INDUSTRIAL
UNIVERSITY OF
HOCHIMINH CITY

Array-Based Sequences

Data Structures and Algorithms with Python

Lecture 5

INNOVATION - UNITY - HUMANITY

www.iuh.edu.vn

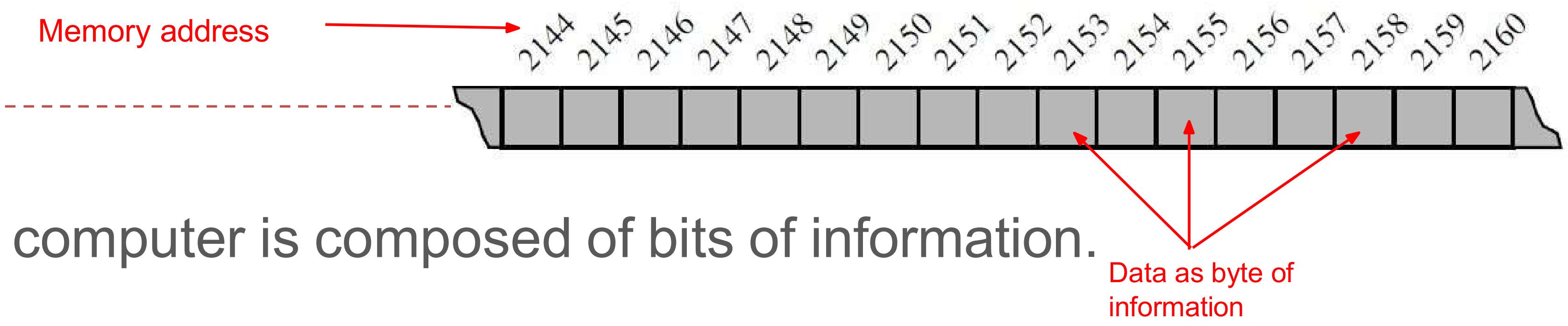
Overview

- Python Sequence Types
- Low-level arrays
  - Referential Arrays
  - Compact Arrays
- Dynamic Arrays and Amortization
  - Implementing a dynamic array
  - Amortized analysis of dynamic array
  - Python’s list class
- Efficiency of Python’s sequence types
  - List and Tuple classes
  - String class
- Using Array-based sequences
  - Storing High scores for a game
  - Sorting a sequence
  - Simple Cryptography
- Multidimensional datasets

Python’s Sequence Types

- Sequence classes: ***list***, ***tuple*** and ***str***
- Common behaviour:
  - Each support indexing to access an individual element of a sequence: e.g seq[k]
  - Each use a low-level concept known as an ***array*** to represent the sequence.
- Difference lies in
  - The abstraction these classes represent and how these complex data structures are implemented.
- A deeper understanding of these data structures is important for a good programmer.

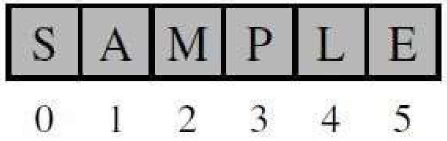
Low-level Arrays

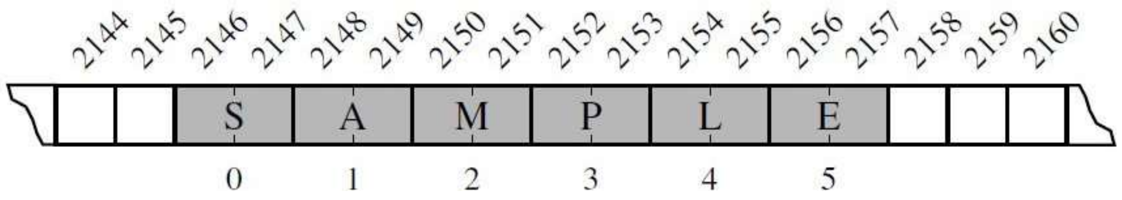


- The primary memory of a computer is composed of bits of information.
- Those bits are typically grouped into larger units that depend upon the precise system architecture. Such a typical unit is a ***byte***, which is equivalent to 8 bits.
- Each byte of information in the memory is associated with a unique number that serves as its address - ***memory address*****.**
- **Random Access Memory (RAM):** Despite the sequential layout of memory locations, computer hardware is designed to access any memory location efficiently using its memory address. That is, it is just as easy to retrieve byte #8675309 as it is to retrieve byte #309.
- Any individual byte of memory can be stored or retrieved in O(1) time.

- An ***array*** is a group of related variables stored one after the another in a contiguous portion of the computer’s memory.
  Example: A text string is stored as an ordered sequence of individual characters.

- Each cell of an array must use the same number of bytes. This allows an arbitrary cell to be accessed in constant time.





- Each unicode character is represented by 16 bits or 2 bytes.

- A six-character string such as ‘SAMPLE’ would be stored in 12 consecutive bytes of memory.

- This is an array of 6 characters and requires 12 bytes of memory.

- Each location within the array is called a **cell** and an integer **index** will be used to describe the cell location in the array.

- As example cell with index 4 contains ‘L’ that is stored in memory locations 2154 and 2155 respectively.

- Address of any cell with index k = start + cellsize * k

Referential Arrays

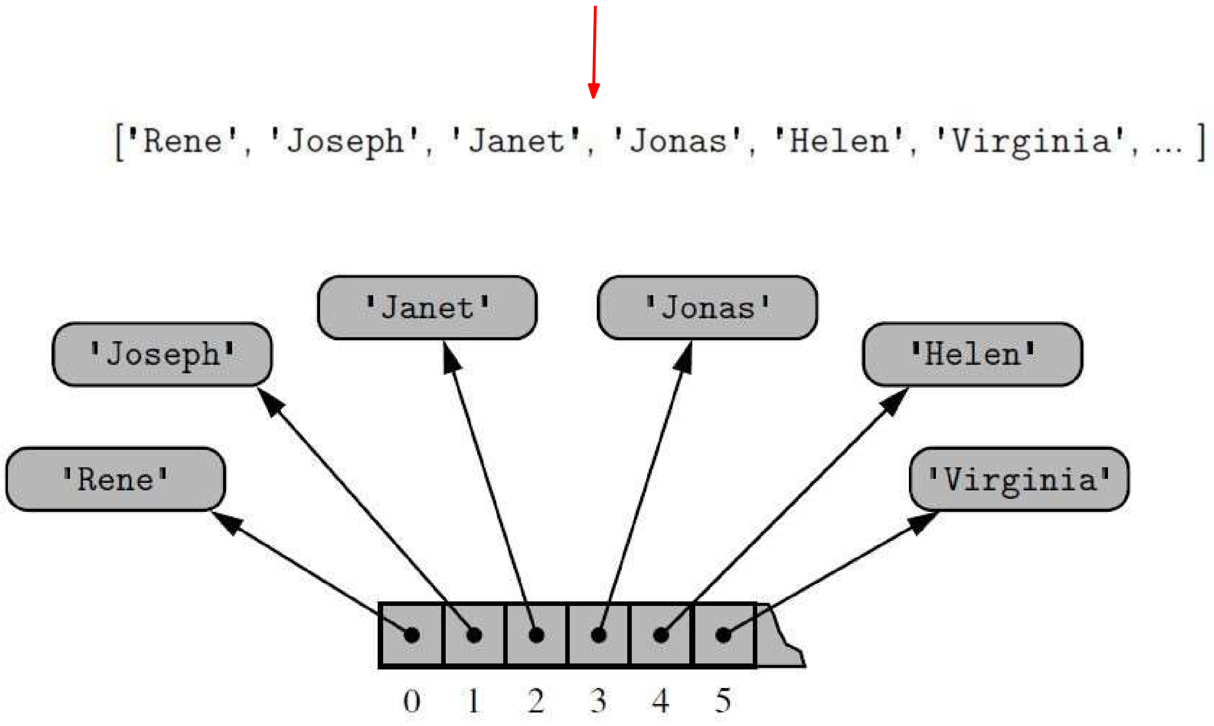
- Python represents a list or tuple instance by using an internal storage mechanism of an array of object ***references***.

- So, at the lowest level, the array consists of a sequence of memory addresses where the elements of the sequence reside.

- Each memory address is of constant size and hence, python can provide ***constant-time access*** to a list or tuple element based on its index.

- The list or tuple will simply keep a sequence of references to the objects to be stored.

Consider a list of 'strings' of different sizes



['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', 'Virginia', ...]
'Janet'
'Jonas'
'Joseph'
'Helen'
'Rene'
'Virginia'
0
1
2
3
4
5

An array storing references to strings.

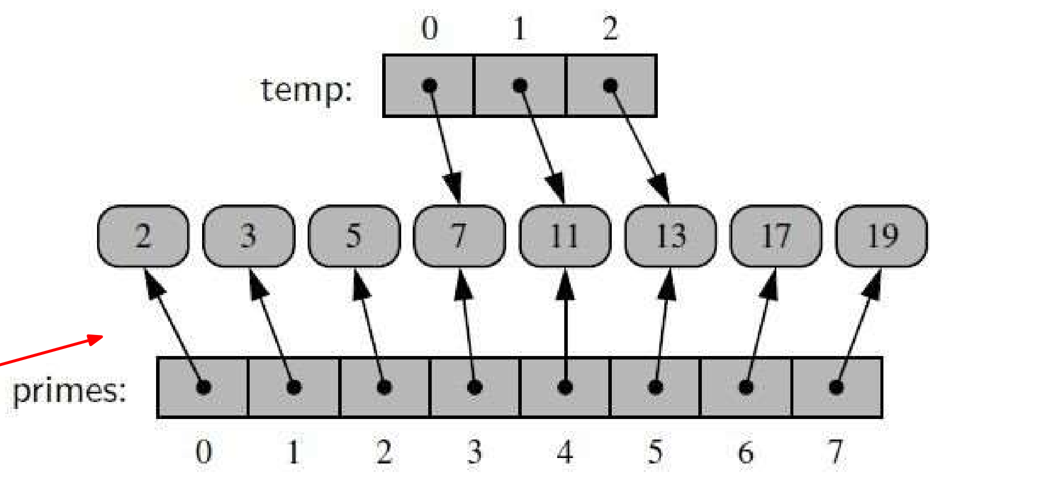
- A single list instance may include **multiple references to the same object** as elements of the list.

- It is also possible for **a single object to be an element of two or more lists**, as those lists simply store references back to that object.

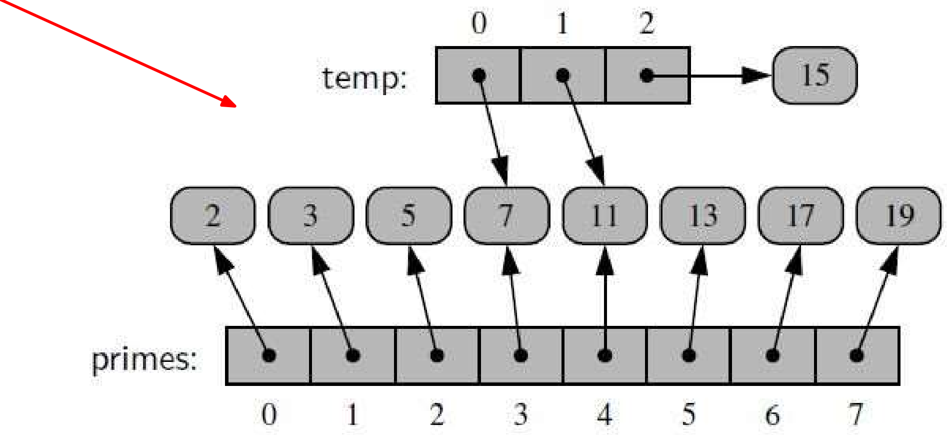
- `temp[2] = 15` only changes the references to an immutable integer object '15'.

- `backup = list(primes)` creates a new list that contains the same references as that present in primes (**shallow copy**).

- A **deep copy** should be used for copying a mutable list.



The result of the command temp = primes[3:6]



The result of the command temp[2] = 15

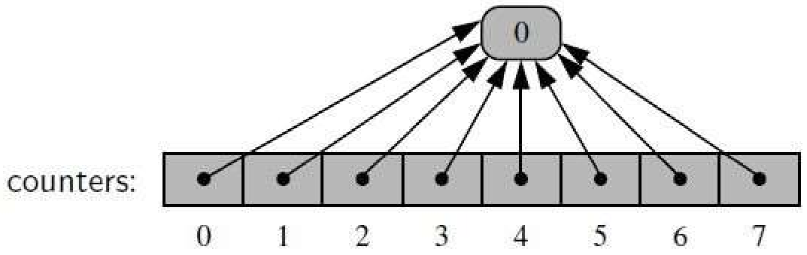
produces a list of

- `counters = [0]*8`
  references to the same element ‘0’ which is an immutable object.

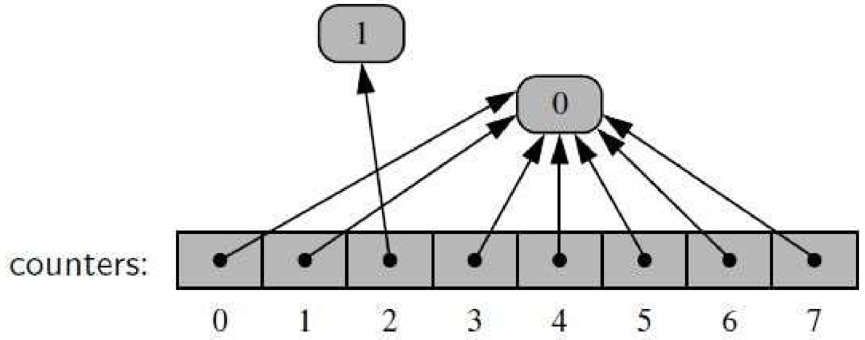
- `counters[2] += 1` creates a new integer with value 0+1 and sets its reference to the cell 2 of the array.

- Extending a list - receives references to newly added elements.

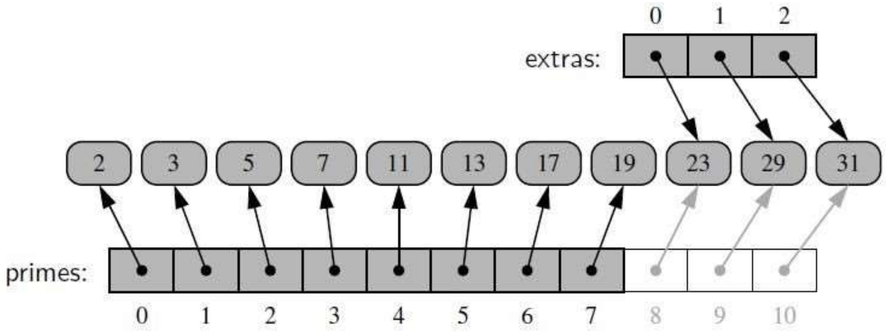
  ```
  extras = [23, 29, 31]
  primes.extend(extras)
  ```



The result of the command data = [0] ∗ 8.



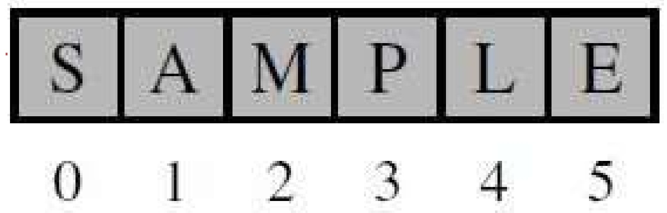
The result of command data[2] += 1



The effect of command primes.extend(extras)

Compact Arrays in Python

- A ***compact array*** stores primary data (in the form of bits) instead of their references. Example - strings.



S	A	M	P	L	E
0	1	2	3	4	5

```
> import array
> primes = array( ‘i’ , [2, 3, 5, 7, 11])
```

- Advantages of Compact arrays:
  - Overall memory usage will be much lower compared to referential arrays - no overhead for storing references.
  - Primary data is stored consecutively in memory. This provides higher computing performance.

- Primary support for compact arrays is in a module named `array`.

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

Type codes supported by the array module.

Dynamic Arrays

- When creating low-level arrays, it is necessary to specify the precise size of that array explicitly during instantiation.

- The same is applicable to Python's `tuple` and `str` instance which are immutable - Once created, its size or content can not altered.

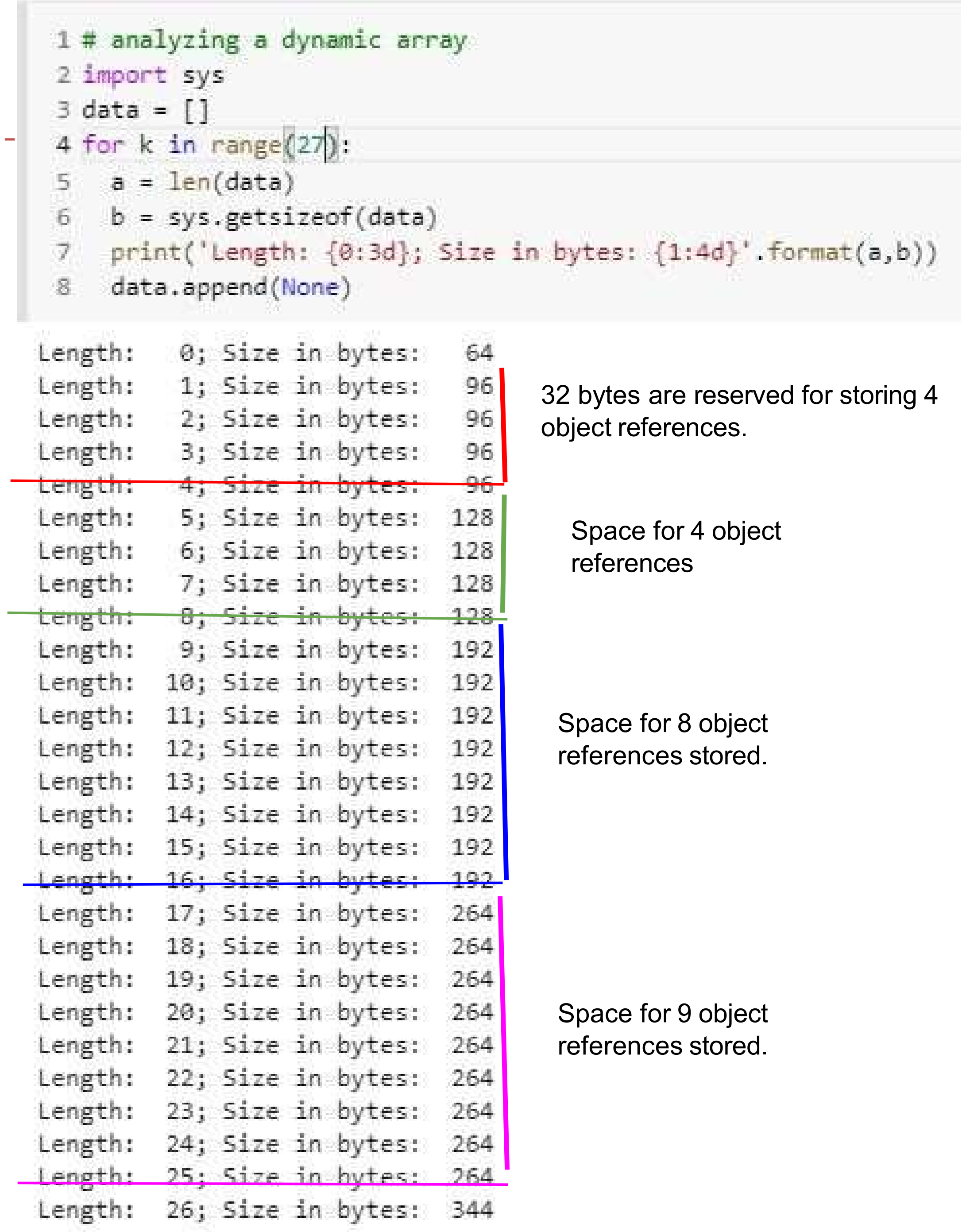
- Python's `list` class represents a ***dynamic array*** where elements could be added to or removed from the list.

- Semantics of Dynamic array
  - It maintains an underlying array that has greater capacity than the current length of the list.
  - If the reserved capacity is exhausted, the system initializes a new array larger in size and the older array is then reclaimed by the system.

- An empty list requires some space (64 bytes on this machine). It is required for keeping following information:
    - Number of actual elements currently stored in the list.
    - The maximum number of elements that could be stored.
    - The reference to the currently allocated array.

- Each memory address is a 64-bit (8 byte) number.

- Reserve capacity increases 32 bytes (4 cells) to 64 (8 cells) and so on as the array is extended continuously by adding new objects.



```python
# analyzing a dynamic array
import sys
data = []
for k in range(27):
    a = len(data)
    b = sys.getsizeof(data)
    print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a,b))
    data.append(None)
```

```
Length:   0; Size in bytes:   64
Length:   1; Size in bytes:   96
Length:   2; Size in bytes:   96
Length:   3; Size in bytes:   96
Length:   4; Size in bytes:   96
Length:   5; Size in bytes:  128
Length:   6; Size in bytes:  128
Length:   7; Size in bytes:  128
Length:   8; Size in bytes:  128
Length:   9; Size in bytes:  192
Length:  10; Size in bytes:  192
Length:  11; Size in bytes:  192
Length:  12; Size in bytes:  192
Length:  13; Size in bytes:  192
Length:  14; Size in bytes:  192
Length:  15; Size in bytes:  192
Length:  16; Size in bytes:  192
Length:  17; Size in bytes:  264
Length:  18; Size in bytes:  264
Length:  19; Size in bytes:  264
Length:  20; Size in bytes:  264
Length:  21; Size in bytes:  264
Length:  22; Size in bytes:  264
Length:  23; Size in bytes:  264
Length:  24; Size in bytes:  264
Length:  25; Size in bytes:  264
Length:  26; Size in bytes:  344
```

32 bytes are reserved for storing 4 object references.

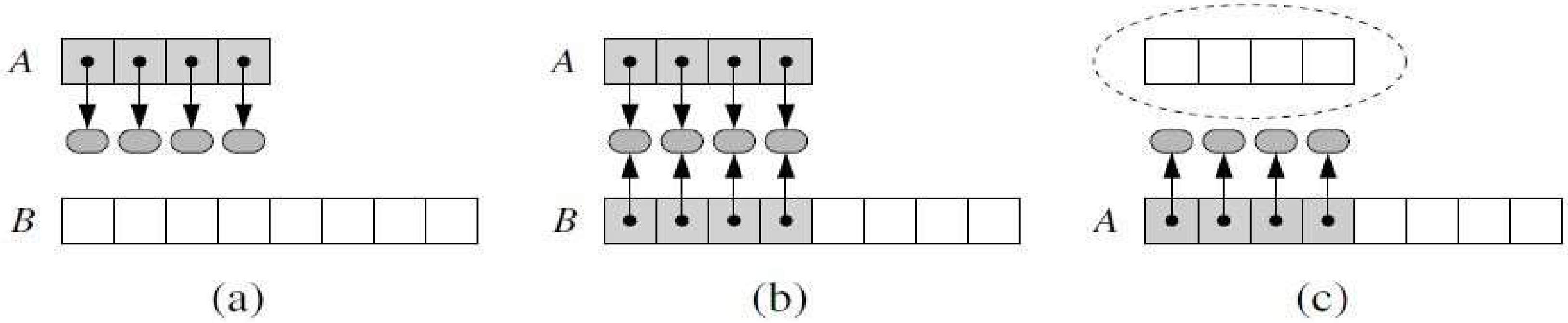
Space for 4 object references

Space for 8 object references stored.

Space for 9 object references stored.

Implementing a dynamic array

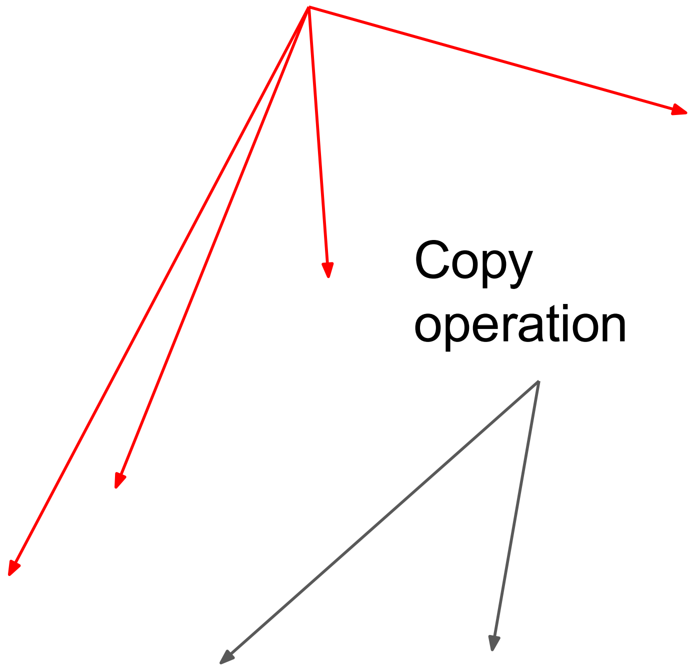
- The key is to provide a means to grow the array A that stores the elements of a list.
- If an element is appended to a list at a time when the underlying array is full, we perform the following steps:
  - Allocate a new array B with larger capacity.
  - Set $B[i] = A[i], \text{ for } i = 0, \cdots, n-1$ where n denotes the current number of items.
  - Set A = B, i.e, we use B as the array supporting the list.
  - Insert the new element in the new array.



(a) (b) (c)

- How large of a new array to create?
  - Common rule - the new array to have twice the capacity of the existing array that is filled.

Creation of new array + copying



Copy operation

Run-time of a series of append operations on a dynamic array

Amortized Analysis

- ***Amortized analysis*** refers to determining the time-averaged running time for a sequence of operations.

- It is different from average-case analysis as it does not make any assumption about the distribution of data values.

- Amortized analysis is a worst-case analysis for a sequence of operations rather than for individual operations.

- The motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst-case analysis provides an overly pessimistic bound.

- Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that expensive operations are rare, we can change them to the cheap operations, and only bound the cheap operations.

- Assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds the total of the real costs of the sequence.

- This artificial cost is called the ***amortized cost*** of an operation.

Amortized Analysis of Dynamic Arrays

- View the computer as a coin-operated appliance that requires the payment of 1 cyber-dollar for a constant amount of computing time.

- When an operation is executed, we should have enough cyber-dollars available in our current bank account for that operation’s running time.

- Thus the total amount of cyber-dollars spent for any computation will be proportional to the total time spent on that computation.

- We can overcharge some operations in order to save up cyber-dollars to pay for others.

Proposition 5.1: *Let S be a sequence implemented by means of a dynamic array with initial capacity one, using the strategy of doubling the array size when full. The total time to perform a series of n append operations in S, starting from S being empty, is O(n).*

Justification:

- Let’s assume each append operation in S costs 1 cyber-dollar excluding time spent for growing the array.

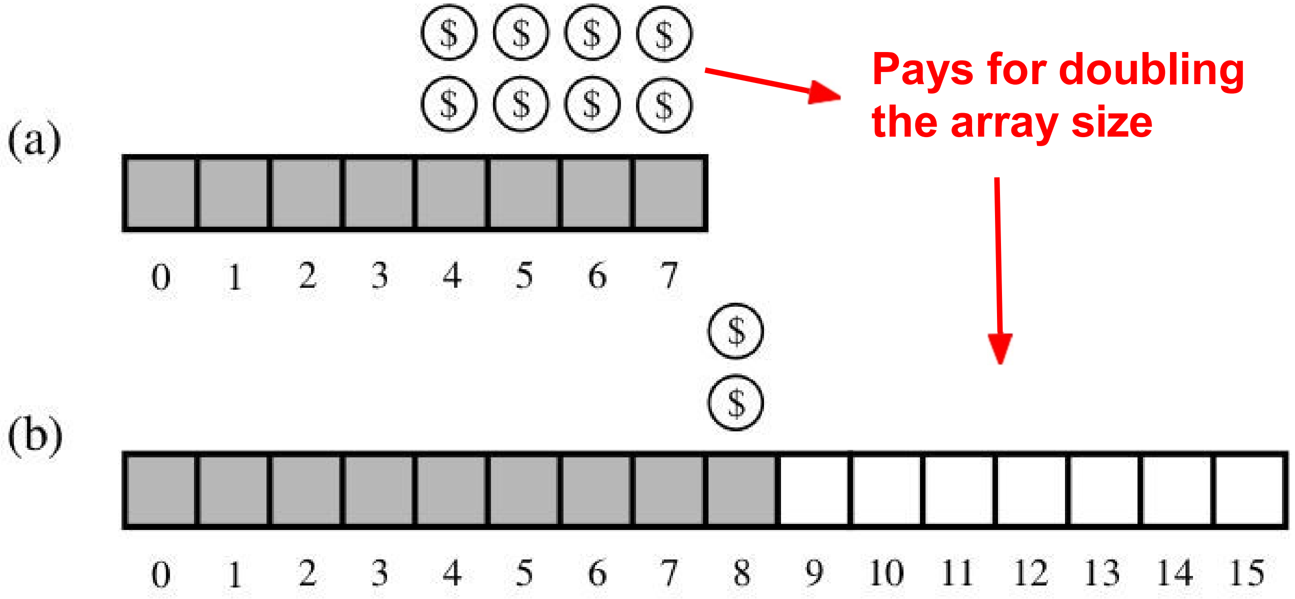
- Let’s assume growing the array from size k to size 2k requires k cyber-dollars.

- Let’s charge 3 cyber-dollars for each append operation that does not cause an overflow.

- There is a profit of 2 cyber-dollars for each insertion that does not grow the array and is stored in the cell where an element is inserted.

- An array overflow occurs when S has $2^i, i \geq 0$ elements.

- Doubling the size of the array will require 2^i cyber-dollars.
- These cyber-dollars are stored in cells 2^{i-1} and $2^i - 1$.
- This is a valid amortization scheme in which each operation is charged 3 cyber-dollars and all computing time is paid for.
- Hence for n append operations, we need 3n cyber-dollars.
- The amortized running time of each append operation is O(1).
- Hence, the total running time of n append operations is O(n).



$$i = 3$$

$$2^i = 8,$$

$$2^{i-1} = 4$$

$$2^i - 1 = 7$$

- Geometric Increase in capacity
  - The O(1) amortized bound per operation can be proven for any geometrically increasing progression of array sizes.

- Beware of Arithmetic Progression:
  - Reserving a constant number of additional cells during array resize leads to poor performance.
  - At an extreme, adding one cell with each append operation leads to a quadratic overall cost: $1 + 2 + \cdots + n \sim \Omega(n^2)$
  - Using increase of 2 or 3 cells at a time is slightly better but still leads to quadratic overall cost.
  - Using a fixed increment for each resize - arithmetic progression of intermediate array sizes, results in overall time that is quadratic in the number of operations.

Increasing array size by 3 Cells.

Increasing array size by 2 Cells.

Proposition 5.2: Performing a series of n append operations on an initially empty dynamic array using a fixed increment with each resize takes Ω(n^2) time.

Justification:

- Let $c > 0$ represent the fixed increment in capacity that is used for each resize event.
- For a series of n append operations, there will be $m = \left\lceil \frac{n}{c} \right\rceil$ resize operations taking $c, 2c, 3c, \cdots, mc$ time (number of operations).
- Overall time will be proportional to the sum $c + 2c + 3c + \cdots + mc$:

$$t \propto c\sum_{i=1}^{m} i = c\frac{m(m+1)}{2} \geq c\frac{\frac{n}{c}(\frac{n}{c}+1)}{2} \geq \frac{n^2}{2c}$$

$$t \sim \Omega(n^2)$$

Memory Usage and Shrinking an Array

- Another consequence of the rule of geometric increase in capacity when appending to a dynamic array is that the final array size is guaranteed to be proportional to the overall number of elements. That is, the data structure uses O(n) memory.

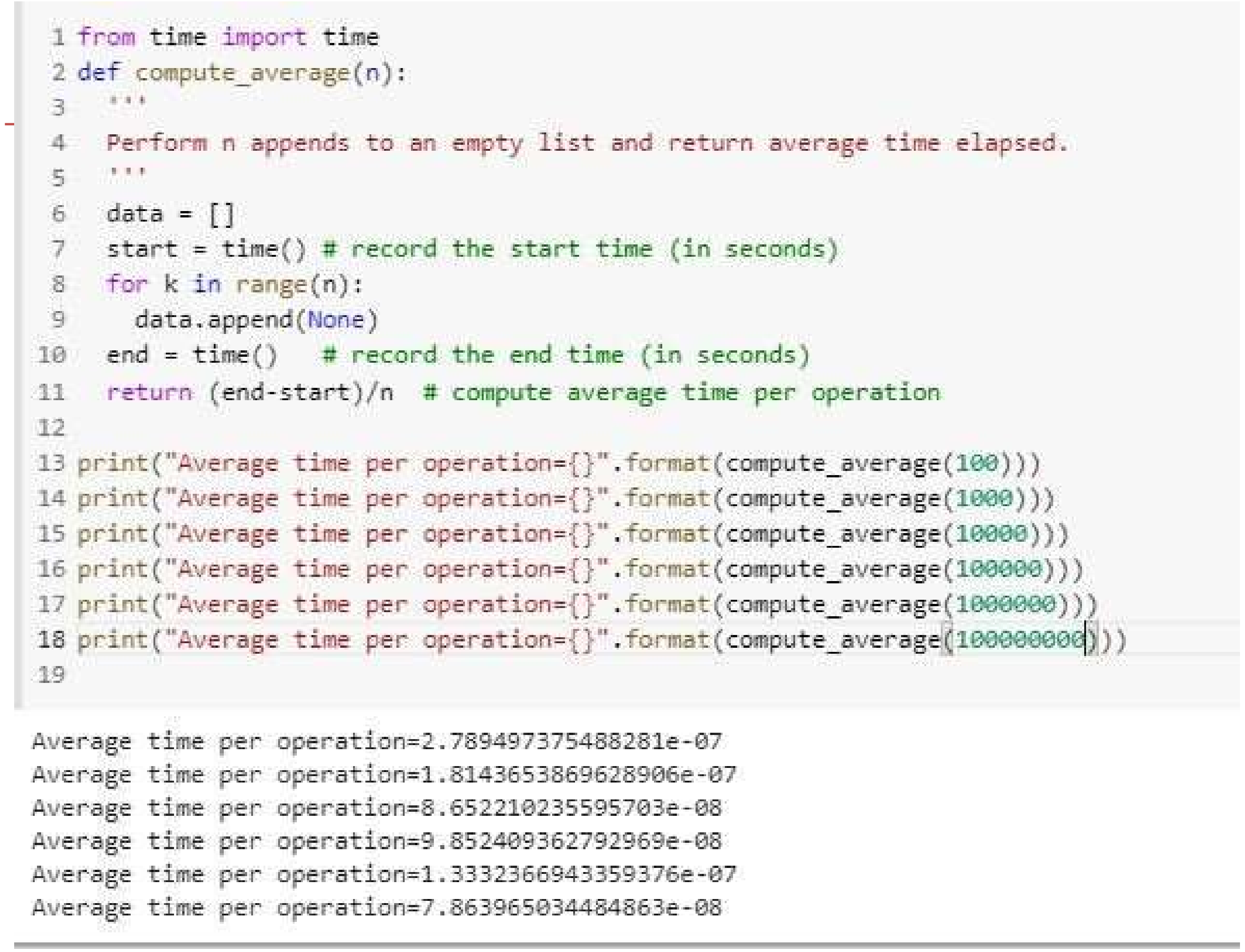
- If a container, such as a Python list, provides operations that cause the removal of one or more elements, greater care must be taken to ensure that a dynamic array guarantees O(n) memory usage.

- Repeated insertions may cause the underlying to grow arbitrarily large which will no longer be a proportional relationship between the actual number of elements.

- A robust implementation of such a data structure will shrink the underlying array, on occasion, while maintaining O(1) amortized bound on individual operations.

Python’s List class

- Python’s implementation of the append method exhibits amortized constant-time behaviour.



```python
from time import time
def compute_average(n):
  '''
  Perform n appends to an empty list and return average time elapsed.
  '''
  data = []
  start = time() # record the start time (in seconds)
  for k in range(n):
    data.append(None)
  end = time()   # record the end time (in seconds)
  return (end-start)/n  # compute average time per operation

print("Average time per operation={}".format(compute_average(100)))
print("Average time per operation={}".format(compute_average(1000)))
print("Average time per operation={}".format(compute_average(10000)))
print("Average time per operation={}".format(compute_average(100000)))
print("Average time per operation={}".format(compute_average(1000000)))
print("Average time per operation={}".format(compute_average(100000000)))
```

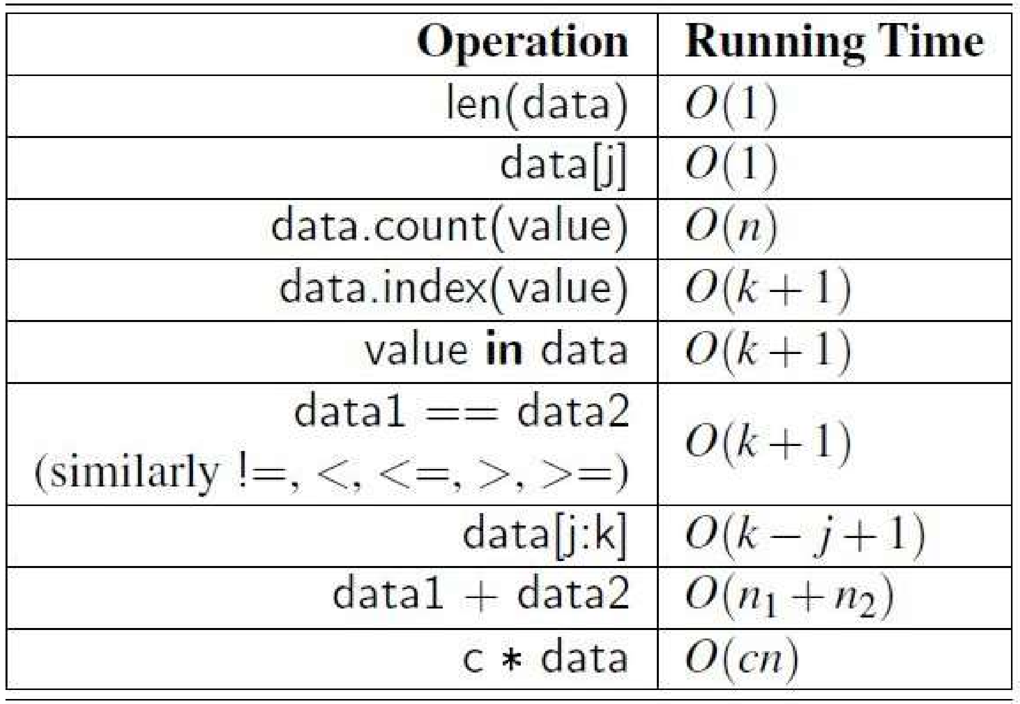
```
Average time per operation=2.789497375488281e-07
Average time per operation=1.8143653869628906e-07
Average time per operation=8.652210235595703e-08
Average time per operation=9.852409362792969e-08
Average time per operation=1.3332366943359376e-07
Average time per operation=7.863965034484863e-08
```

- As we can see the amortized time for each append is independent of n.

n	100	1000	10,000	100,000	1,000,000	100,000,000
µs	0.278	0.181	0.087	0.099	0.133	0.079

Efficiency of Python’s Sequence Types

- Tuples are typically more memory efficient than lists because they are immutable.
- Tuple class implements non-mutating behaviour of list class.
- Constant-time operations:
  - len(data) and data[j]
- Searching for occurrence of a value
  - Computing count requires looping over all elements in the array
  - The loop for Checking containment of an element or determining index of an element immediately exit once they find the leftmost occurrence of desired value.
  - Lexicographic comparisons - in the worst case, evaluation requires an iteration time proportional to the length of the shorter of two sequences
  - Creating a new instances (last 3) - running time depends on the length of new array size to be created.

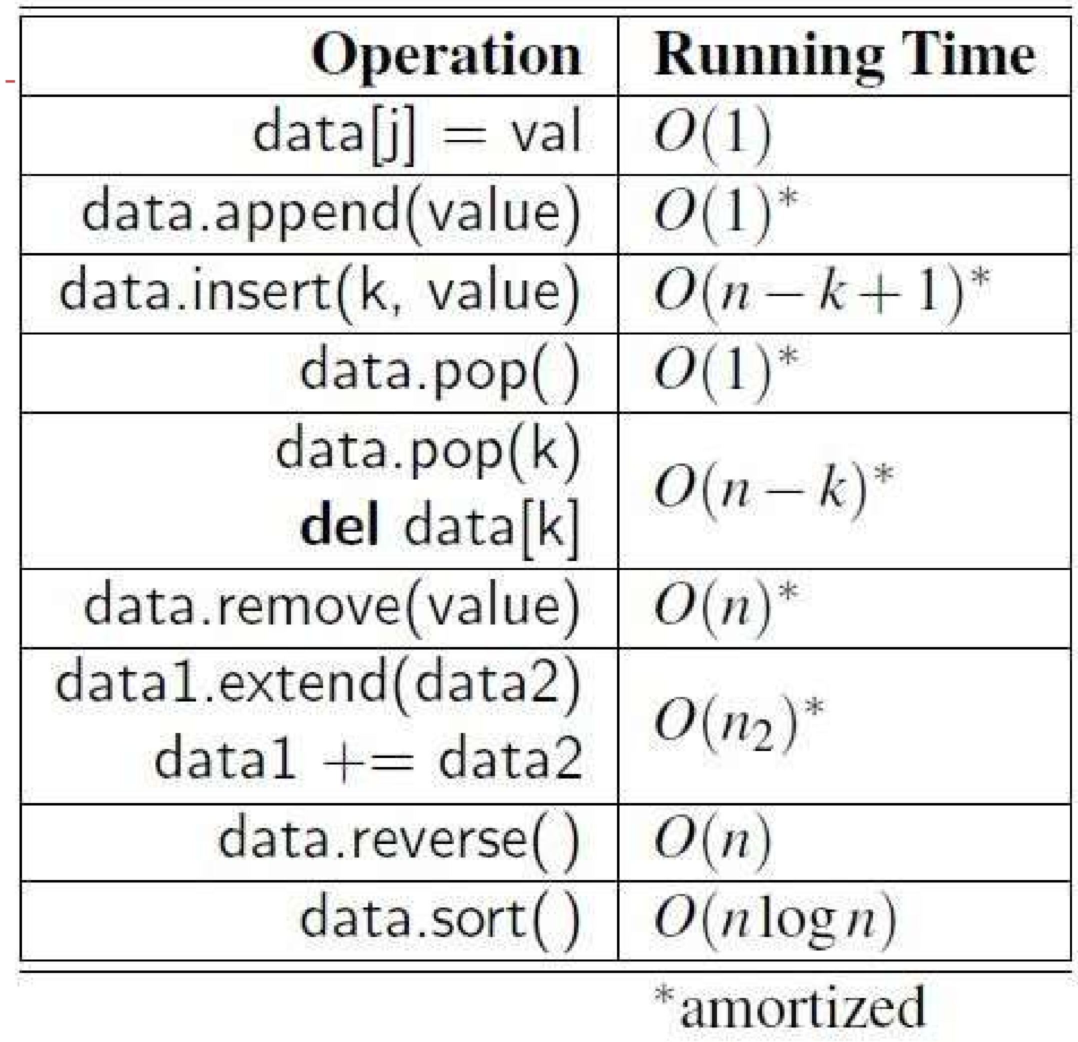


Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Asymptotic performance of the **non-mutating behaviors** of the list and tuple classes.

Mutating Behaviours

- `__setitem__()` method requires constant time.
- Adding element to a list
- Removing element from a list
- Extending a list
- Constructing new 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
	$O(n_2)^*$

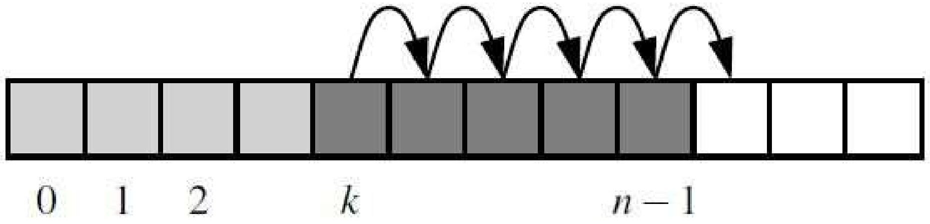
data.reverse()	$O(n)$
data.sort()	$O(n \log n)$

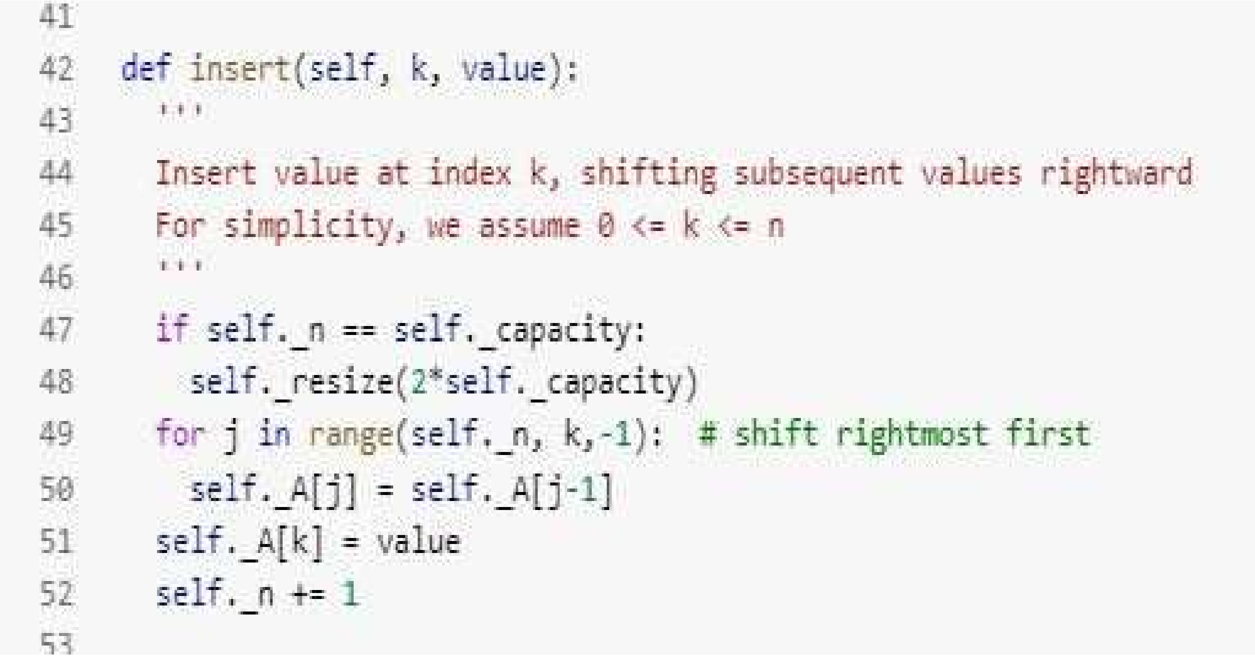
*amortized

Asymptotic performance of the **mutating behaviors** of the list class

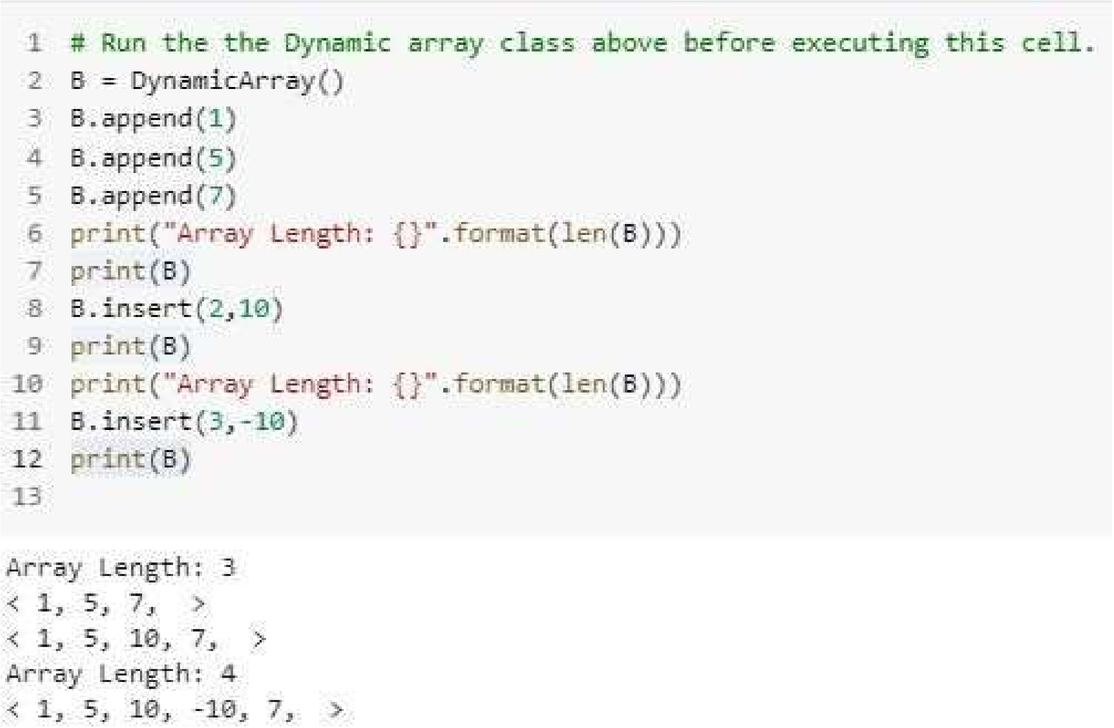
Adding element to a list

- `insert(k, value)`, 0 <= k <= n requires shifting all subsequent elements back one slot to make room.
  - ○ Array resizing with a worst-case run-time of $\sim\Omega(n)$ but only O(1) amortized time per operation.
  - ○ Shifting of elements to make room for the new item with amortized $O(n-k+1)$ performance for inserting at index k.





```
def insert(self, k, value):
  '''
  Insert value at index k, shifting subsequent values rightward
  For simplicity, we assume 0 <= k <= n
  '''
  if self._n == self._capacity:
    self._resize(2*self._capacity)
  for j in range(self._n, k,-1):  # shift rightmost first
    self._A[j] = self._A[j-1]
  self._A[k] = value
  self._n += 1
```



```
# Run the the Dynamic array class above before executing this cell.
B = DynamicArray()
B.append(1)
B.append(5)
B.append(7)
print("Array Length: {}".format(len(B)))
print(B)
B.insert(2,10)
print(B)
print("Array Length: {}".format(len(B)))
B.insert(3,-10)
print(B)
```

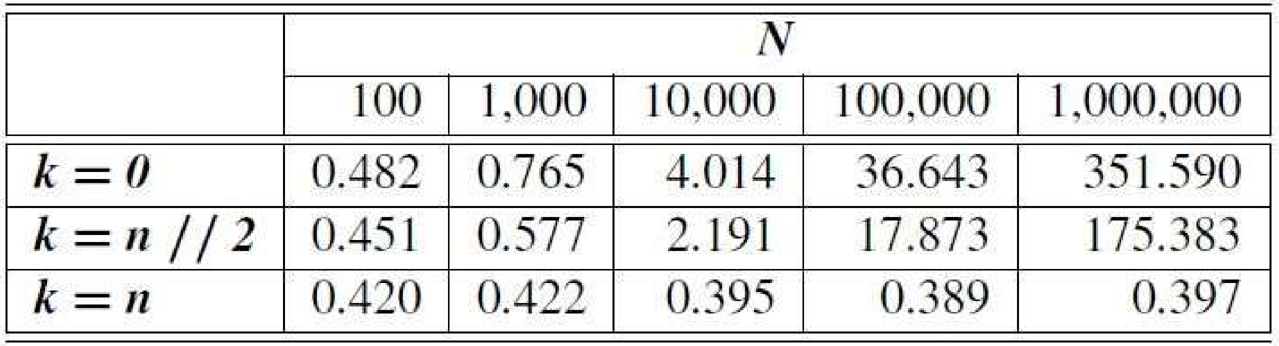
```
Array Length: 3
< 1, 5, 7,   >
< 1, 5, 10, 7,   >
Array Length: 4
< 1, 5, 10, -10, 7,   >
```

Analyzing the run-time performance of Python’s `insert()` operation.

- Inserting at the beginning of the list is most expensive requiring linear time per operation.

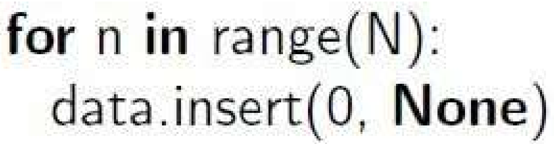
- Inserting at the middle is half the time as inserting at the beginning, still Omega(n) time.

- Inserting at end displays O(1) behaviour.

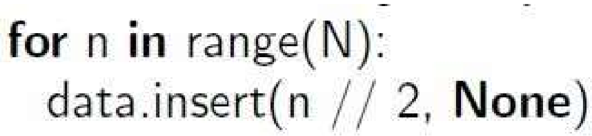


	N				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n // 2$	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

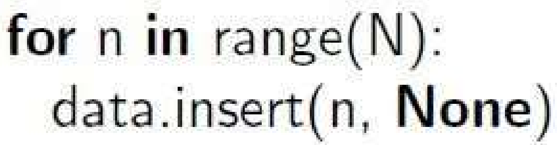
Average running time of `insert(k, val)` measured in microseconds over a sequence of N calls.



```
for n in range(N):
  data.insert(0, None)
```

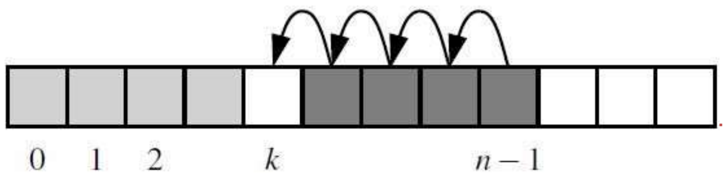


```
for n in range(N):
  data.insert(n // 2, None)
```



```
for n in range(N):
  data.insert(n, None)
```

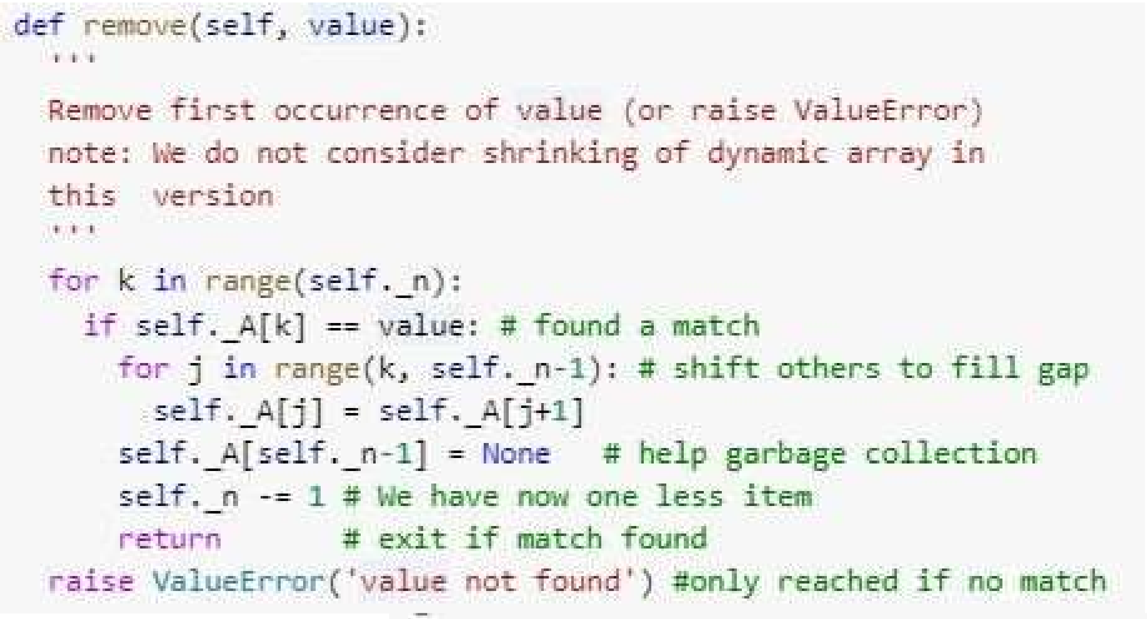
Removing Elements from a List



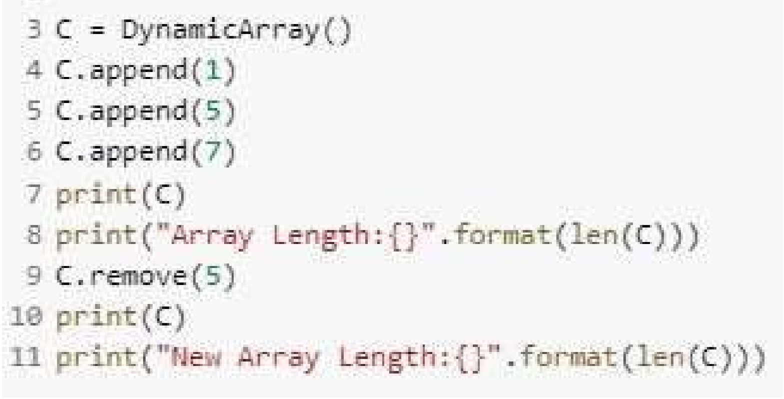
- `pop()` - removes the last element from a list ~ O(1).

- `pop(k)` removes the element that is at index k < n of a list ~ O(n-k) time requirement as the amount of shifting depends on the choice of k. pop(0) is most expensive using $\Omega(n)$ time.

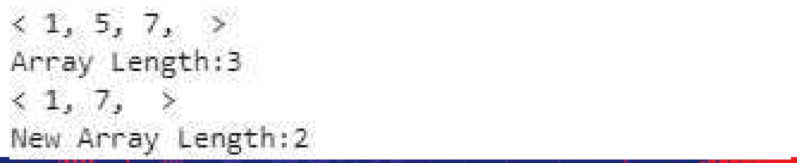
- `remove()` can remove a specified value (without specifying any index) from the list, requires $\Omega(n)$ time.



```python
def remove(self, value):
  '''
  Remove first occurrence of value (or raise ValueError)
  note: We do not consider shrinking of dynamic array in
  this  version
  '''
  for k in range(self._n):
    if self._A[k] == value: # found a match
      for j in range(k, self._n-1): # shift others to fill gap
        self._A[j] = self._A[j+1]
      self._A[self._n-1] = None   # help garbage collection
      self._n -= 1 # We have now one less item
      return       # exit if match found
  raise ValueError('value not found') #only reached if no match
```



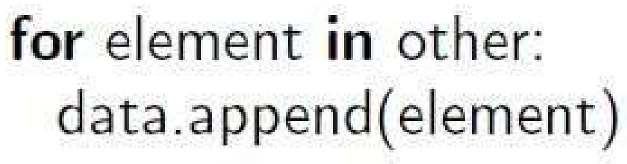
```python
C = DynamicArray()
C.append(1)
C.append(5)
C.append(7)
print(C)
print("Array Length:{}".format(len(C)))
C.remove(5)
print(C)
print("New Array Length:{}".format(len(C)))
```



```
< 1, 5, 7,  >
Array Length:3
< 1, 7,  >
New Array Length:2
```

Extending a List

- Used to add all elements of one list to the end of a second list.
- `data.extend(other)` is equivalent to:



```
for element in other:
  data.append(element)
```

- Running time is proportional to length of the other list ~ O(n) amortized as the first list could be resized to accommodate additional elements.

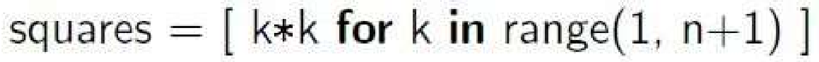
- The extend method is preferable to repeated calls to append. It offers three advantages:
  - ○ Built-in python implementations are comparatively efficient as they are implemented natively in a compiled language (compared to interpreted python code).
  - ○ There is less overhead in calling a single function that accomplishes all the work, versus many individual function calls.
  - ○ The increased efficiency of extend comes from the fact the resulting size of the updated list can be calculated in advance.

Constructing New list

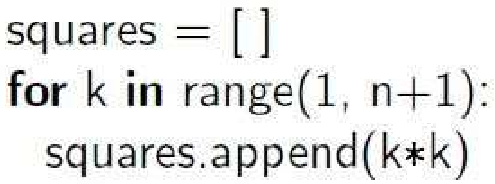
- Asymptotic efficiency is linear in length of the list that is created.

- List comprehension syntax is significantly faster than building the list by repeatedly appending.

- It is more efficient to create a list by using **`[0]*n`** than building such a list incrementally.



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



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

Python’s String Class

Some Notations:

- Length of a string: n
- Length of a second string: m

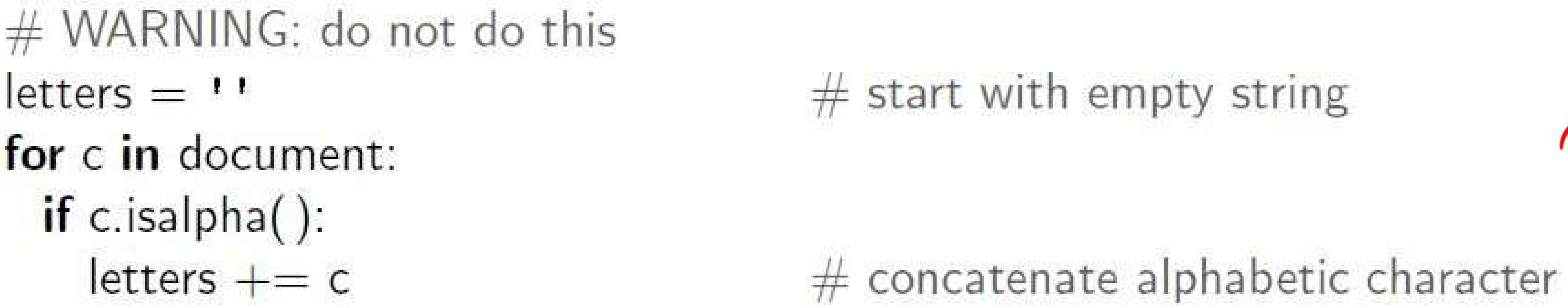
Some common behaviours:

- Methods that produce a new string (e.g., `capitalize()`, `center()`, `strip()`) require time that is linear in length of the string that is produced.

- Behaviours that test Boolean conditions on a string (e.g. `islower()`) take O(n) time, examining all n characters in the worst case.

- Pattern Matching - finding one string within a larger string
    - Examples: `__contains__, find, index, count, replace, split`
    - Naive implementation will lead to O(mn) time complexity. Because, there are n-m+1 starting indices for the pattern and we spend O(m) time at each starting position checking if the pattern matches.

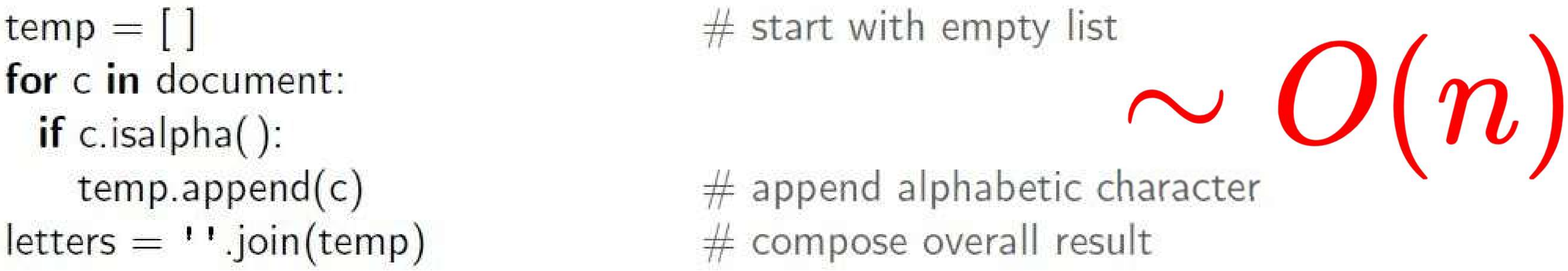
- Composing Strings
    - Creating new string from another string.
    - Example: We have a large string named ‘document’. Our goal is to create a new string ‘letters’ that contains only alphabetic characters of the original string (with spaces, numbers and punctuation removed).
    - Implementation #1



```
# WARNING: do not do this
letters = ''                          # start with empty string
for c in document:
  if c.isalpha():
    letters += c                      # concatenate alphabetic character
```

$\sim O(n^2)$

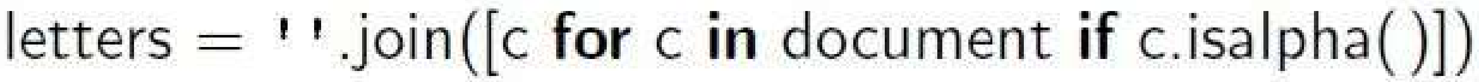
- Implementation #2



```
temp = [ ]                                  # start with empty list
for c in document:
  if c.isalpha( ):
    temp.append(c)                          # append alphabetic character
letters = ''.join(temp)                     # compose overall result
```

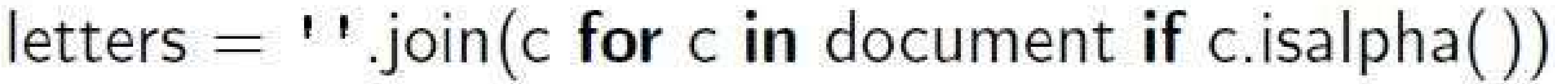
$\sim O(n)$

- Implementation #3 (use list comprehension instead of repeated calls to append):



```
letters = ''.join([c for c in document if c.isalpha( )])
```

- Implementation #4 (avoids the temporary list altogether)



```
letters = ''.join(c for c in document if c.isalpha( ))
```

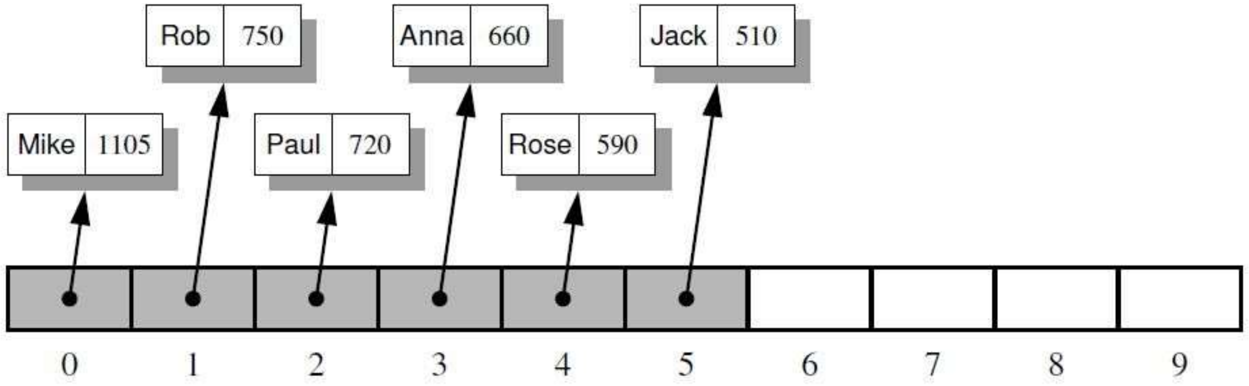
Using Array-Based Sequences

We will study the following three examples:

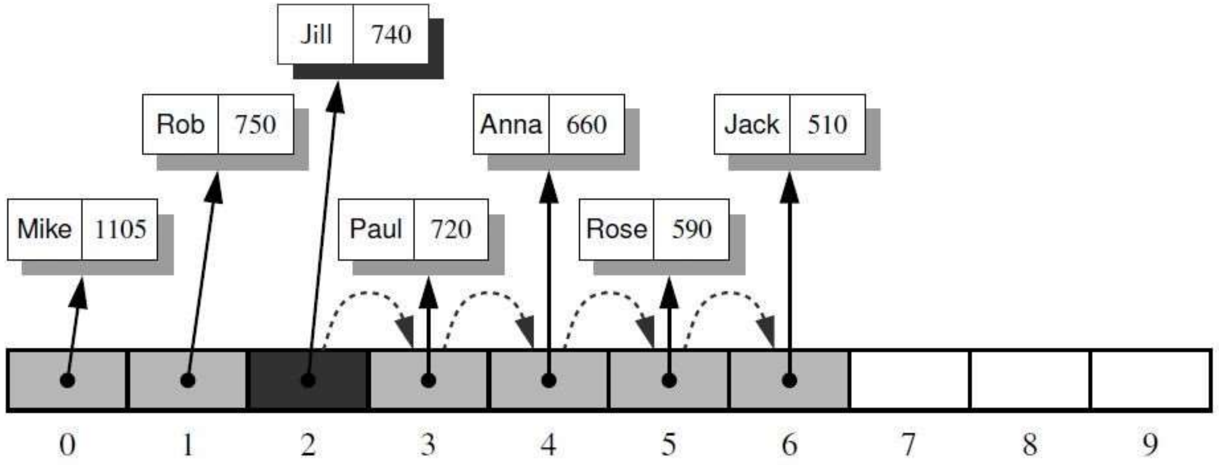
- Storing High scores for a game
- Sorting a Sequence
- Simple Cryptography

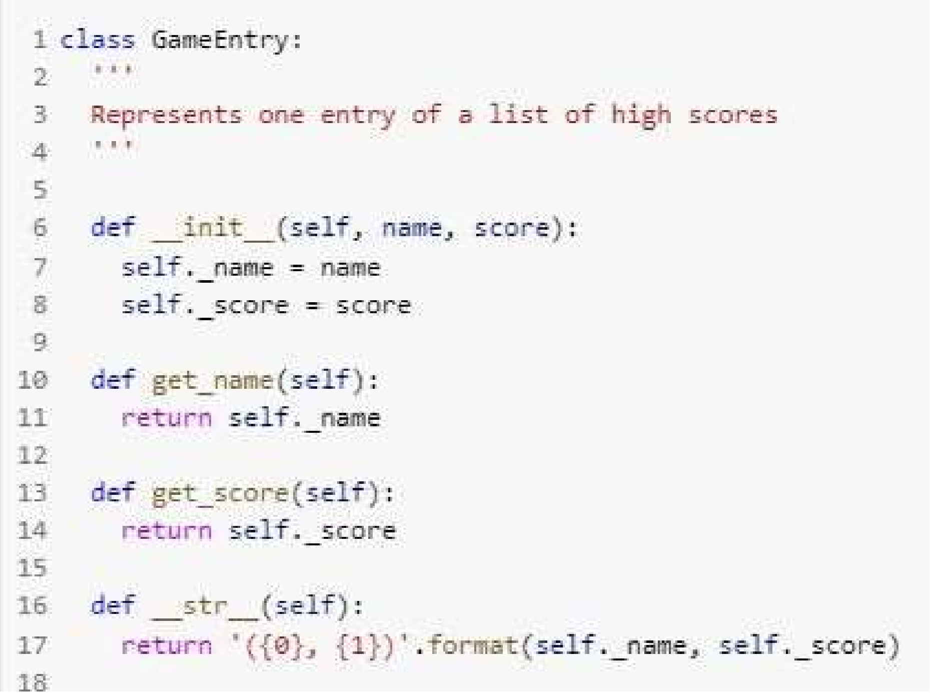
Storing High Scores for a game

- The objective is to store high score entries for a video game.
- Each game entry has a name and a score. Corresponding class is GameEntry.
- To maintain a sequence of high scores, we create another class called Scoreboard.
- Scoreboard can have limited number of entries. A new score only qualifies for the scoreboard if it is strictly higher than the lowest of the high scores in the board.
- When a new entry is made to the scoreboard, elements are shifted to the right to create an appropriate space for new score.

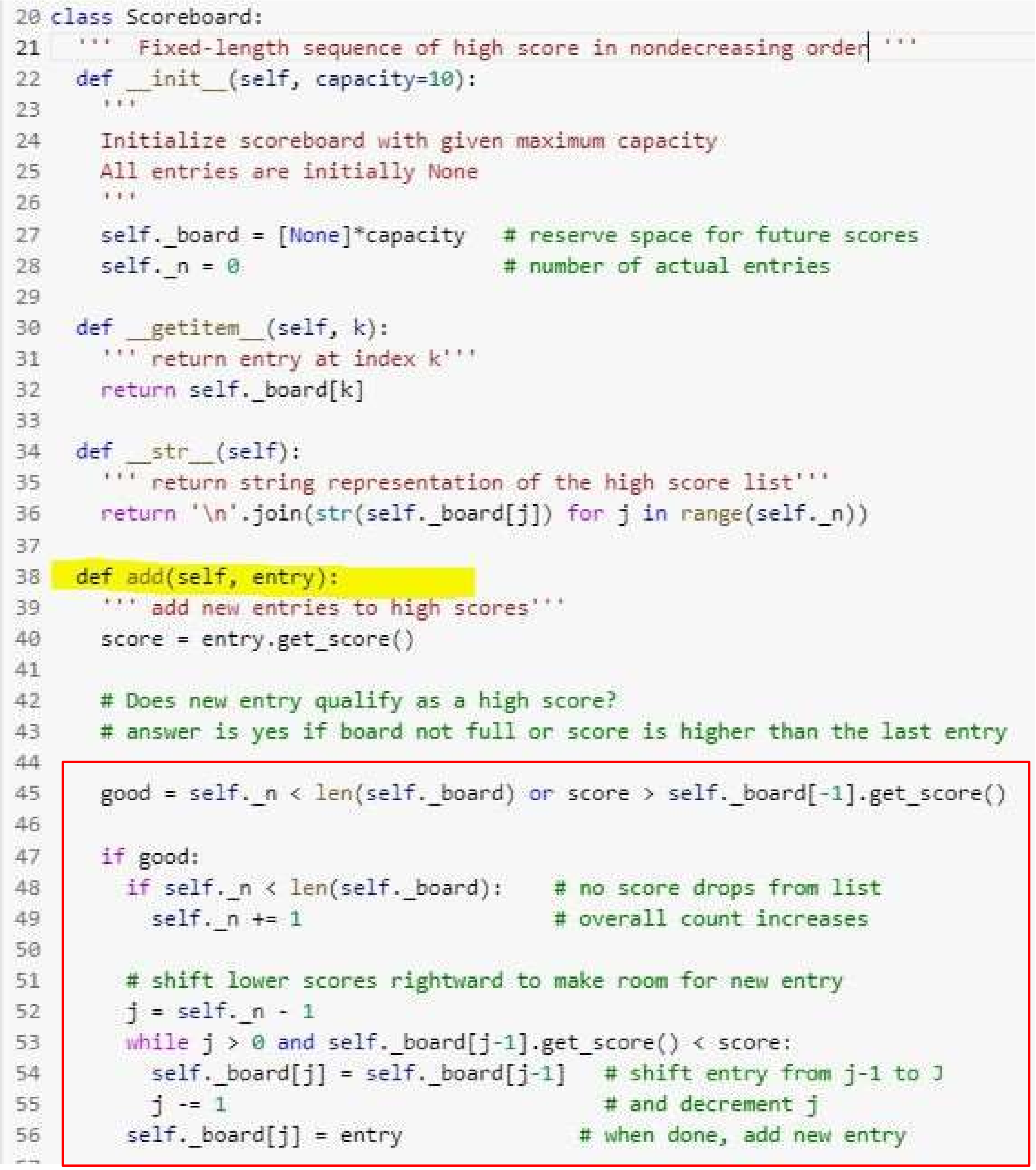


ScoreBoard implemented as a dynamic array

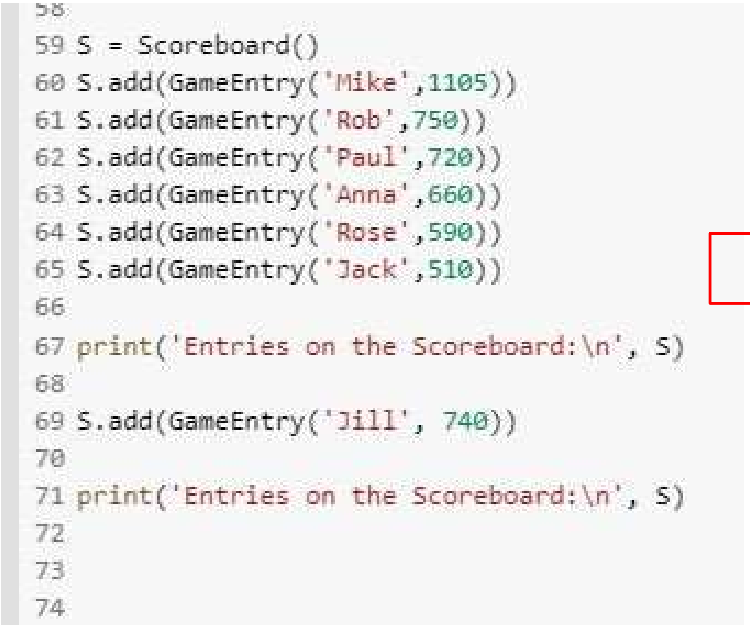




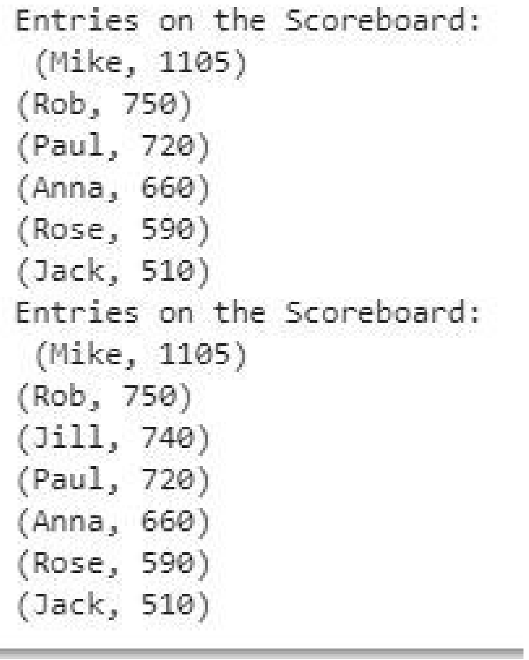
```python
class GameEntry:
  '''
  Represents one entry of a list of high scores
  '''

  def __init__(self, name, score):
    self._name = name
    self._score = score

  def get_name(self):
    return self._name

  def get_score(self):
    return self._score

  def __str__(self):
    return '({0}, {1})'.format(self._name, self._score)

```



```python
class Scoreboard:
  '''  Fixed-length sequence of high score in nondecreasing order '''
  def __init__(self, capacity=10):
    '''
    Initialize scoreboard with given maximum capacity
    All entries are initially None
    '''
    self._board = [None]*capacity   # reserve space for future scores
    self._n = 0                     # number of actual entries

  def __getitem__(self, k):
    ''' return entry at index k'''
    return self._board[k]

  def __str__(self):
    ''' return string representation of the high score list'''
    return '\n'.join(str(self._board[j]) for j in range(self._n))

  def add(self, entry):
    ''' add new entries to high scores'''
    score = entry.get_score()

    # Does new entry qualify as a high score?
    # answer is yes if board not full or score is higher than the last entry

    good = self._n < len(self._board) or score > self._board[-1].get_score()

    if good:
      if self._n < len(self._board):    # no score drops from list
        self._n += 1                    # overall count increases

      # shift lower scores rightward to make room for new entry
      j = self._n - 1
      while j > 0 and self._board[j-1].get_score() < score:
        self._board[j] = self._board[j-1]   # shift entry from j-1 to J
        j -= 1                              # and decrement j
      self._board[j] = entry                # when done, add new entry
```



```python
S = Scoreboard()
S.add(GameEntry('Mike',1105))
S.add(GameEntry('Rob',750))
S.add(GameEntry('Paul',720))
S.add(GameEntry('Anna',660))
S.add(GameEntry('Rose',590))
S.add(GameEntry('Jack',510))

print('Entries on the Scoreboard:\n', S)

S.add(GameEntry('Jill', 740))

print('Entries on the Scoreboard:\n', S)



```

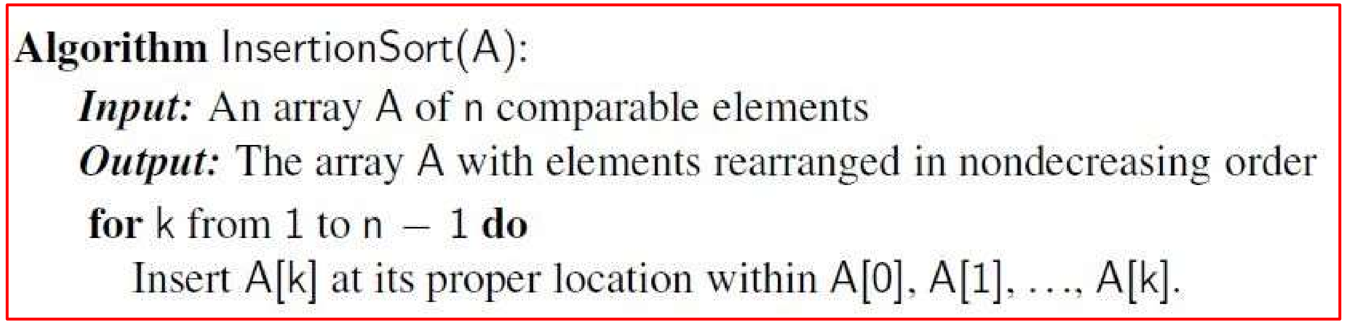


```
Entries on the Scoreboard:
 (Mike, 1105)
(Rob, 750)
(Paul, 720)
(Anna, 660)
(Rose, 590)
(Jack, 510)
Entries on the Scoreboard:
 (Mike, 1105)
(Rob, 750)
(Jill, 740)
(Paul, 720)
(Anna, 660)
(Rose, 590)
(Jack, 510)
```

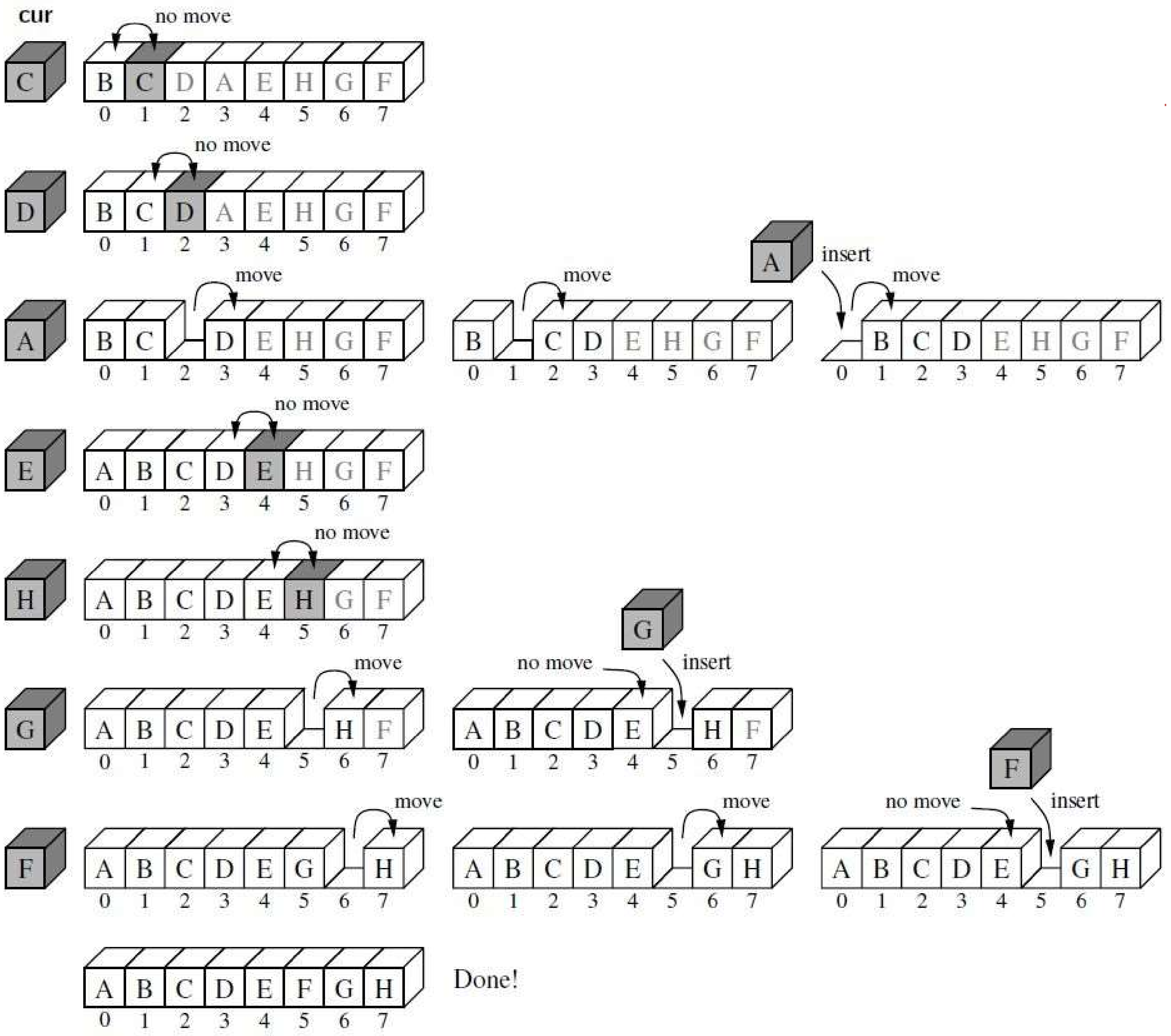
Sorting a Sequence

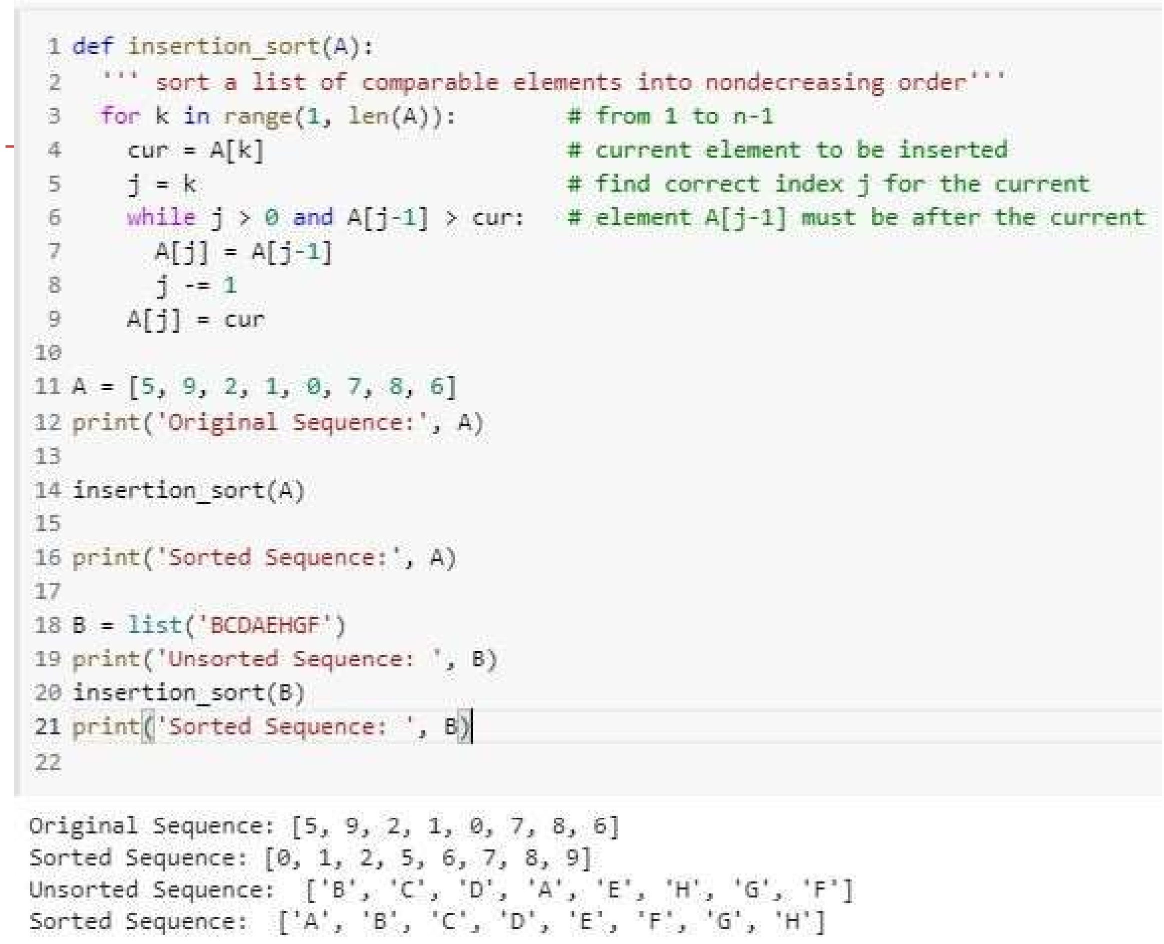
- Objective is to sort an unordered sequence of elements into non-decreasing order of its values.

- Insertion-sort Algorithm
  - Start with the first element in the array. One element by itself is sorted.
  - Consider the second element. If it is smaller than the first, we swap them.
  - Consider the third element. We swap it leftward until it is in the proper order with the first two elements.
  - And so on ...



```
Algorithm InsertionSort(A):
    Input: An array A of n comparable elements
    Output: The array A with elements rearranged in nondecreasing order
    for k from 1 to n − 1 do
        Insert A[k] at its proper location within A[0], A[1], . . ., A[k].
```





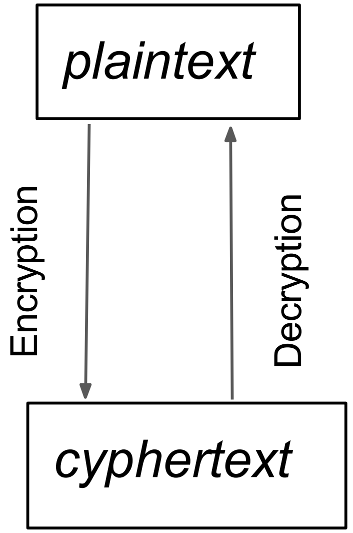
```
def insertion_sort(A):
  ''' sort a list of comparable elements into nondecreasing order'''
  for k in range(1, len(A)):          # from 1 to n-1
    cur = A[k]                        # current element to be inserted
    j = k                             # find correct index j for the current
    while j > 0 and A[j-1] > cur:     # element A[j-1] must be after the current
      A[j] = A[j-1]
      j -= 1
    A[j] = cur

A = [5, 9, 2, 1, 0, 7, 8, 6]
print('Original Sequence:', A)

insertion_sort(A)

print('Sorted Sequence:', A)

B = list('BCDAEHGF')
print('Unsorted Sequence: ', B)
insertion_sort(B)
print('Sorted Sequence: ', B)

```

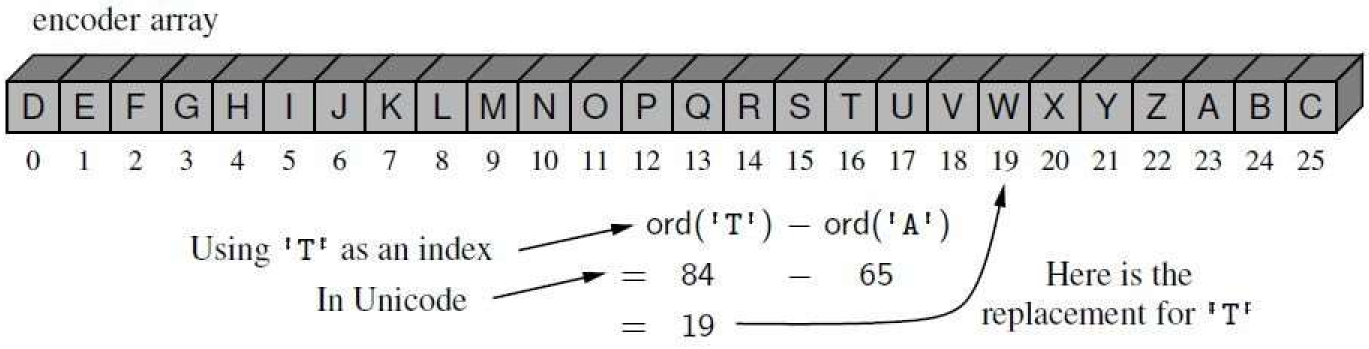
```
Original Sequence: [5, 9, 2, 1, 0, 7, 8, 6]
Sorted Sequence: [0, 1, 2, 5, 6, 7, 8, 9]
Unsorted Sequence:  ['B', 'C', 'D', 'A', 'E', 'H', 'G', 'F']
Sorted Sequence:  ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

Simple Cryptography

- ***Caesar Cipher***: Involves replacing each letter in a message with the letter that is a certain number of letters after it in the alphabet.

- The replacement rule could be represented using another string.




encoder array

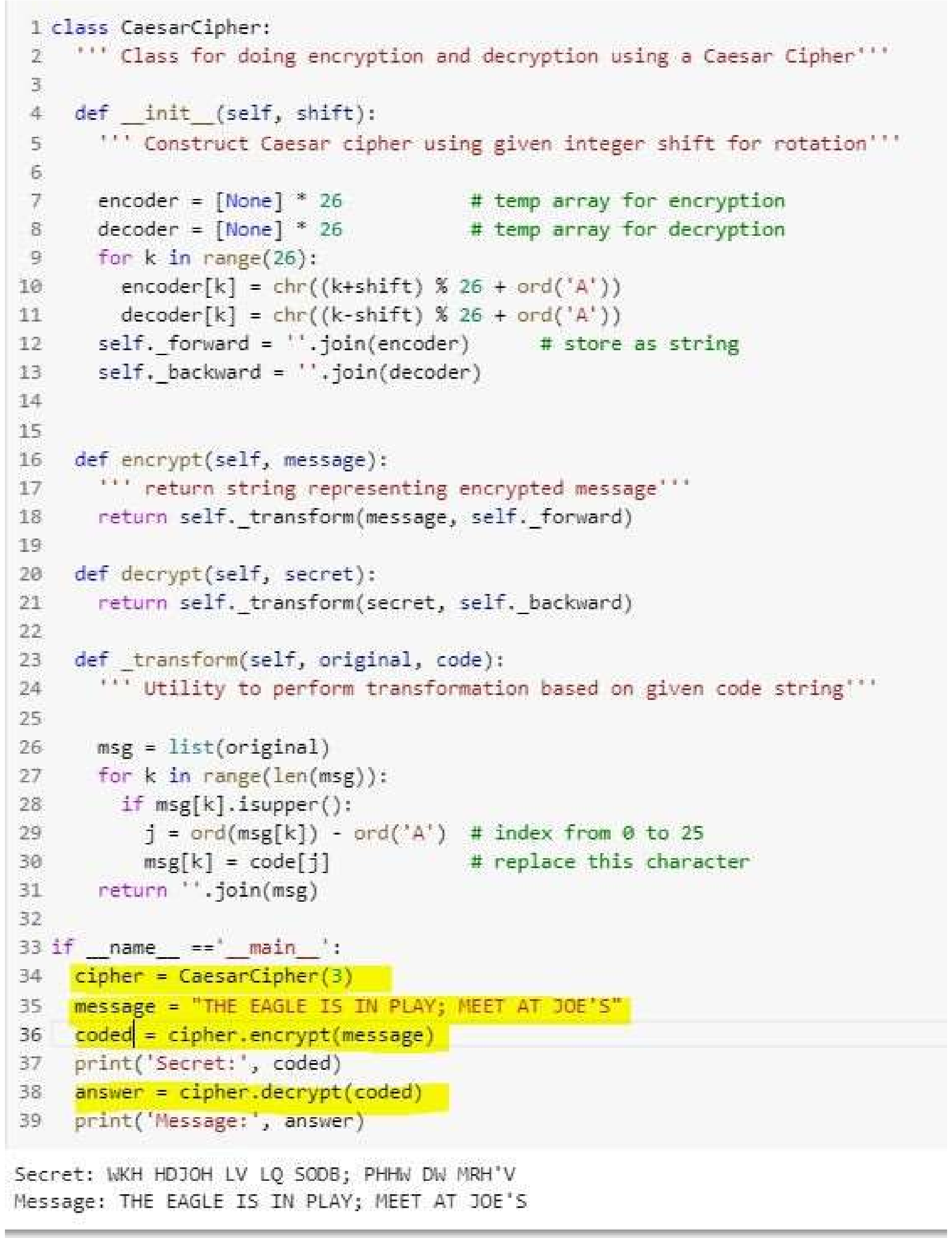
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

- In order to find a replacement for a character in our Caesar cipher, we need to map the characters A to Z to the respective numbers 0 to 25.

- The formula for doing that conversion is j = ord(c) − ord(A).

- Encryption: Replace each letter i with the letter (i+r)% 26

- Decryption: replace each letter i with the letter (i-r)%26



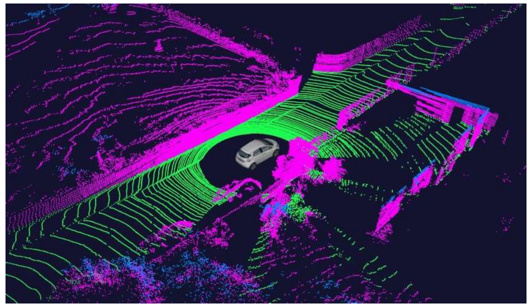
```python
class CaesarCipher:
  ''' Class for doing encryption and decryption using a Caesar Cipher'''

  def __init__(self, shift):
    ''' Construct Caesar cipher using given integer shift for rotation'''

    encoder = [None] * 26          # temp array for encryption
    decoder = [None] * 26          # temp array for decryption
    for k in range(26):
      encoder[k] = chr((k+shift) % 26 + ord('A'))
      decoder[k] = chr((k-shift) % 26 + ord('A'))
    self._forward = ''.join(encoder)      # store as string
    self._backward = ''.join(decoder)


  def encrypt(self, message):
    ''' return string representing encrypted message'''
    return self._transform(message, self._forward)

  def decrypt(self, secret):
    return self._transform(secret, self._backward)

  def _transform(self, original, code):
    ''' Utility to perform transformation based on given code string'''

    msg = list(original)
    for k in range(len(msg)):
      if msg[k].isupper():
        j = ord(msg[k]) - ord('A')  # index from 0 to 25
        msg[k] = code[j]            # replace this character
    return ''.join(msg)

if __name__ =='__main__':
  cipher = CaesarCipher(3)
  message = "THE EAGLE IS IN PLAY; MEET AT JOE'S"
  coded = cipher.encrypt(message)
  print('Secret:', coded)
  answer = cipher.decrypt(coded)
  print('Message:', answer)
```

```
Secret: WKH HDJOH LV LQ SODB; PHHW DW MRH'V
Message: THE EAGLE IS IN PLAY; MEET AT JOE'S
```

Multi-dimensional Data Sets

- Have more than one dimensions. A two-dimensional list may be called a matrix.

- Examples:
  - Geographical information may be naturally represented in two-dimensions.
  - Range data obtained from a lidar is a 3D data.
  - Monochrome images are 2D images



	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

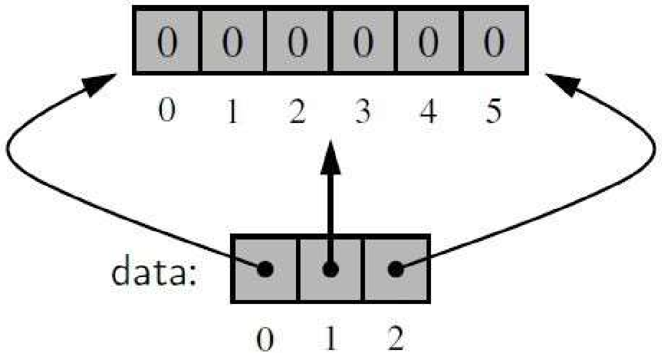
Two-dimensional integer dataset (Number of stores in various regions in district)

22	18	709	5	33
45	32	830	120	750
4	880	45	66	61

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

Each element can be accessed by using a syntax: data[1][3]

Constructing multi-dimensional list



- Quickly initialize a one-dimensional list:

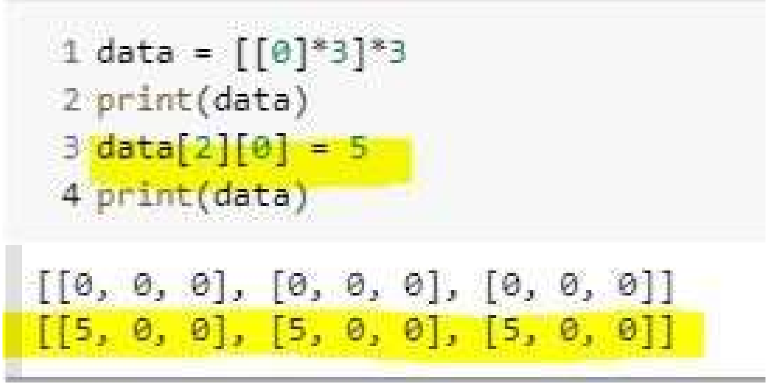
  Data = [0] * n

- Initialize multi-dimensional list:

Creates 1-D list

```
data = ([0] * c) * r          # Warning: this is a mistake

data = [ [0] * c ] * r        # Warning: still a mistake
```

The problem is that all r entries of the list known as data are references to the same instance of a list of c zeros



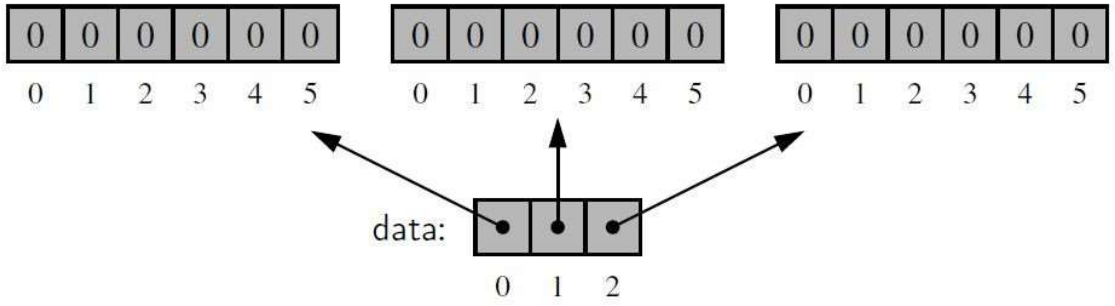
```
data = [[0]*3]*3
print(data)
data[2][0] = 5
print(data)
```

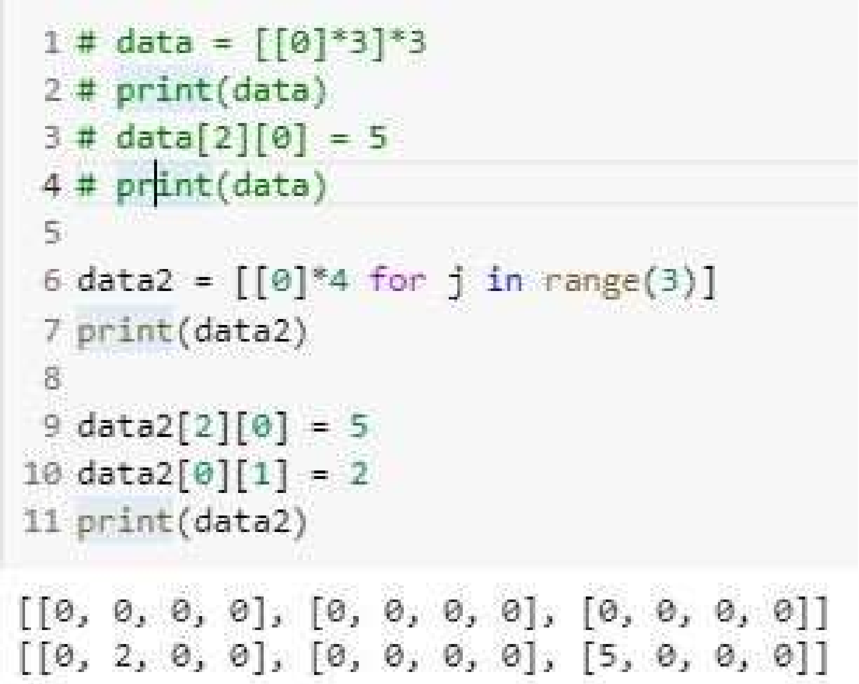
```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[5, 0, 0], [5, 0, 0], [5, 0, 0]]
```

# Correct implementation for initializing a 2-dimensional list

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

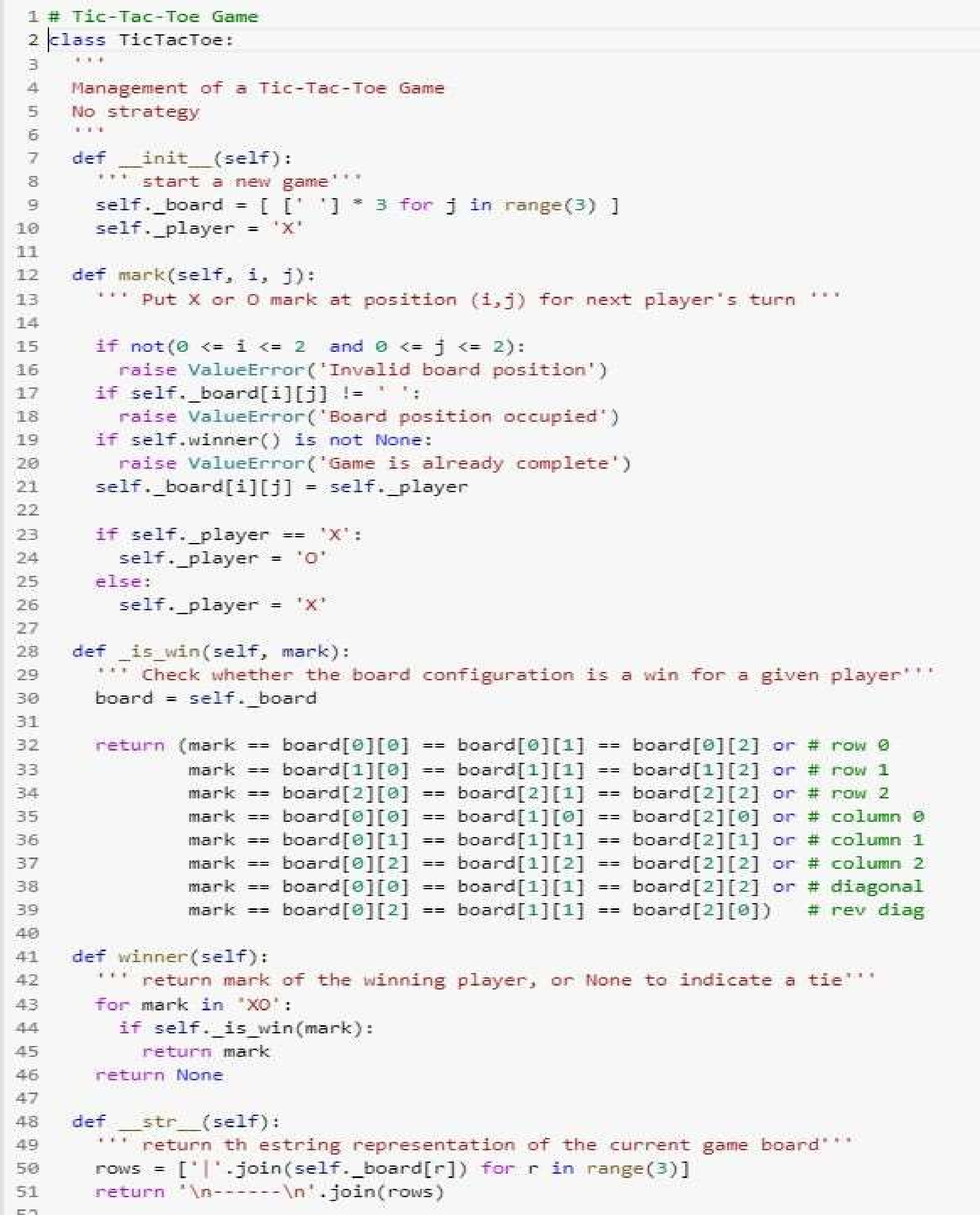
This ensures that each cell of the primary list refers to an independent instance of a secondary list



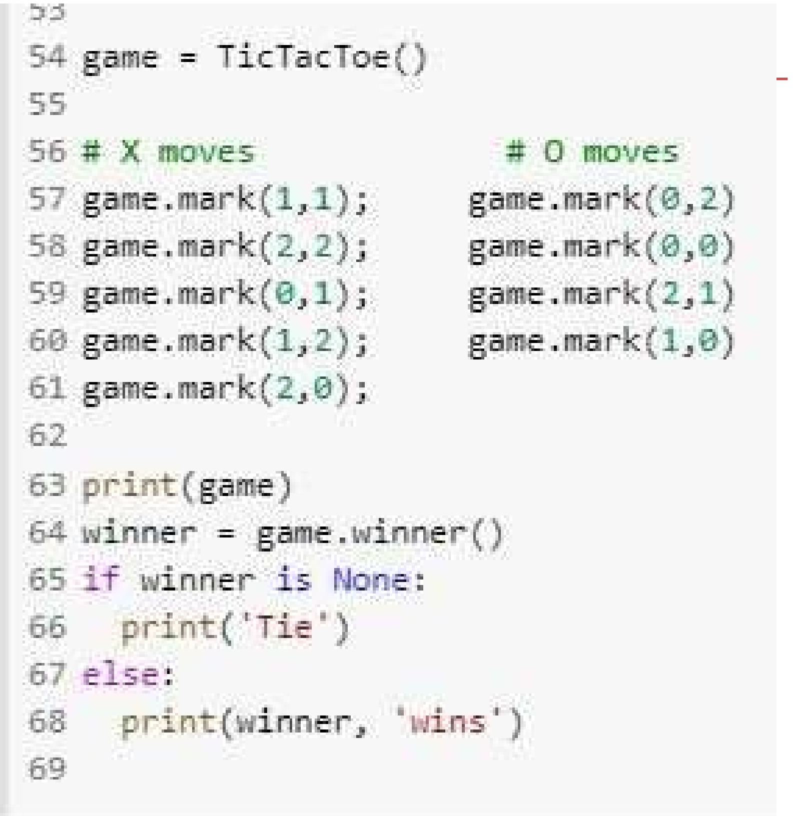


```
# data = [[0]*3]*3
# print(data)
# data[2][0] = 5
# print(data)

data2 = [[0]*4 for j in range(3)]
print(data2)

data2[2][0] = 5
data2[0][1] = 2
print(data2)
```

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
[[0, 2, 0, 0], [0, 0, 0, 0], [5, 0, 0, 0]]
```



```
# Tic-Tac-Toe Game
class TicTacToe:
  '''
  Management of a Tic-Tac-Toe Game
  No strategy
  '''
  def __init__(self):
    ''' start a new game'''
    self._board = [ [' '] * 3 for j in range(3) ]
    self._player = 'X'

  def mark(self, i, j):
    ''' Put X or O mark at position (i,j) for next player's turn '''

    if not(0 <= i <= 2  and 0 <= j <= 2):
      raise ValueError('Invalid board position')
    if self._board[i][j] != ' ':
      raise ValueError('Board position occupied')
    if self.winner() is not None:
      raise ValueError('Game is already complete')
    self._board[i][j] = self._player

    if self._player == 'X':
      self._player = 'O'
    else:
      self._player = 'X'

  def _is_win(self, mark):
    ''' Check whether the board configuration is a win for a given player'''
    board = self._board

    return (mark == board[0][0] == board[0][1] == board[0][2] or # row 0
            mark == board[1][0] == board[1][1] == board[1][2] or # row 1
            mark == board[2][0] == board[2][1] == board[2][2] or # row 2
            mark == board[0][0] == board[1][0] == board[2][0] or # column 0
            mark == board[0][1] == board[1][1] == board[2][1] or # column 1
            mark == board[0][2] == board[1][2] == board[2][2] or # column 2
            mark == board[0][0] == board[1][1] == board[2][2] or # diagonal
            mark == board[0][2] == board[1][1] == board[2][0])   # rev diag

  def winner(self):
    ''' return mark of the winning player, or None to indicate a tie'''
    for mark in 'XO':
      if self._is_win(mark):
        return mark
    return None

  def __str__(self):
    ''' return th estring representation of the current game board'''
    rows = ['|'.join(self._board[r]) for r in range(3)]
    return '\n------\n'.join(rows)
```



```
game = TicTacToe()

# X moves                 # O moves
game.mark(1,1);        game.mark(0,2)
game.mark(2,2);        game.mark(0,0)
game.mark(0,1);        game.mark(2,1)
game.mark(1,2);        game.mark(1,0)
game.mark(2,0);

print(game)
winner = game.winner()
if winner is None:
  print('Tie')
else:
  print(winner, 'wins')

```

```
O|X|O
------
O|X|X
------
X|O|X
Tie
```

Summary

- We studied the following in this lecture:
  - Working of the low-level arrays - initialization, accessing elements etc.
  - Referential arrays, Compact arrays.
  - Creating and managing dynamic array
  - Understanding python array-based sequences: list, tuple, string
  - Analyzing the run-time complexity of managing these array-based sequences
  - Few applications of array-based sequences
  - Creating multi-dimensional arrays