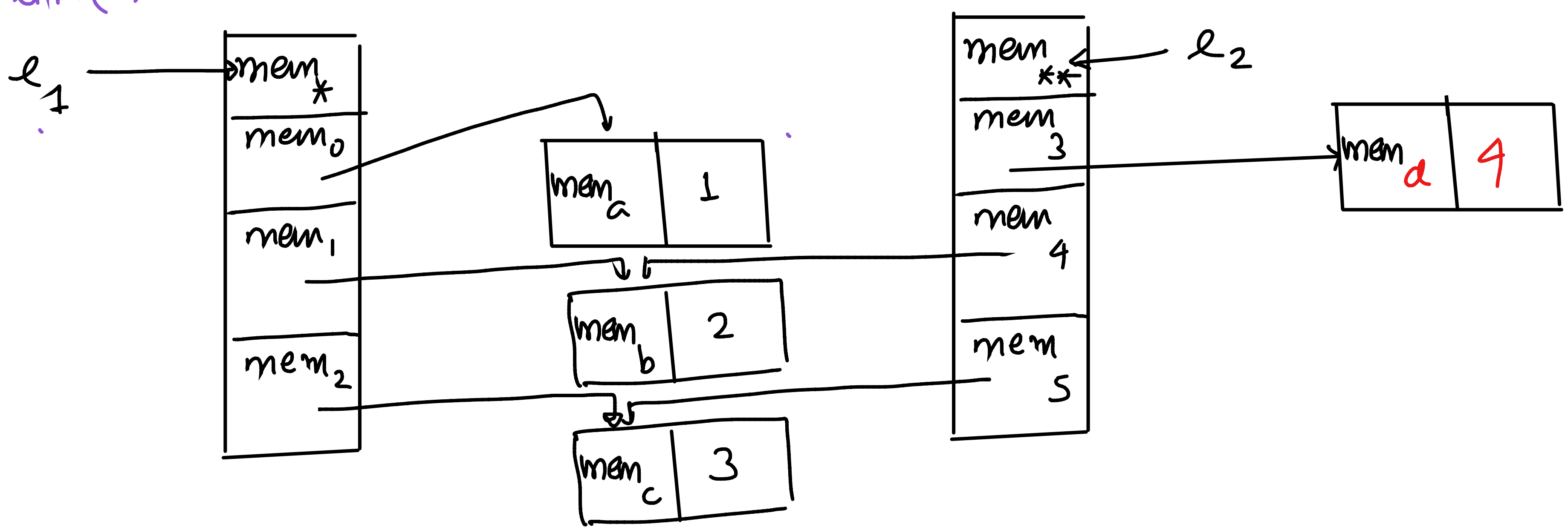
(5) $l_1[0] = 1$

line(1)

line(2)



line(3)



Check what happens for lines (4) and (5)

and similar variants

Shallow copy

line

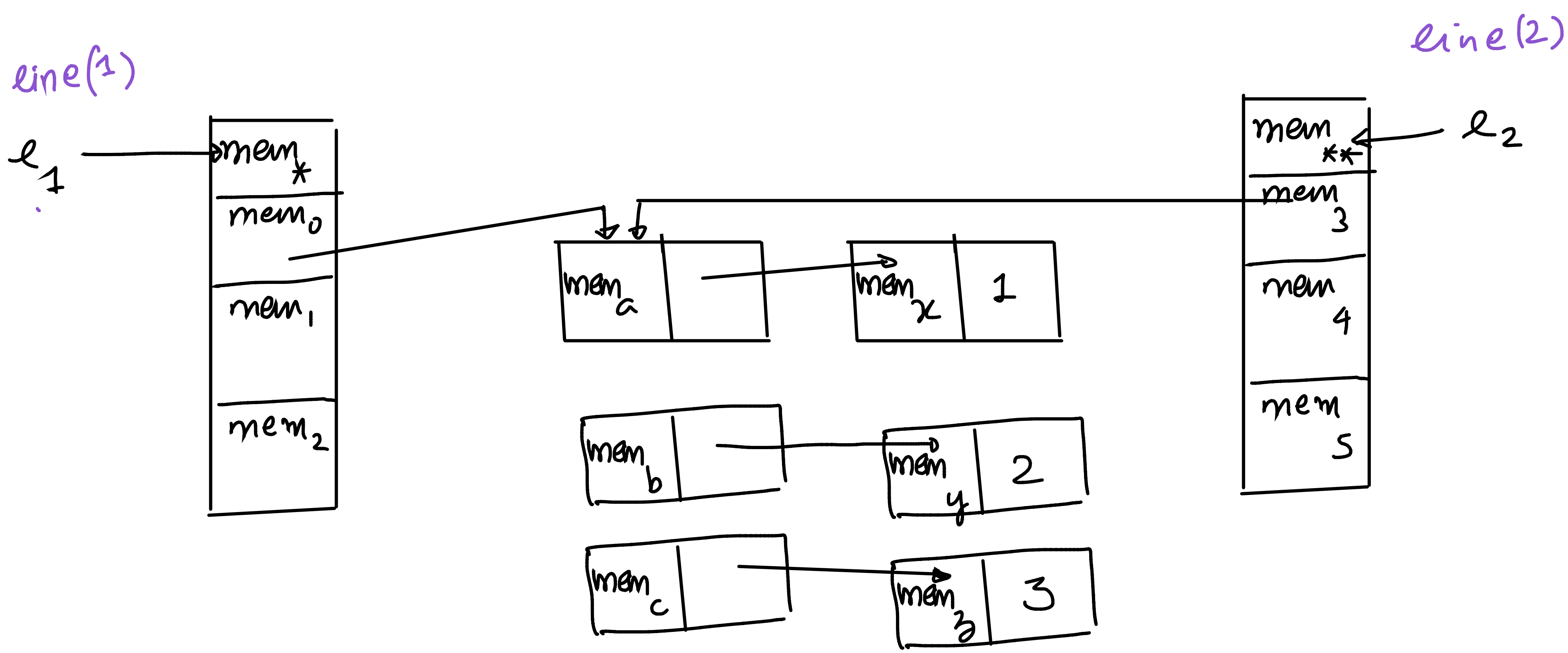
(1) $l_1 = [[1], [2], [3]]$

(2) $l_2 = list(l_1)$

(3) $l_1[0] = 4$

(4) $l_1[0] = 2$

(5) $l_1[0] = 1$



Shallow copy

line

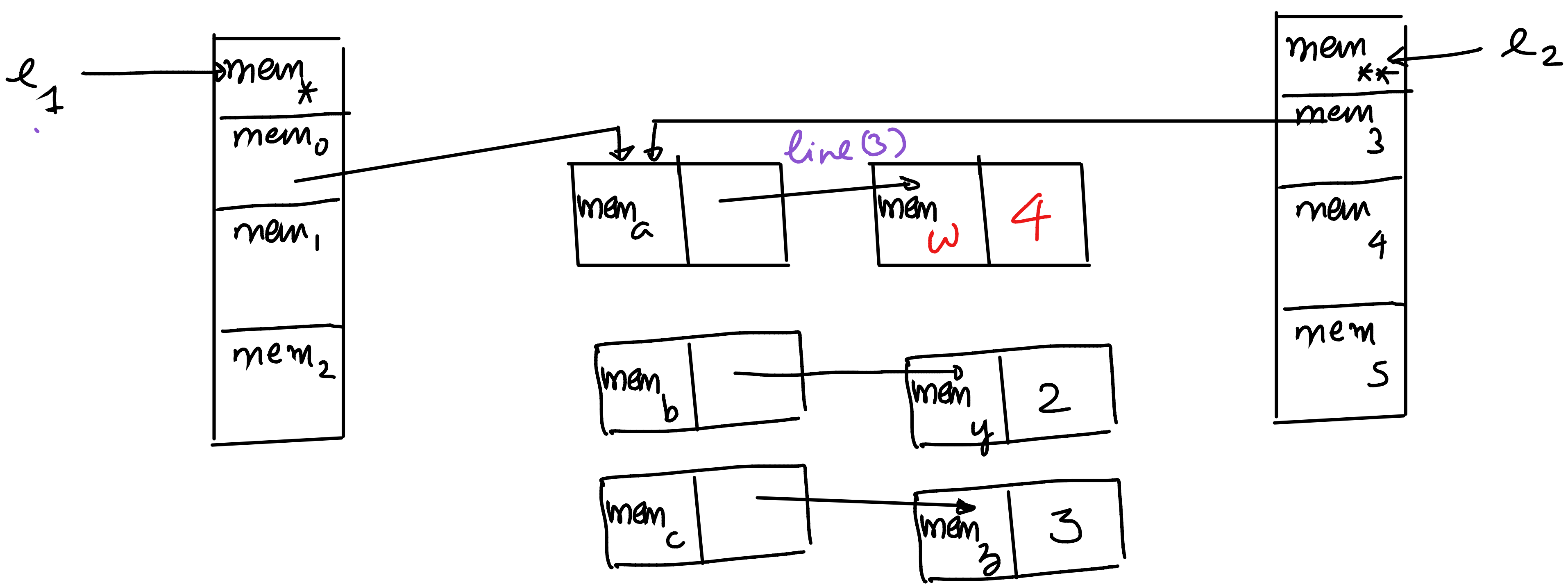
(1) $l_1 = [[1],[2],[3]]$

(2) $l_2 = list(l_1)$

(3) $l_1[0] = 4$

(4) $l_1[0] = 2$

(5) $l_1[0] = 1$

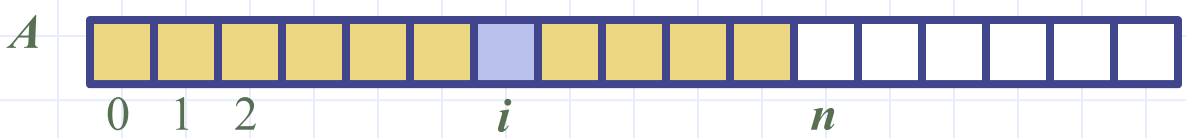


Check what happens for lines (4) and (5) and similar variants

Array-Based Sequences

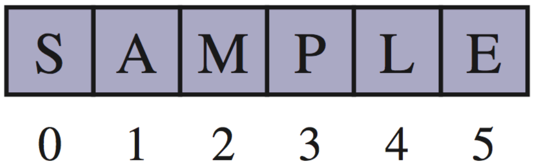
Python Sequence Classes

- 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.

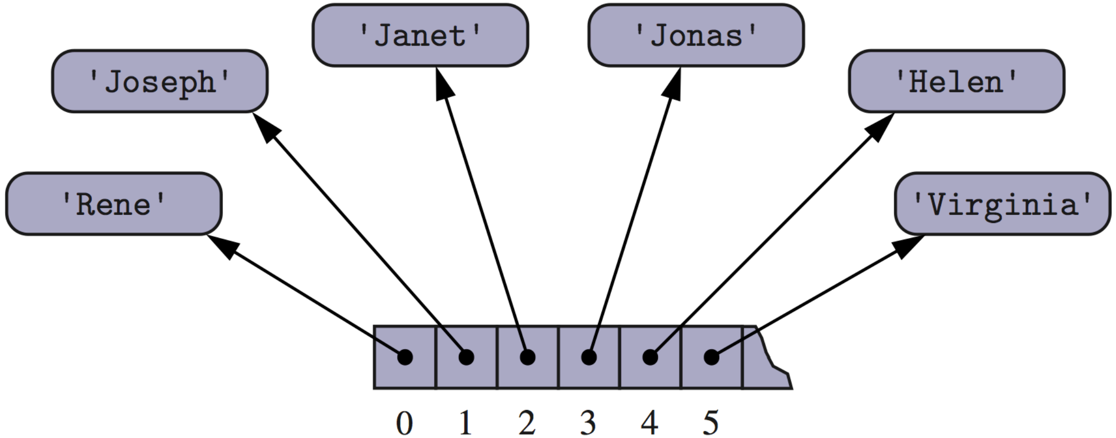


Arrays of Characters or Object References

- An array can store primitive elements, such as characters, giving us a **compact array.**



- An array can also store references to objects.



Compact Arrays

- 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.
- 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.

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

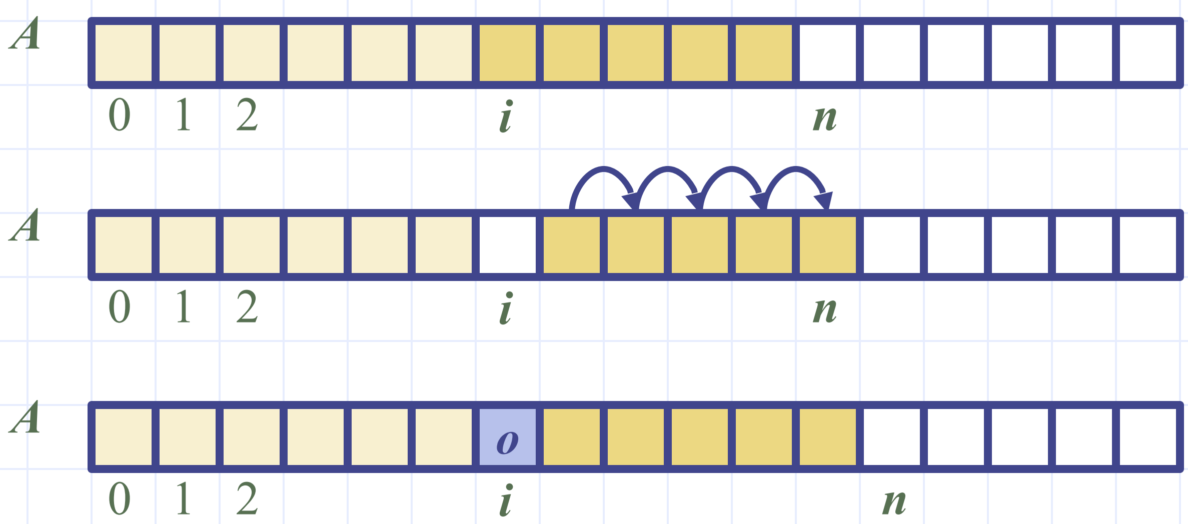
Type Codes in the array Class

- 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

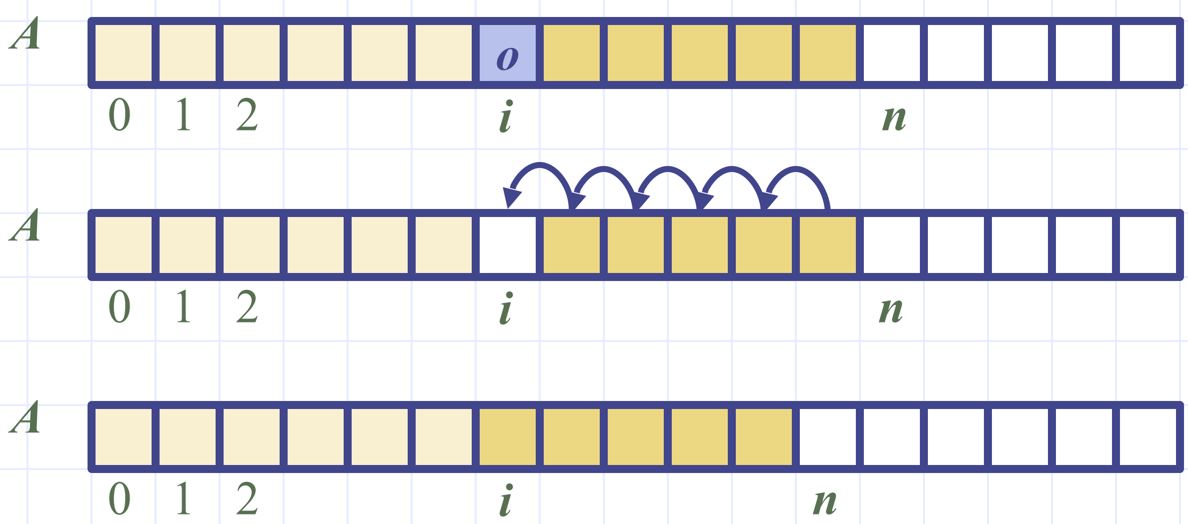
Insertion

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



Element Removal

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



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…

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 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

2nd overflow

'n' add operations'

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

1st overflow

kth overflow

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

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

- 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

2nd overflow

'n' add operations'

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$

$$n + 2^{k+1} - 1 =$$

1st overflow

kth overflow

$$3n - 1$$

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

geometric series