Algorithmics 2

Yahya Benkaouz

Outline

- Array based Sequences
- Sorting and searching algorithms on Arrays
- Stacks, Queues and Deques

Array based Sequences

Python’s Sequence Types

We will explore Python’s various “sequence” classes, namely the built in classes:

list: List items are ordered, changeable, and allow duplicate values.

```
thislist = ["apple", "banana", "cherry"]
```

tuple : A tuple is a collection which is ordered and unchangeable.

```
thistuple = ("apple", "banana", "cherry")
```

str: Strings are arrays of bytes representing unicode characters.

```
a = "Hello"
```

Python’s Sequence Types

Common characteristics:

- Support **indexing** to access an individual element of a sequence

```
thislist[0] -> "apple”
thistuple[1] -> "banana”
a[2] -> “l”
```

- Use a low-level concept known as an **array** to represent the sequence

Python’s Sequence Types

Differences :

- In the abstractions that these classes represent.
- Internal representation of instance of these classes.

Usage:

- Widely used in Python.
- Building block upon which complex data structures are developed.

=> a Need of a clear understanding of the public behavior and the inner working of these classes

Python’s Sequence Types

What we will cover :

- Public Behaviors
  - The basic usage of lists, strings, and tuples may seem straightforward.
  - There are several important subtleties regarding the behaviors associated with these classes *(the meaning of making a copy? The meaning of taking a slice?....).*

- Implementation details
  - A misunderstanding of a behavior can easily lead to inadvertent bugs in a program.
  - The efficiency of a program depends greatly on the efficiency of the components upon which it relies

- Asymptotic and experimental Analysis

Low-level Arrays

- To accurately describe the way in which Python represents the sequence types
  - we must first discuss aspects of the low-level computer architecture.

- 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.

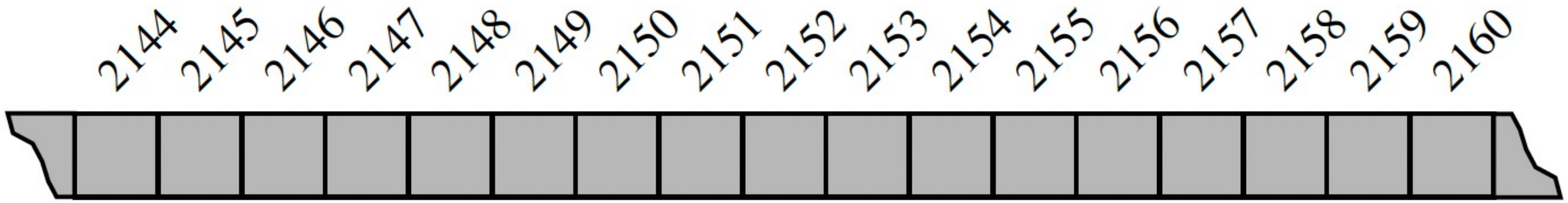
- A typical unit is a byte = 8 bits.

Low-level Arrays

- A computer system will have a huge number of bytes of memory.

- To keep track of what information is stored in what byte, the computer uses an abstraction known as a memory address.
    - The computer system can refer to the data in “byte #2150” versus the data in “byte #2157”

- We often portray the address in sequential fashion



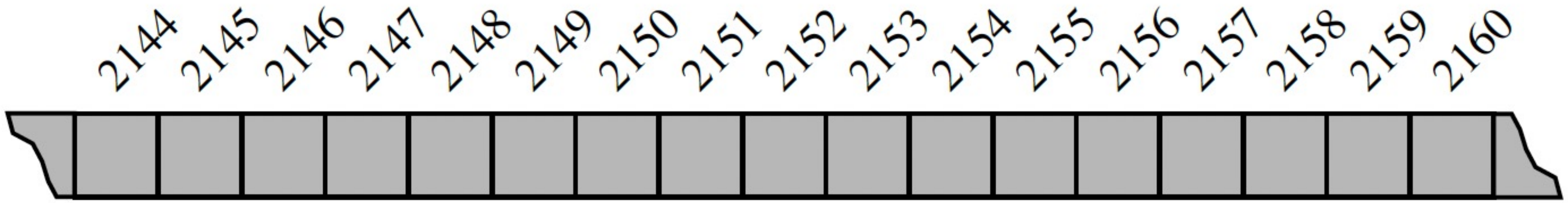
A representation of a portion of a computer’s memory, with individual bytes labeled with consecutive memory addresses

Low-level Arrays

- Computer hardware is designed, in theory, so that any byte of the main memory can be efficiently accessed based upon its memory address
    - It is just as easy to retrieve byte #8675309 as it is to retrieve byte #309

- We say that a computer's main memory performs as random access memory (RAM).

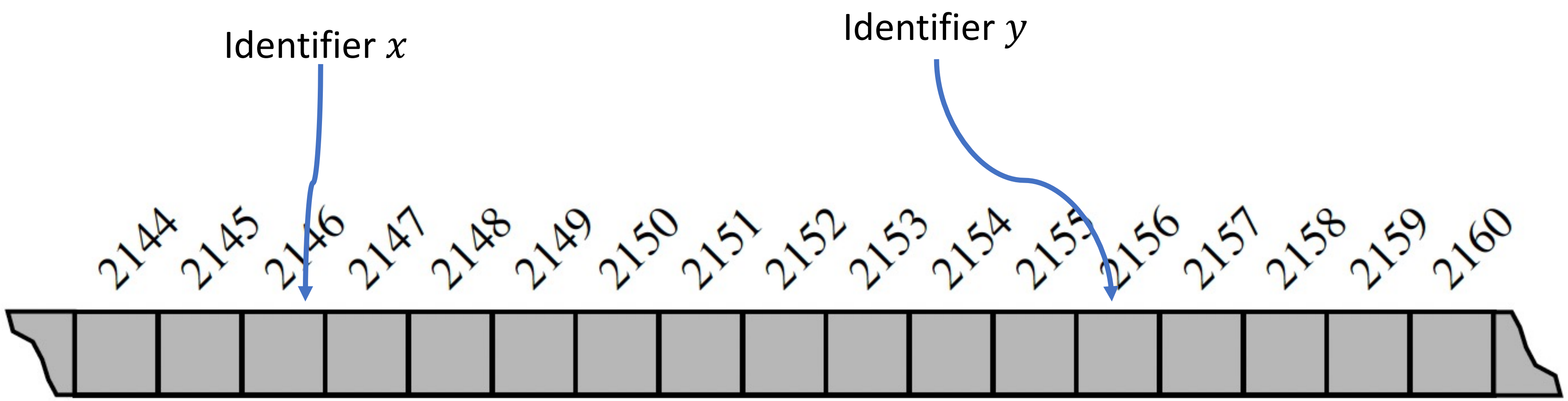
- Using the notation for asymptotic analysis, we say that any individual byte of memory can be stored or retrieved in **O(1) time**.



A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses

Low-level Arrays

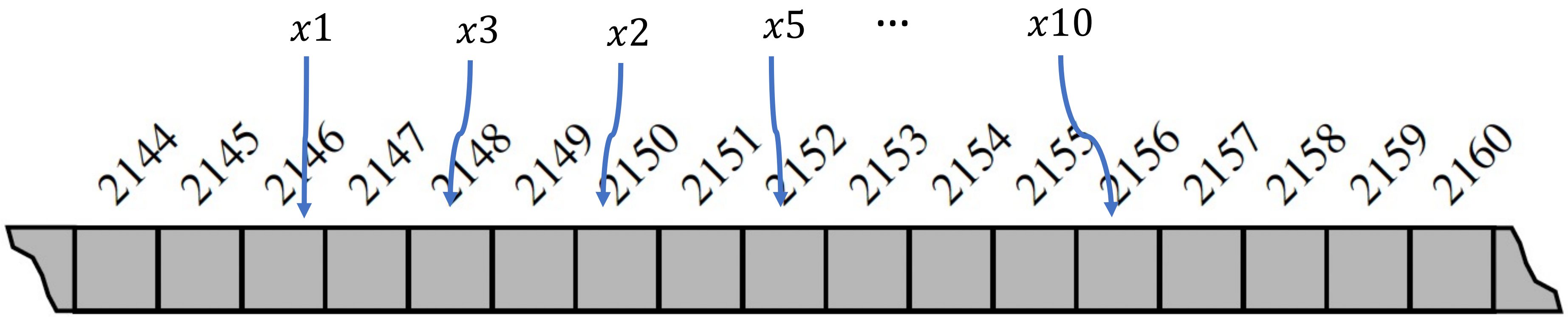
- In programming languages, we keep track of the association between an identifier and the memory address in which the associated value is stored.
  - For example, identifier x might be associated with one value stored in memory, while y is associated with another value stored in memory.



A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses

Low-level Arrays

- A common programming task is to keep track of a sequence of related objects.
    - For example:
        - we may want a video game to keep track of the top ten scores for that game.
    - rather than use ten different variables for this task,
    - we would prefer to use a single name for the group and use index numbers to refer to the high scores in that group.

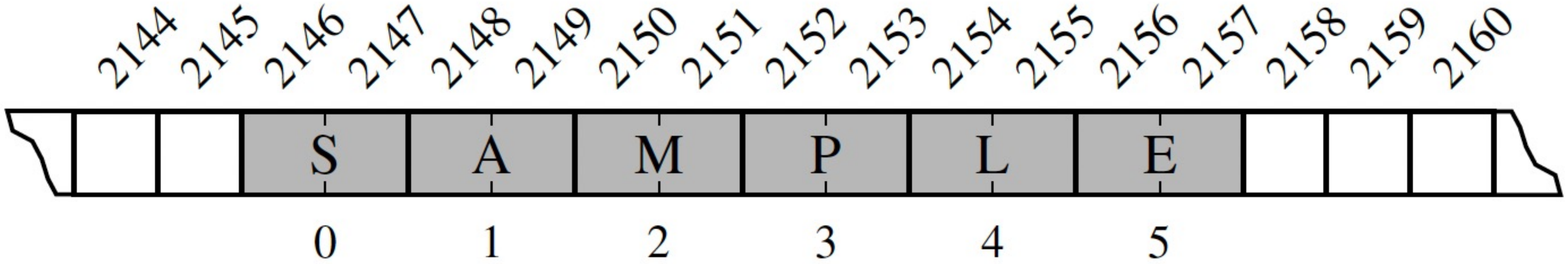


A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses

Low-level Arrays

- **Array** : A group of related variables can be stored one after another in a contiguous portion of the computer's memory.

- For example: a text string is stored as an ordered sequence of individual characters.



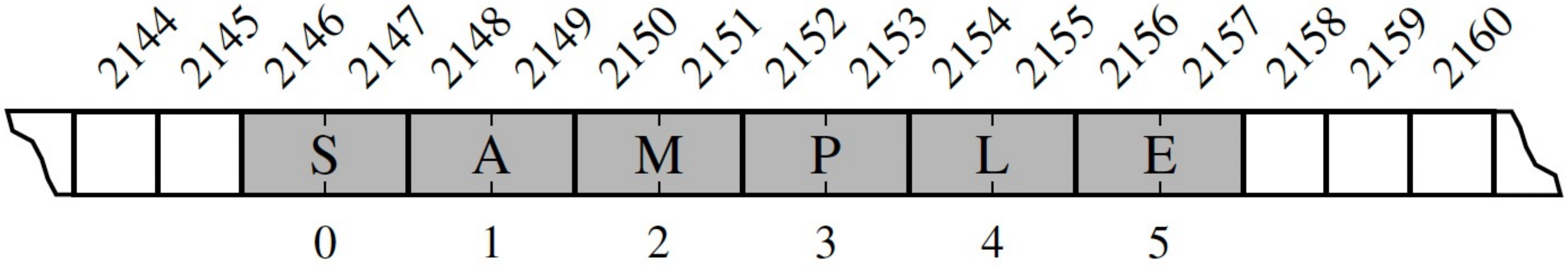
A Python string embedded as an array of characters in the computer's memory.

*Python internally represents each Unicode character with **16 bits** (i.e., **2 bytes).***

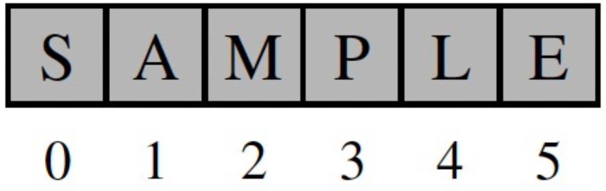
- **Cell** : refers to each location within an array
- An integer index describes its location within the array, (cells numbers starts with 0)

Low-level Arrays

- Each cell of an array must use the same number of bytes.
  - => Allows an arbitrary cell of the array to be accessed in constant time based on its index
  - Memory @ of element at index i = starting @ + cellsize * index



- A programmer envision a more typical **high-level abstraction of an array.**



Referential Arrays

- Motivating example:
  - Assume that we want a medical information system to keep track of the patients currently assigned to beds in a certain hospital.

  - Assume that the hospital has 200 beds (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 instance, we can use a Python's list:

[“Rene” , “Joseph”, “Janet”, “Jonas”, “Helen”, “Virginia”, ...]

Referential Arrays

[“Rene” , “Joseph”, “Janet”, “Jonas”, “Helen”, “Virginia”, ...]

- Python must respect the requirement:

  Each cell of an array must use the same number of bytes

- Strings naturally have different lengths:
  - Python could attempt to reserve enough space for each cell to hold the maximum length string.
    - => Wasteful

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

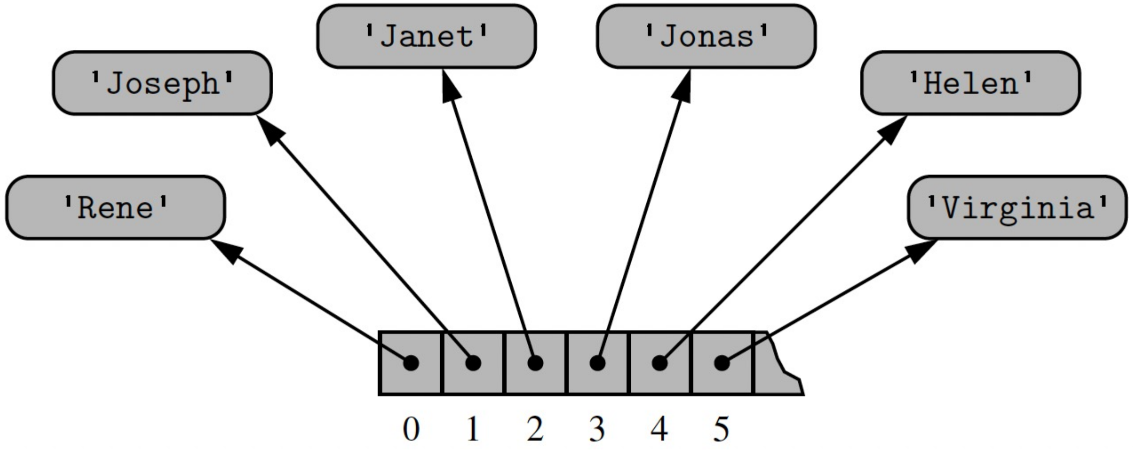
Referential Arrays

[“Rene” , “Joseph”, “Janet”, “Jonas”, “Helen”, “Virginia”, ...]

- **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.

The number of bits used to store the memory address of each element is fixed (e.g., 64-bits per address)

=> Python can support constant-time access to a list or tuple element based on its index.



An array storing references to strings
*The element of the list might be instance of a **Patient Class***
*A None **Object** can be an element of the list (empty bed)*

Referential Arrays

- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
    - A single object might be an element of two or more lists.
    - A single list instance may include multiple references to the same object as elements of the list.

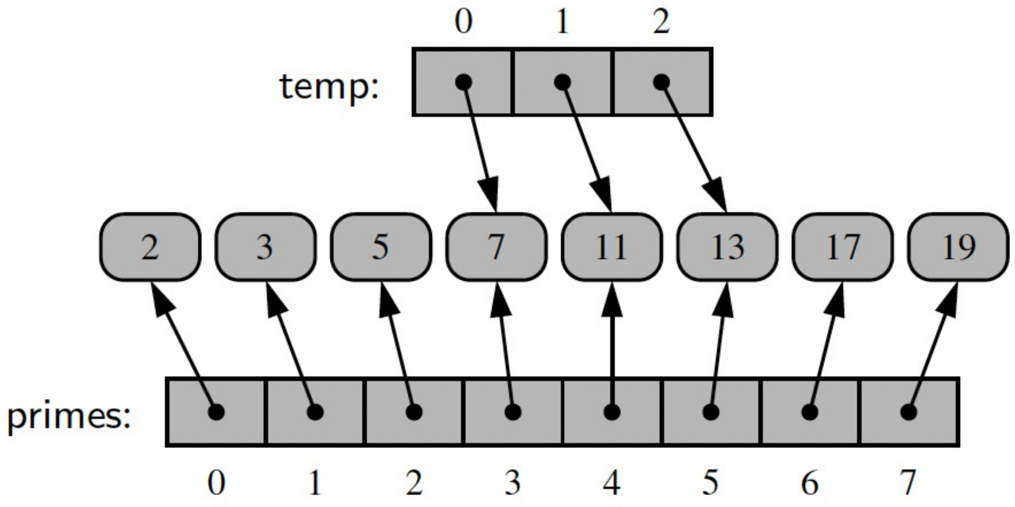
Referential Arrays

- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
    - A single object might be an element of two or more lists.

Example:

- When you compute a slice of a list, the result is a new list instance,
- but that new list has references to the same elements that are in the original list.

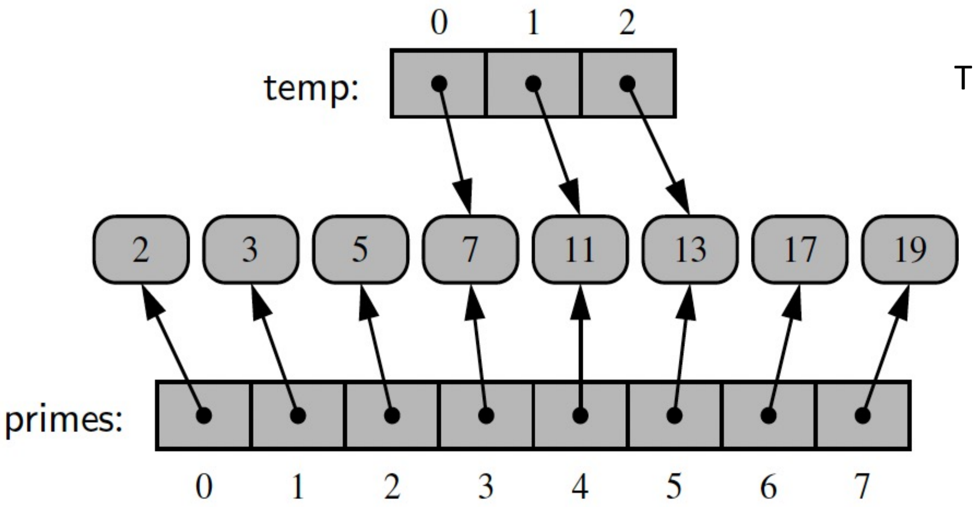
The result of the command ***temp = primes[3:6]***



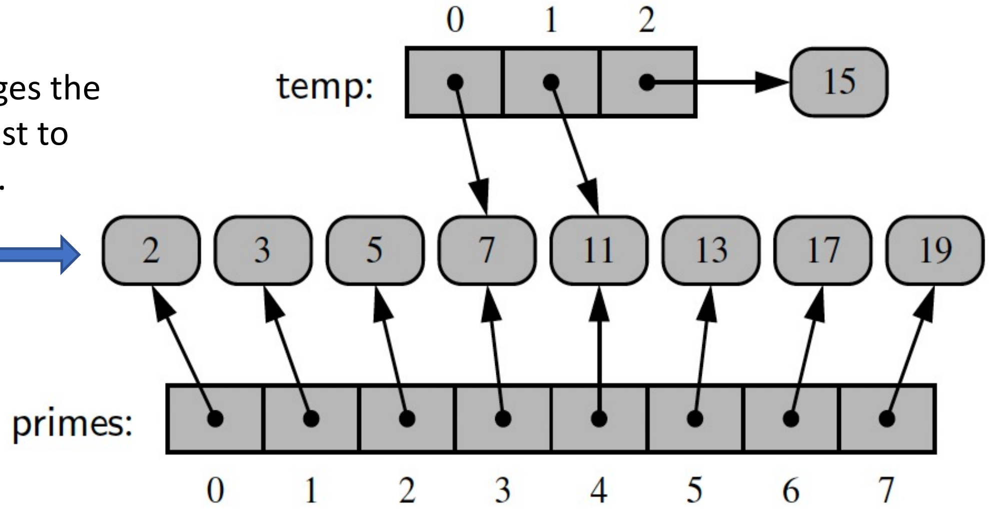
Referential Arrays

- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
  - A single object might be an element of two or more lists.

The fact that the two lists share elements is not that significant, as neither of the lists can cause a change to the shared object.
(as integer is immutable)



The command ***temp[2] = 15*** changes the reference in cell 2 of the temp list to reference a different object.



Referential Arrays

- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
    - A single object might be an element of two or more lists.
    - A single list instance may include multiple references to the same object as elements of the list.

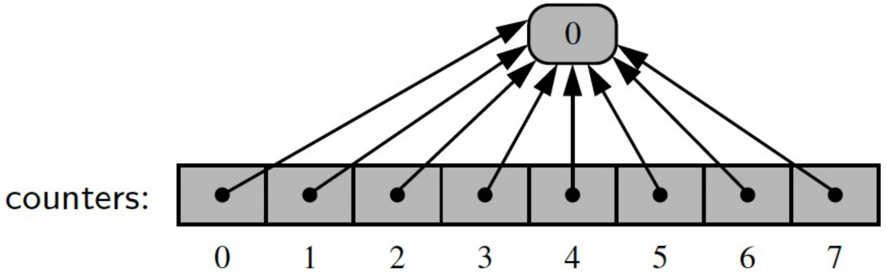
Referential Arrays

- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
  - A single list instance may include multiple references to the same object as elements of the list.

Example:

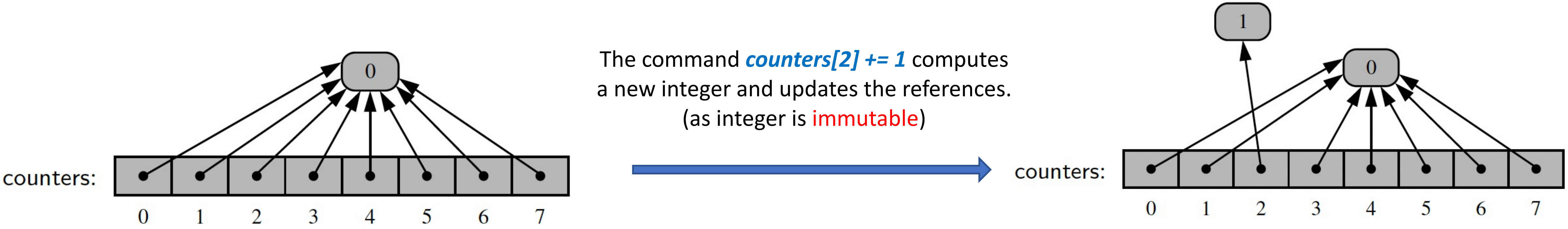
- Initializing an array of integers: ***counters = [0] * 8***

- This syntax produces a list of length eight, with all eight elements being the value zero.
- Technically, all eight cells of the list reference the same object.



Referential Arrays

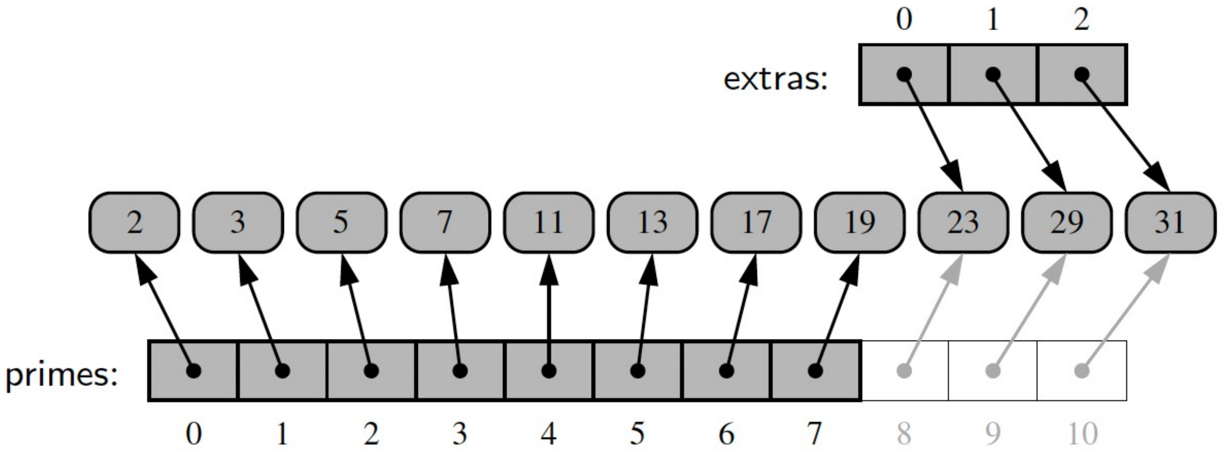
- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
  - A single list instance may include multiple references to the same object as elements of the list.



The command ***counters[2] += 1*** computes a new integer and updates the references. (as integer is immutable)

Referential Arrays

- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
  - A single list instance may include multiple references to the same object as elements of the list.



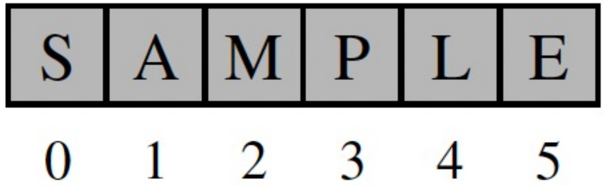
The effect of command ***primes.extend(extras)***

The *extend* command: add all elements from one list to the end of another list.

Compact Arrays

- Strings are represented in Python using an array of characters (not an array of references).

- The array is referred to as a **compact array** :
    - The array is storing the bits that represent the primary data (characters in the case of strings)

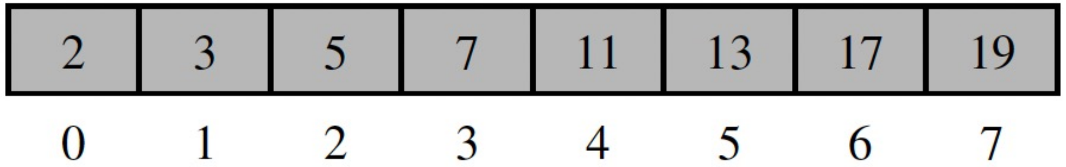


S	A	M	P	L	E
0	1	2	3	4	5

- Advantage of “Compact Arrays”:
    - The overall memory usage will be much lower for a compact structure. (because there is no overhead devoted to the explicit storage of the sequence of memory references (in addition to the primary data)).
    - The primary data are stored consecutively in memory. (It is not the case for a referential structure)

Compact Arrays

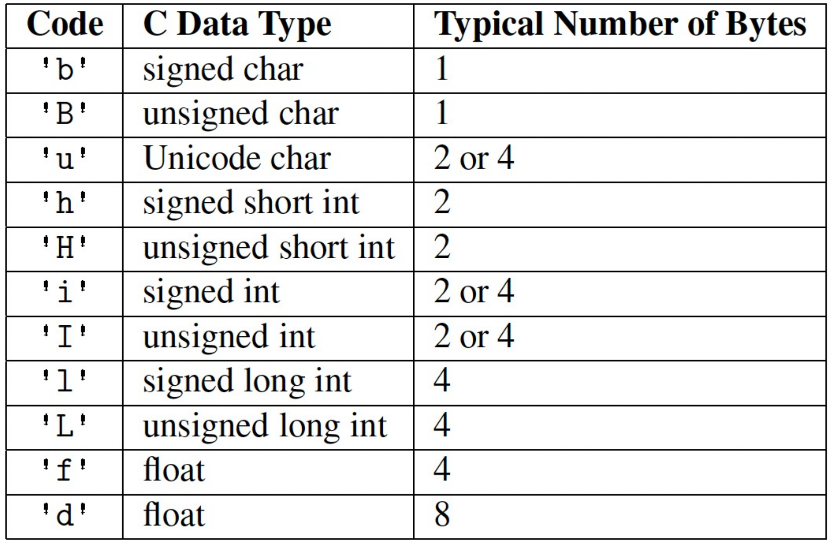
- Python array:
- The array class of the module array provides compact storage for primitive datatypes.



Integers stored compactly as elements of a Python array

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

Typecode: a character that designates the type of data that will be stored in the array

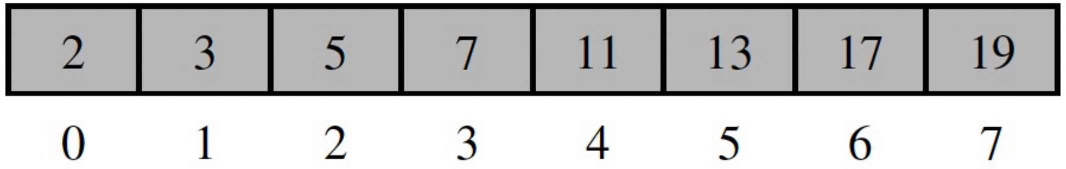


Code	C Data Type	Typical Number of Bytes
'b'	signed char	1
'B'	unsigned char	1
'u'	Unicode char	2 or 4
'h'	signed short int	2
'H'	unsigned short int	2
'i'	signed int	2 or 4
'I'	unsigned int	2 or 4
'l'	signed long int	4
'L'	unsigned long int	4
'f'	float	4
'd'	float	8

To do: Try the **getsizeof** method of the **sys** module and compare the size of a list of integers and a compact arrays of integers.

Compact Arrays

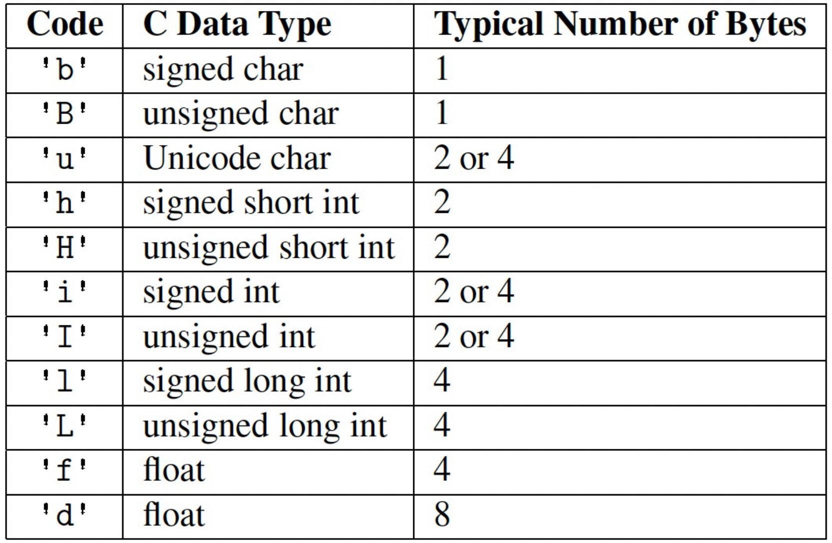
- Python array:
- The array class of the module array provides compact storage for primitive datatypes.



Integers stored compactly as elements of a Python array

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

Typecode: a character that designates the type of data that will be stored in the array



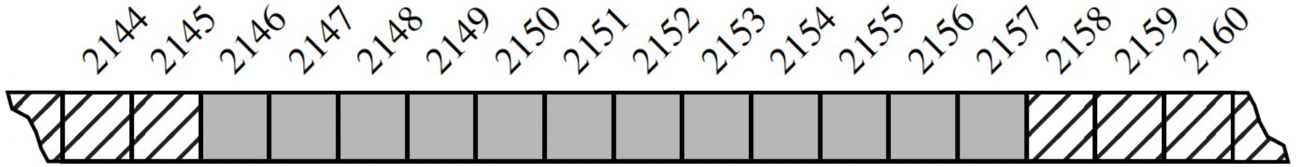
Code	C Data Type	Typical Number of Bytes
'b'	signed char	1
'B'	unsigned char	1
'u'	Unicode char	2 or 4
'h'	signed short int	2
'H'	unsigned short int	2
'i'	signed int	2 or 4
'I'	unsigned int	2 or 4
'l'	signed long int	4
'L'	unsigned long int	4
'f'	float	4
'd'	float	8

The **array** module does not provide support for making compact arrays of user defined data types.

Compact arrays of user defined data types can be created with the lower level support of a module named **ctypes**.

Dynamic Arrays

- **When creating a low-level array** in a computer system,
  - The precise size of that array must be explicitly declared in order for the system to properly allocate a consecutive piece of memory for its storage.



An array of 12 bytes allocated in memory locations 2146 through 2157

- The capacity of an array cannot trivially be increased by expanding into subsequent cells.
  - For **tuple** and **str**: Immutable Classes => No need to increase the size of the underlying array.
  - For **list**: Allows to add elements, with no apparent limit on the overall capacity of the list.

Python relies on an algorithmic technique known as a **dynamic array**

Dynamic Arrays

- **Dynamic Array:**

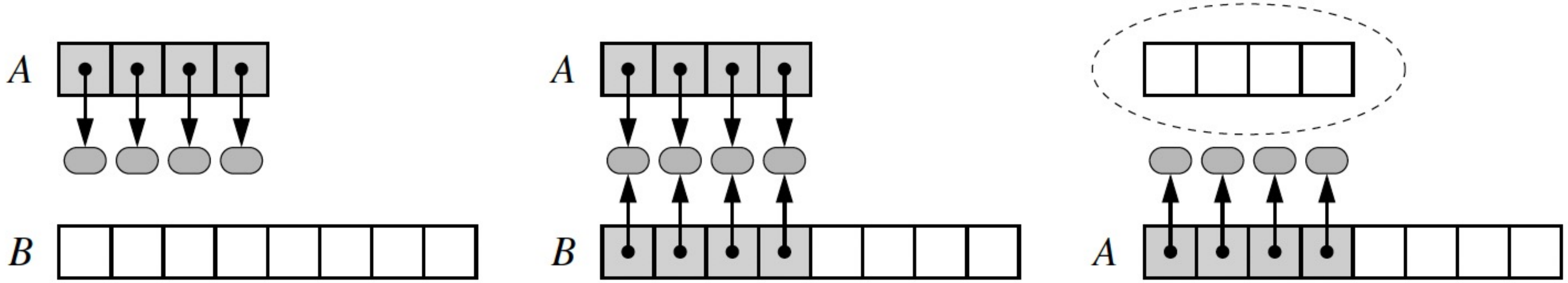
- The key is to provide means to grow the array A that stores the elements of a list.
  - We cannot actually grow that array, as its capacity is fixed.
  - If an element is appended to a list at a time when the underlying array is full, we perform the following steps:

    1. Allocate a new array B with larger capacity.

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

    3. Set $A = B$.

    4. Insert the new element in the new array.

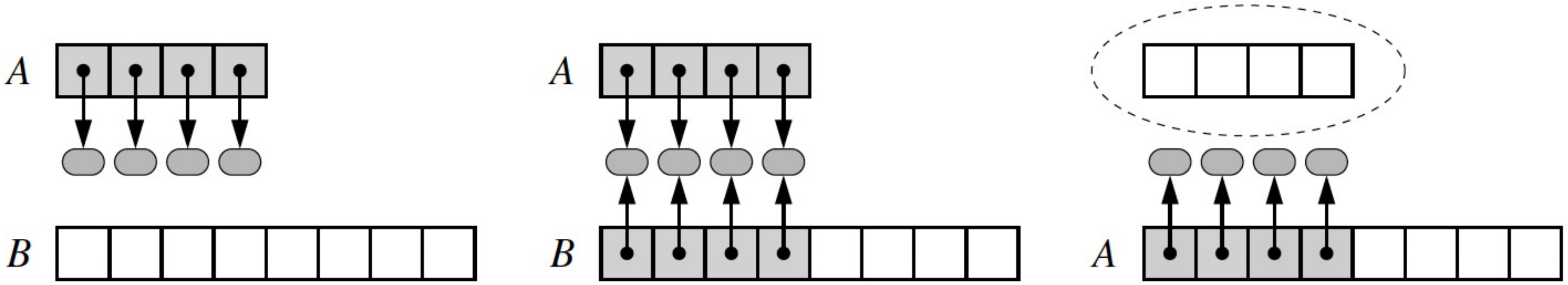


Dynamic Arrays

- **Dynamic Array:**

- The issue : how large the new array should be?

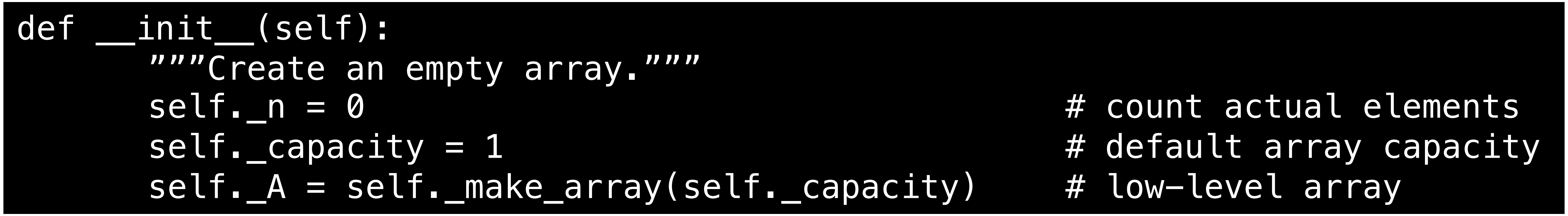
- A commonly used rule is for **the new array to have twice the capacity of the existing array** that has been filled.



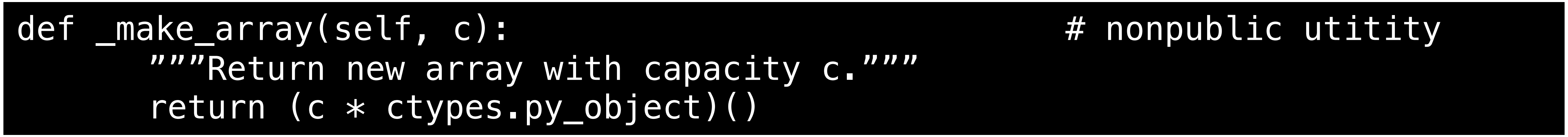
Dynamic Arrays

- **Dynamic Array:**

- Let us create the class “DynamicArray”.
- To manipulate low level arrays, we will **import ctypes**



```
def __init__(self):
      ”””Create an empty array.”””
      self._n = 0                                   # count actual elements
      self._capacity = 1                            # default array capacity
      self._A = self._make_array(self._capacity)    # low-level array
```



```
def _make_array(self, c):                           # nonpublic utitity
      ”””Return new array with capacity c.”””
      return (c ∗ ctypes.py_object)()
```

Dynamic Arrays

- **Dynamic Array:**

- Let us create the class “DynamicArray”.
- To manipulate low level arrays, we will **import ctypes**

```
def __len__(self):
       ”””Return number of elements stored in the array.”””
       return self._n
```

```
def __getitem__(self, k):
       ”””Return element at index k.”””
       if not 0 <= k < self._n:
             raise IndexError(‘invalid index’)
       return self._A[k]
```

Dynamic Arrays

- **Dynamic Array:**

- Let us create the class “DynamicArray”.
- To manipulate low level arrays, we will **import ctypes**

DynamicArray
_n **_capacity** **_A**

- **Exercise:**
- Define a **_resize(self, c)** that resize an array to capacity c

  <u>Reminder:</u>

  1. Allocate a new array B with larger capacity.

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

  3. Set $A = B$.

- Define an **append(self, obj)** method, that adds an object to the end of the array

Dynamic Arrays

- **Dynamic Array:**

- Define a **_resize(self, c)** that resize an array to capacity c

  <u>Reminder:</u>

  1. Allocate a new array B with larger capacity.

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

  3. Set $A = B$.

```
def _resize(self, c):                          # nonpublic utitity
       ”””Resize internal array to capacity c.”””

       B = self._make_array(c)                 # new (bigger) array

       for k in range(self._n):                # for each existing value
             B[k] = self._A[k]

       self._A = B                             # use the bigger array

       self._capacity = c
```

Dynamic Arrays

- **Dynamic Array:**

- Define an **append(self, obj)** method, that adds an object to the end of the array

```
def append(self, obj):
       ”””Add object to end of the array.”””

       if self._n == self._capacity:                 # not enough room
             self._resize(2 * self._capacity)        # so double capacity

       self._A[self._n] = obj

       self._n += 1
```

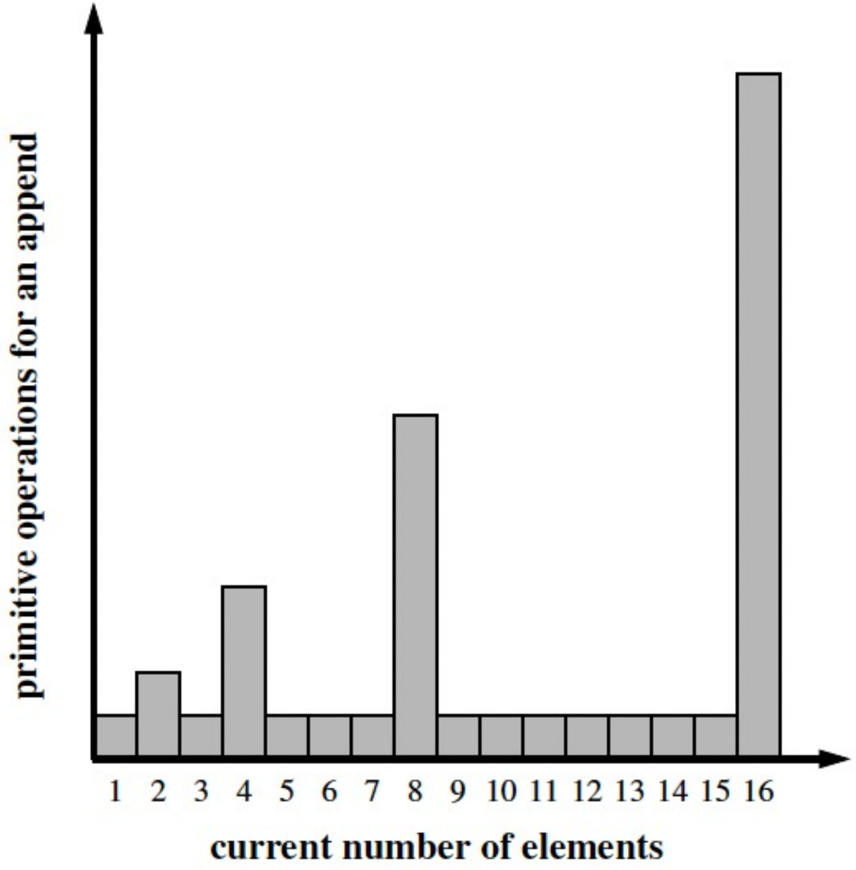
- **To do :**
- Write an __str__ method for the DynamicArray class
- Write a Python test program of that class

Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

- By doubling the capacity during an array replacement, our new array allows us to add **n** new elements before the array must be replaced again.

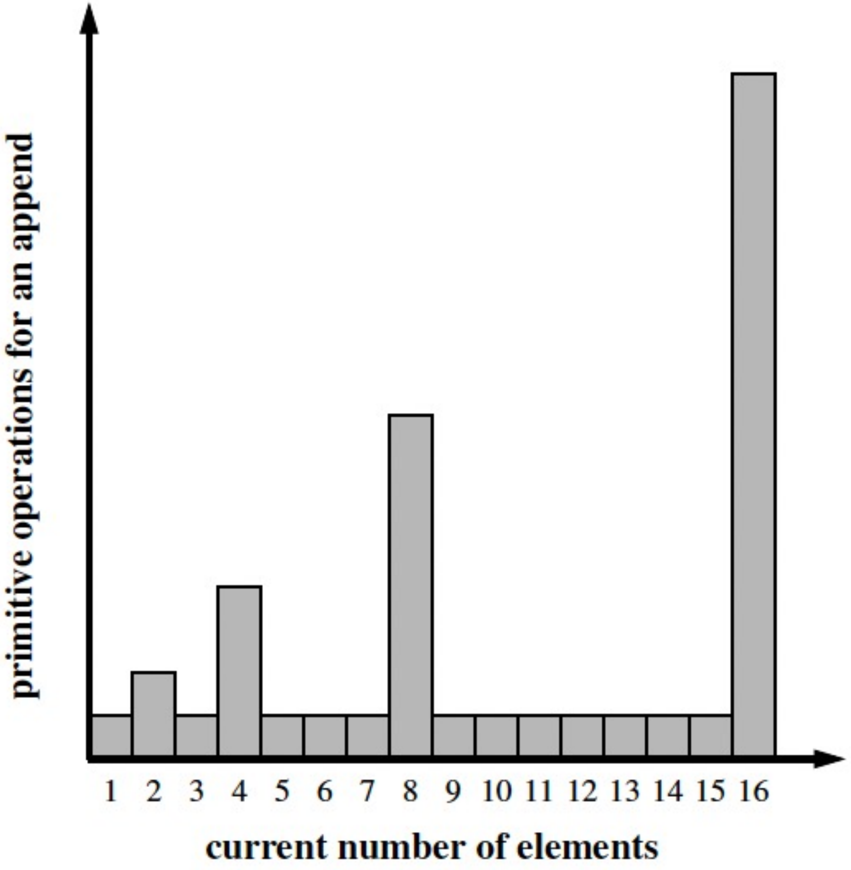
- There are many simple append operations for each expensive one.



Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

- **Proposition:** 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 $\boldsymbol{O(n)}$.



Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

- Amortized analysis:

- To perform an amortized analysis, we use an accounting technique where **we view the computer as a coin-operated appliance** that requires the payment of one 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" to pay for that operation's running time.

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

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

Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

- ## Justification:

- Let us assume :
  - 1 cyber-dollar is enough to pay for the execution of each append operation in S (excluding the time spent to grow the array)
  - Growing the array from size k to size $2k$ requires k cyber-dollars for the time spent initializing the new array
- => **We shall charge each append operations 3 cyber-dollars**
  - Each append operations is overcharged by 2 cyber dollars

  (Think of the 2 cyber-dollars profited in an insertion that does not grow the array as being "stored" with the cell in which the element was inserted)

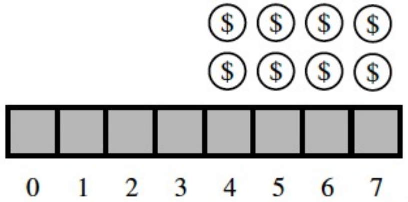
Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

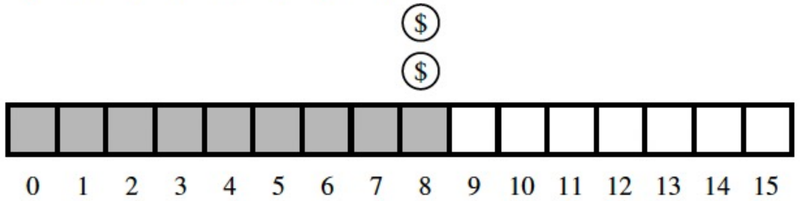
- Justification:

- An overflow occurs when the array S has 2^i elements, for some integer $i \geq 0$, and the size of the array used by the array representing S is 2^i .
  - => Doubling the size of the array requires 2^i cyber-dollars
  - **These cyber-dollars could be found stored in cells 2^{i-1} through $2^i - 1$**

An 8-cell array is full, with two cyber-dollars "stored" at cells 4 through



An append operation causes an overflow and a doubling of capacity.



Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table. Inserting the new element is paid for by one of the cyber-dollars charged to the current append operation, and the two cyber-dollars profited are stored at cell 8.

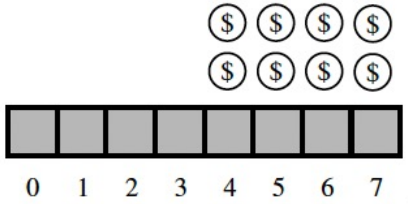
Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

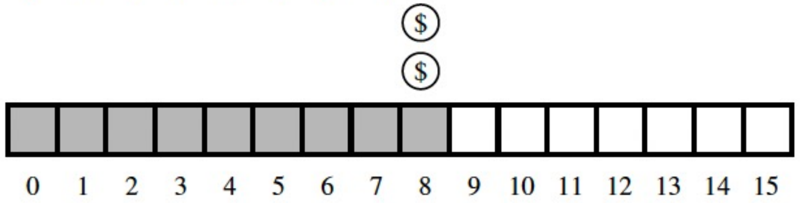
- Justification:

- The amortization scheme is **valid** (in which each operation is charged 3 cyber-dollars)
- For n append operations, we can pay using $3n$ cyber-dollars
- => The amortized **running time of each append operation is $\boldsymbol{O(1)}$**
  (for n append operations is $O(n)$)

An 8-cell array is full, with two cyber-dollars "stored" at cells 4 through



An append operation causes an overflow and a doubling of capacity.



Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table. Inserting the new element is paid for by one of the cyber-dollars charged to the current append operation, and the two cyber-dollars profited are stored at cell 8.

Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

- Geometric Increase in Capacity:

- The $O(1)$ amortized bound per operation can be proven for any geometrically increasing progression

- **Choice of the Geometric size:**
    - => Tradeoff: Runtime efficiency vs Memory usage

    - $Base = 2$ (doubling the size)
        - if the last insertion causes a resize event, the array essentially ends up twice as large as it needs to be.
    - $Base = 1.25$:
        - Increase the array by only 25% of its current size (Less risk wasting memory, More resizing events)
- In all cases, still $O(1)$ amortized bound

Dynamic Arrays

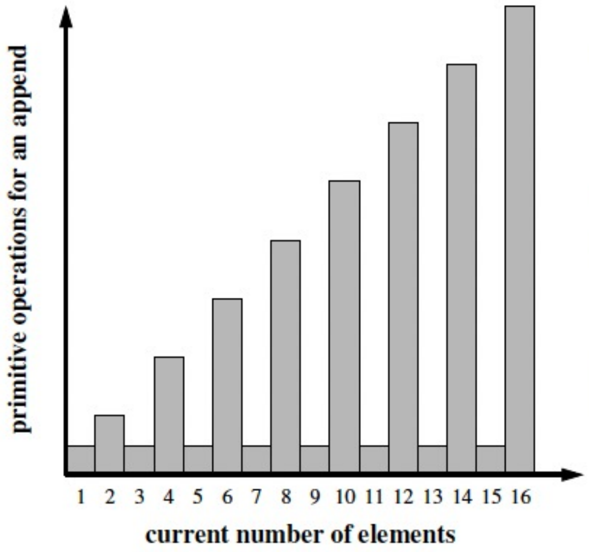
- **Dynamic Array** (Amortized Analysis):

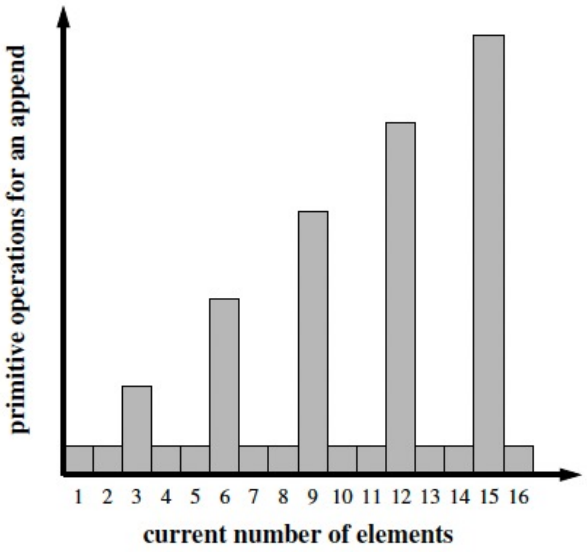
- Arithmetic Progression:

- **What if a constant number of additional cells are reserved each time an array is resized?**

- At an extreme, an increase of only 1 cell causes each append operation to resize the array, leading to a familiar $1 + 2 + 3 + \cdots . + n$ summation and $O(n^2)$ **overall cost**.

Assumes increase of 2





Assumes increase of 3

Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

- Arithmetic Progression:

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

- **Justification:**
- Let $c > 0$ represent the fixed increment in capacity that is used for each resize event.
- During the series of n append operations, time will have been spent initializing arrays of size c is $2c, 3c, \ldots, mc$ for $m = \lceil n/c \rceil$

=> The overall time would be proportional to $c + 2c + 3c + \cdots . + mc$.

$$\sum_{i=1}^{m} ci = c \cdot \sum_{i=1}^{m} i = c\frac{m(m+1)}{2} \geq c\frac{\frac{n}{c}(\frac{n}{c}+1)}{2} \geq \frac{n^2}{2c}.$$

→ $O(n^2)$ for the n append operations

Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):
- Arithmetic Progression:
- **Proposition:** Performing a series of n append operations on an initially empty dynamic array using a fixed increment with each resize takes $O(n^2)$ time.

- **To do :**
- Implement an append method with an arithmetic progression for the DynamicArray class
- Compare the running time for the different strategies
  - from time import time
  - time(): return time

Dynamic Arrays

- **Dynamic Array** (Amortized Analysis):

```
from time import time # import time function from time module

def compute average(n):
        ”””Perform n appends to an empty list and return average time elapsed.”””
        data = DynamicArray()
        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 per operation
```

Dynamic Arrays

- Shrinking an Array:

- The geometric increase in capacity when appending to a dynamic array
  => The final array size is guaranteed to be proportional to the overall number of elements.
  The data structure uses $O(n)$ memory.

- What if the datastructure using such array allows removing elements?

- There will **no** longer be a **proportional relationship** between the actual number of elements and the array capacity after many elements are removed.

- Repeated insertions may cause the underlying array to grow arbitrarily large

Dynamic Arrays

- Shrinking an Array:

- Idea: Shrink the underlying array occasionally

- Avoid the fact that the structure rapidly oscillate between growing and shrinking the underlying array.

- Shrinking Strategy:
  - The array capacity is **halved** whenever the number of actual elements falls **bellow one fourth** of that capacity
  - => The array capacity is at most four times the number of elements

  - **To do :**
  - Implement the shrinking strategy within the DynamicArray class

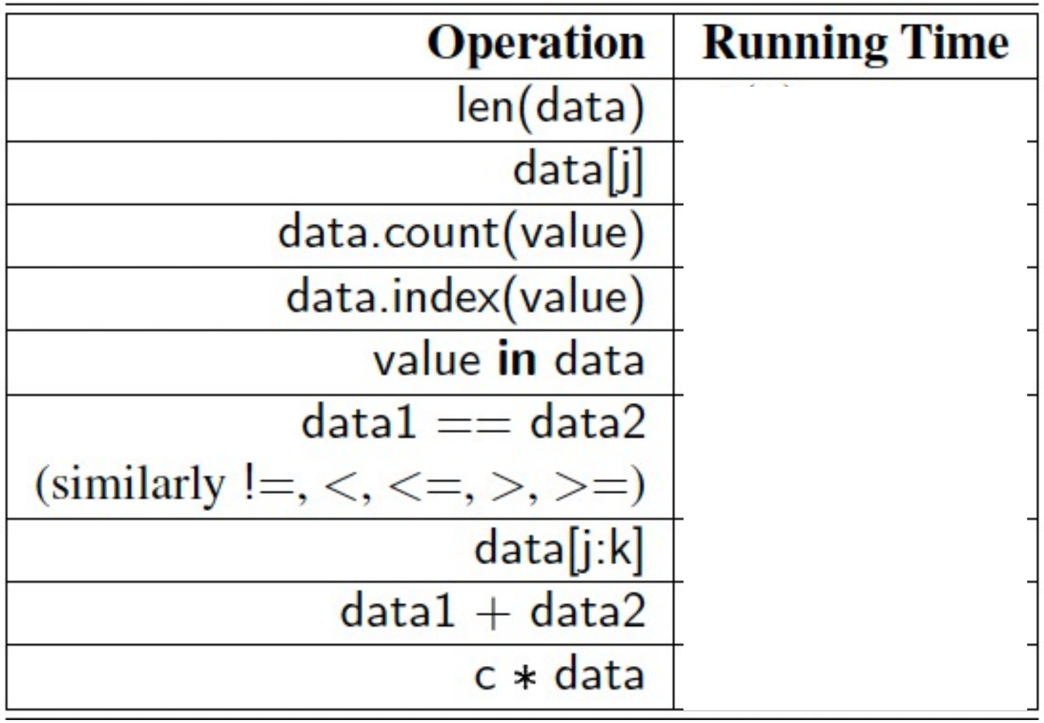
Performance of Sequential Types

- The nonmutating behaviors of the **list class** are precisely those that are supported by the **tuple class**.

- We note that tuples are typically more memory efficient than lists

  - There is no need for an underlying dynamic array with surplus capacity (Because of **immutability**)

Performance of Sequential Types



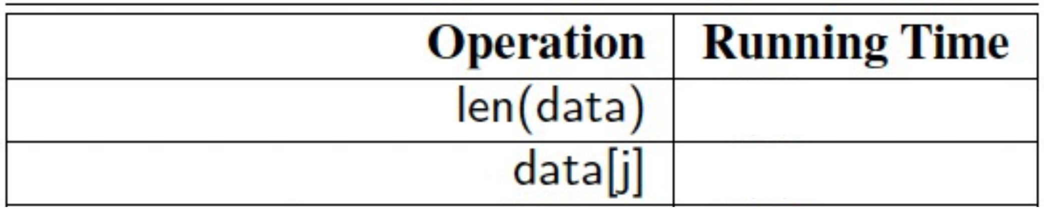
Operation	Running Time
len(data)	
data[j]	
data.count(value)	
data.index(value)	
value **in** data	
data1 == data2 (similarly !=, <, <=, >, >=)	
data[j:k]	
data1 + data2	
c * data	

*Asymptotic performance of the **nonmutating** behaviors of **the list and tuple classes**.*

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, k denote the leftmost index at which they disagree or else k = min(n_1,n_2).

Performance of Sequential Types



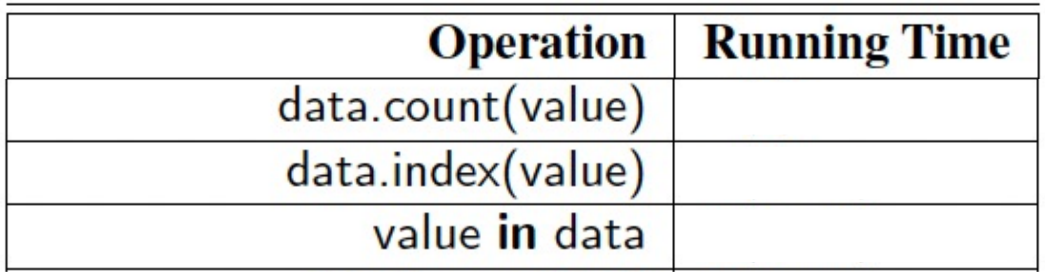
Operation	Running Time
len(data)	
data[j]	

Constant-Time Operations

- The length of an instance is returned in constant time because an instance explicitly maintains such state information.

- The constant-time efficiency of syntax $data[j]$ is assured by the underlying access into an array.

Performance of Sequential Types



Operation	Running Time
data.count(value)	
data.index(value)	
value **in** data	

For the containment check and index method,
k represents the index of the leftmost occurrence (with k = n if there is no occurrence).

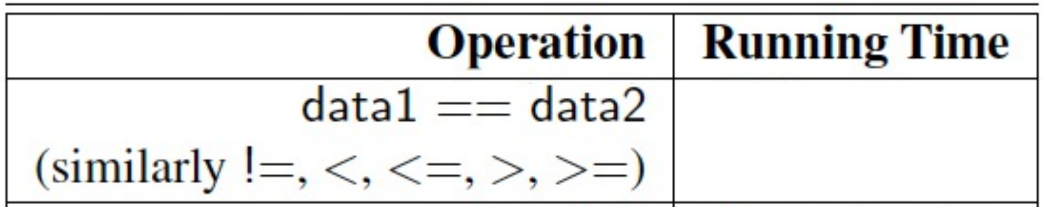
Searching for Occurrences of a Value

- Each of the count, index, and __contains__ methods proceed through iteration of the sequence from left to right.

- The loop for computing the count must proceed through the entire sequence

- The loops for checking containment of an element or determining the index of an element immediately exit once they find the leftmost occurrence of the desired value, if one exists.

Performance of Sequential Types



Operation	Running Time
data1 == data2 (similarly !=, <, <=, >, >=)	

For comparisons between two sequences,
k denote the leftmost index at which they disagree or else k = min(n_1,n_2).

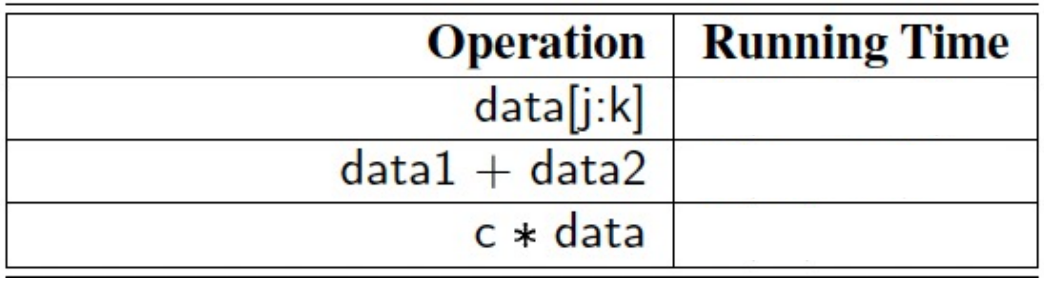
Lexicographic comparisons

- In the worst case, evaluating such a condition requires an iteration taking time proportional to the length of the shorter of the two sequences

- However, in some cases the result of the test can be evaluated more efficiently.

- For example, if evaluating [7, 3, ...] < [7, 5, ...], it is clear that the result is True without examining the remainders of those lists, because the second element of the left operand is strictly less than the second element of the right operand.

Performance of Sequential Types



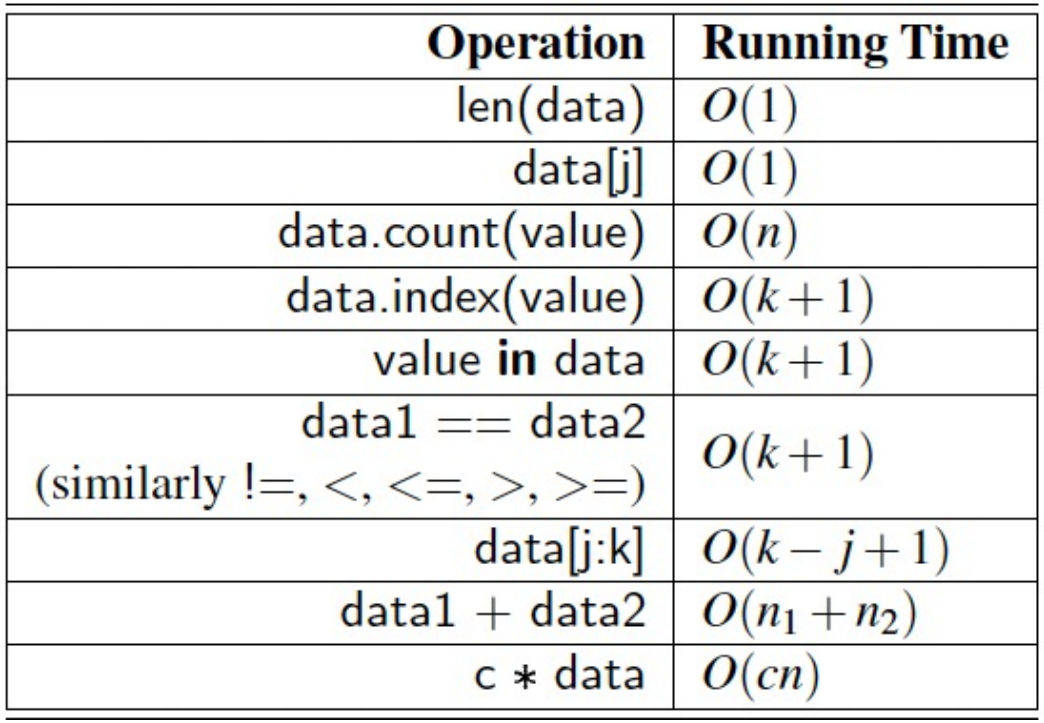
Operation	Running Time
data[j:k]	
data1 + data2	
c * data	

Creating New Instances

- The running time depends on the construction and initialization of the new result,

- => Therefore, the asymptotic behavior is proportional to the length of the result.

Performance of Sequential Types



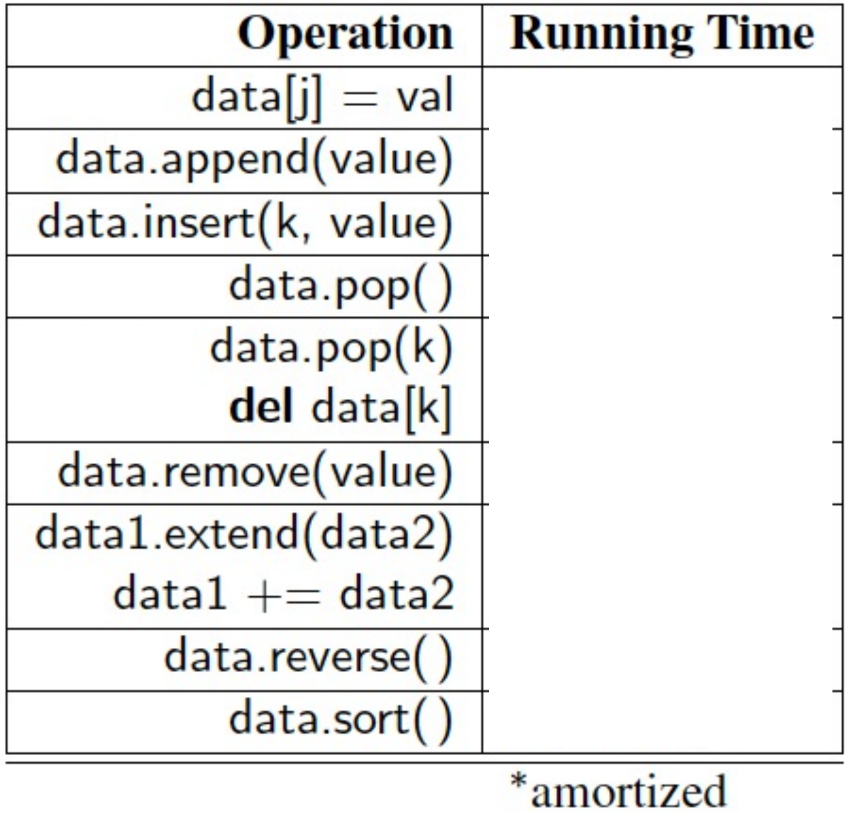
Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Asymptotic performance of the ***nonmutating*** *behaviors of* ***the list and tuple classes****.*

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, k denote the leftmost index at which they disagree or else k = min(n_1,n_2).

Performance of Sequential Types



Operation	Running Time
data[j] = val	
data.append(value)	
data.insert(k, value)	
data.pop()	
data.pop(k) **del** data[k]	
data.remove(value)	

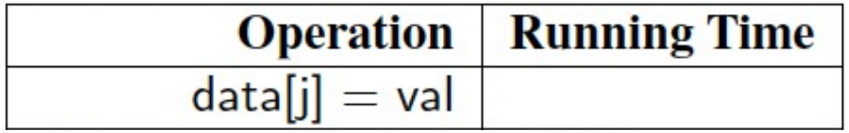
data1.extend(data2)
data1 += data2	

data.reverse()	
data.sort()	

*amortized

*Asymptotic performance of the **mutating** behaviors of **the list classes**.*

Performance of Sequential Types



Operation	Running Time
data[j] = val	

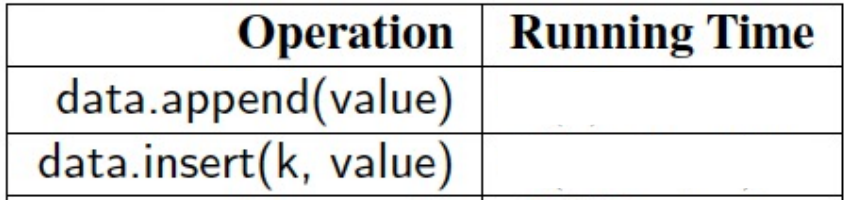
Assignment

- It is supported by the special __setitem__ method.

- This operation has worst-case $O(1)$ running time because it simply replaces one element of a list with a new value.

- No other elements are affected, and the size of the underlying array does not change.

Performance of Sequential Types

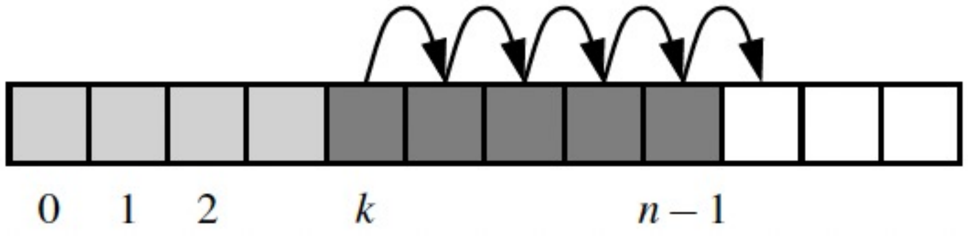


Operation	Running Time
data.append(value)	
data.insert(k, value)	

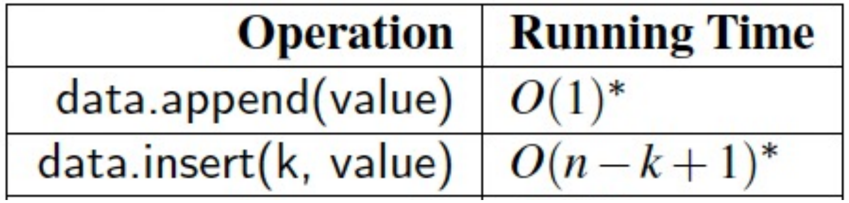
Adding Elements to a List

- The append(value) method: In the worst case, it requires $O(n)$ time because the underlying array is resized, but it uses $O(1)$ time in the amortized sense.

- The method insert(k, value): inserts a given value into the list at index $0 \le k \le n$ while shifting all subsequent elements back one slot to make room.



Performance of Sequential Types

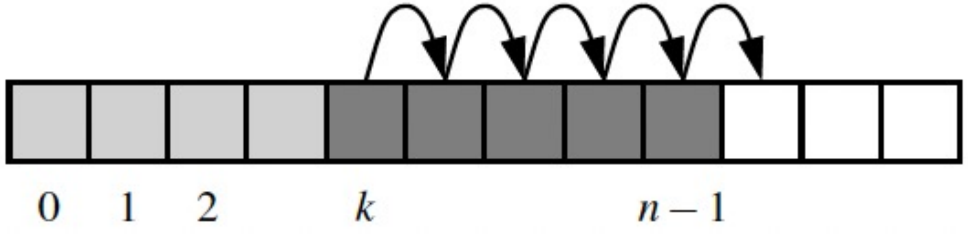


Operation	Running Time
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$

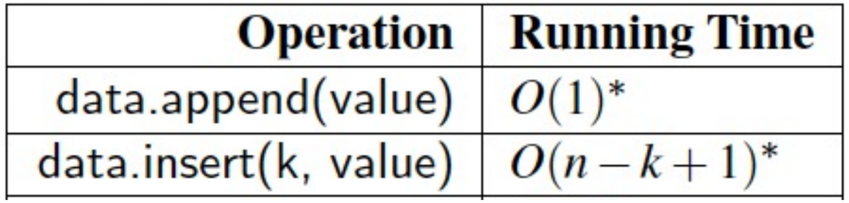
Adding Elements to a List

- **To do : Provide an implementation of the insert method within the DynamicArray class**
- The method insert(k, value): inserts a given value into the list at index $0 \leq k \leq n$ while shifting all subsequent elements back one slot to make room.

- Note that the addition of one element may require a resizing of the dynamic array.



Performance of Sequential Types

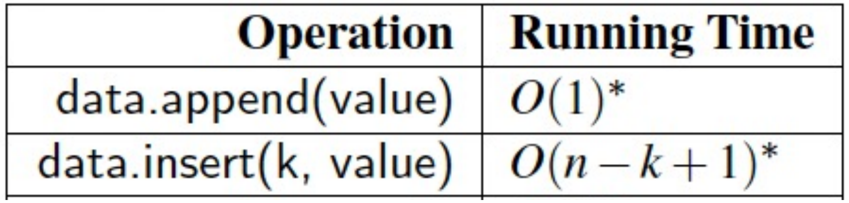


Operation	Running Time
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$

Adding Elements to a List

```
def insert(self, k, value):
       ”””Insert value at index k, shifting subsequent values rightward.”””
       # (for simplicity, we assume 0 <= k <= n in this version)
       if self._n == self._capacity:                      # not enough room
              self._resize(2 ∗ self._capacity)            # so double capacity
       for j in range(self._n, k, –1):                    # shift rightmost first
              self._A[j] = self._A[j–1]
       self._A[k] = value                                 # store newest element
       self._n += 1
```

Performance of Sequential Types

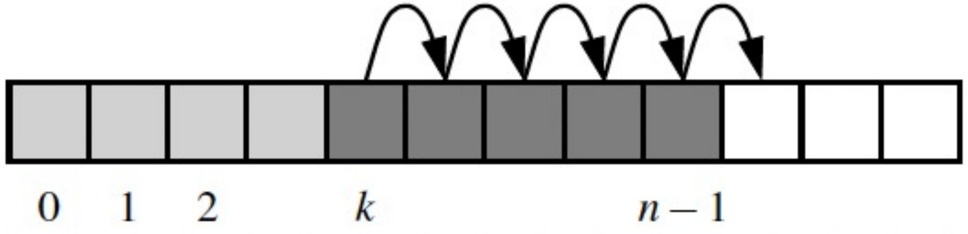


Operation	Running Time
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$

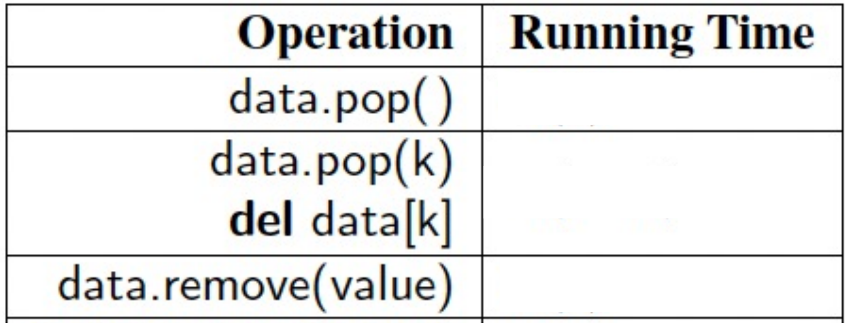
Adding Elements to a List

- The resizing requires $\mathrm{O}(n)$ worst-case time but only $O(1)$ amortized time, as per append.

- Shifting elements leads to an amortized $O(n-k+1)$ performance for inserting at index k.



Performance of Sequential Types

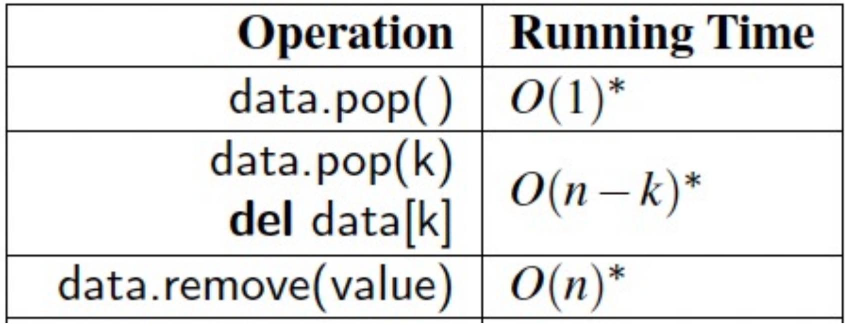


Operation	Running Time
data.pop( )	
data.pop(k) **del** data[k]	
data.remove(value)	

Removing Elements from a List

- Python’s list class offers several ways to remove an element from a list.
  - **pop() :** removes the last element from a list.
    - All other elements remain in their original location.
    - The bound is amortized because Python will occasionally shrink the underlying dynamic array to conserve memory.

Performance of Sequential Types

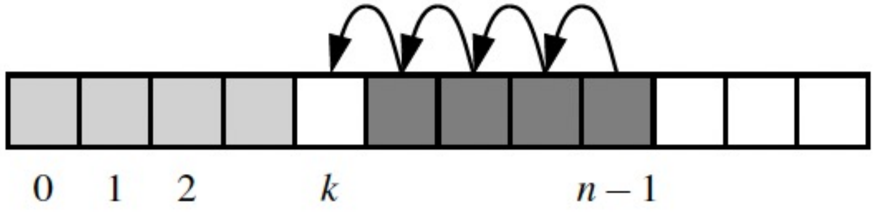


Operation	Running Time
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$

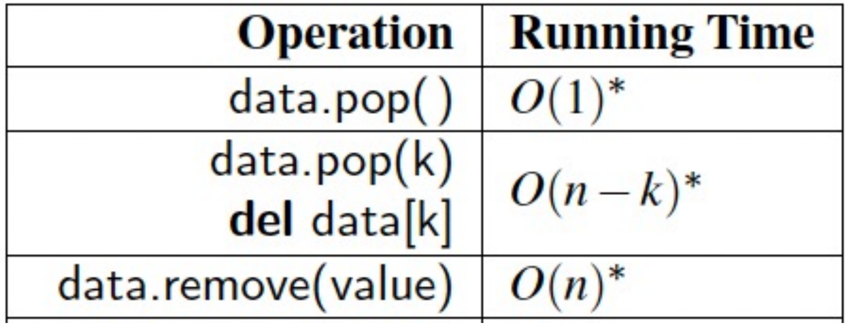
Removing Elements from a List

- Python’s list class offers several ways to remove an element from a list.
    - **pop(k)** : removes the element that is at index k < n of a list, shifting all subsequent elements leftward to fill the gap that results from the removal.

        - pop(0) is the most expensive call, using Ω(n) time.



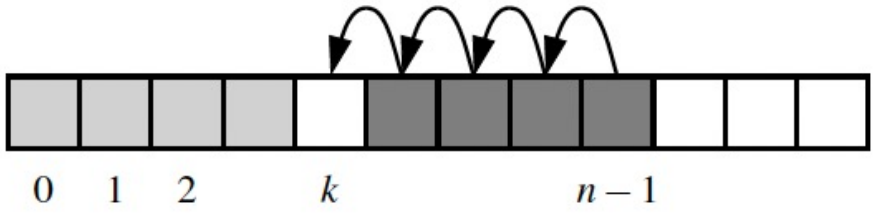
Performance of Sequential Types



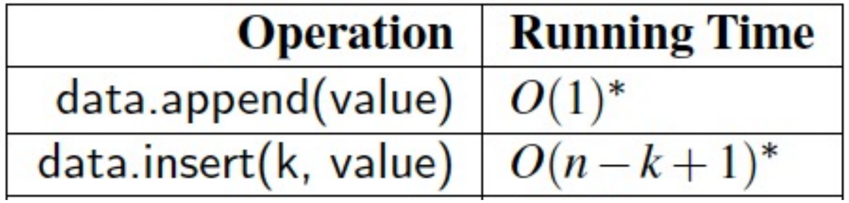
Operation	Running Time
data.pop()	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$

Removing Elements from a List

- Python’s list class offers several ways to remove an element from a list.
    - **remove(value)** : removes the first occurrence of the *value* from a list
    - Every call requires $O(n)$ time.
        - Searches from the beginning until finding the value at index k
        - Iterates from k to the end in order to shift elements leftward.
    - **To do : Provide an implementation of the remove method within the DynamicArray class**



Performance of Sequential Types

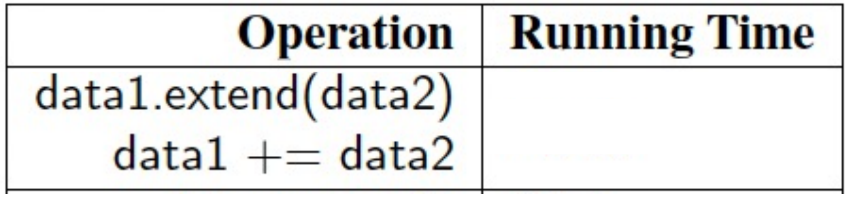


Operation	Running Time
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$

Removing Elements from a List:

```
def remove(self, value):
        ”””Remove first occurrence of value (or raise ValueError).”””
       # note: we do not consider shrinking the 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 one less item
                    return                           # exit immediately
             raise Value Error(‘value not found’)    # only reached if no match
```

Performance of Sequential Types



Operation	Running Time
data1.extend(data2)	
data1 += data2	

Extending a List

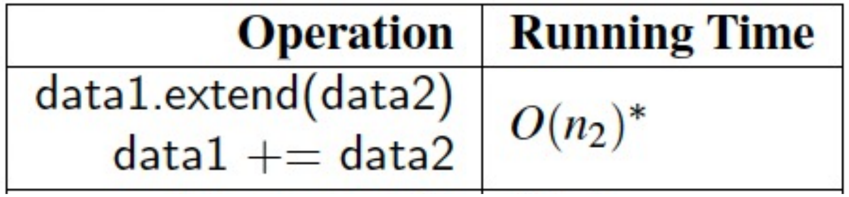
- **extend() :** is used to add all elements of one list to the end of a second list.

```
data.extend(other)
```

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

- Same result
- The running time is proportional to the length of the other list
- Amortized because the underlying array for the first list may be resized to accommodate the additional elements.

Performance of Sequential Types



Operation	Running Time
data1.extend(data2)	$O(n_2)^*$
data1 += data2	

Extending a List

- **extend() :** is used to add all elements of one list to the end of a second list.

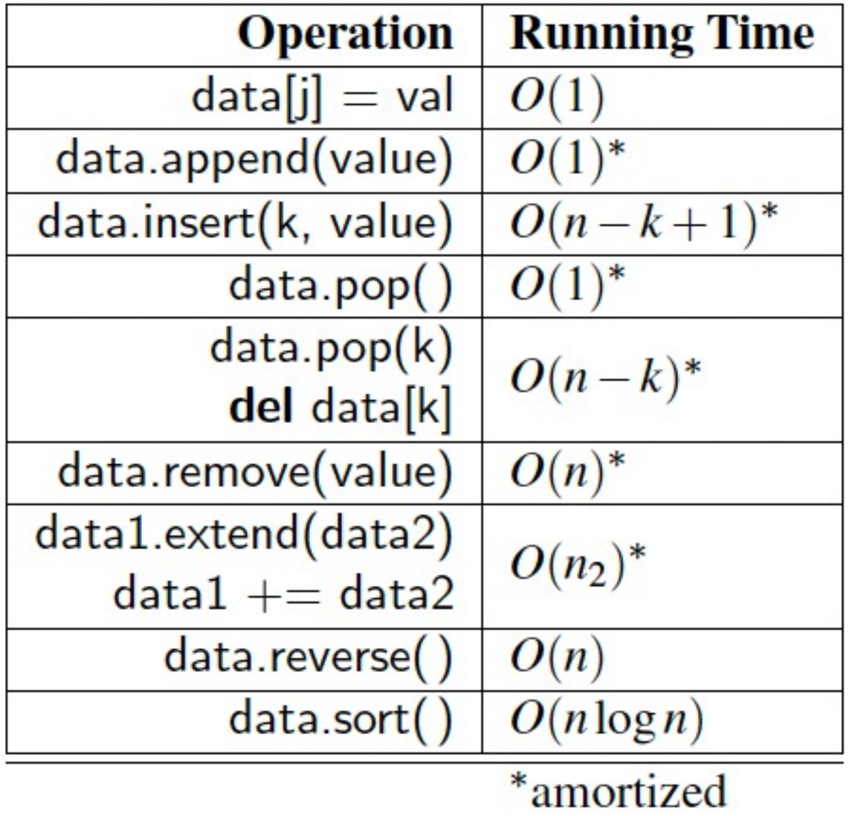
```
data.extend(other)
```

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

In practice, the **extend method is preferable to repeated calls**:

- Python methods are often implemented natively in a compiled language (rather than as interpreted Python code).
- Less overhead to a single function call that accomplishes all the work, versus many individual function calls.
- Increased efficiency of extend comes from the fact that the resulting size of the updated list can be calculated in advance (Avoid resizing).

Performance of Sequential Types



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 classes**.*

- **To do :**
- Provide an implementation of the **reverse** method within the DynamicArray class

- Provide an implementation of the **sort** method within the DynamicArray class

Python’s String Class

- Strings are very important in Python.

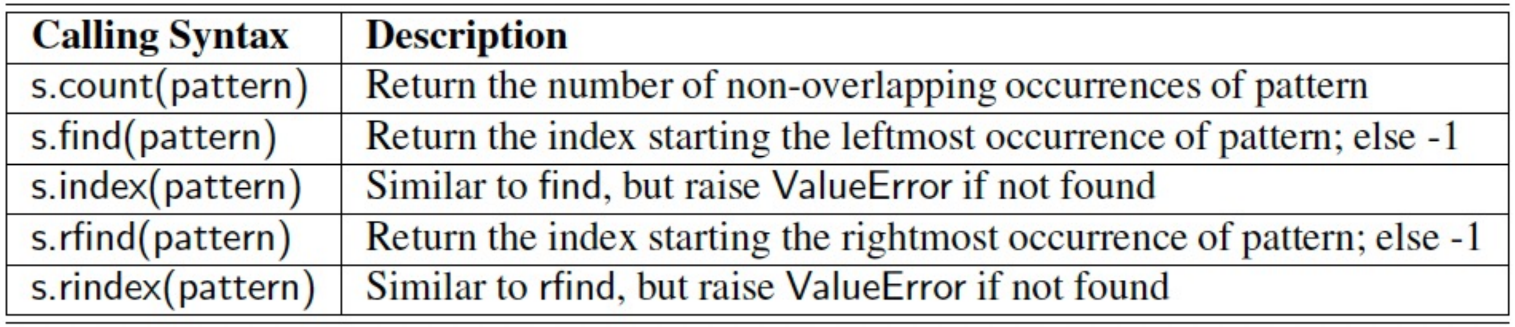
- Strings are particularly important in most programming applications, as text is often used for input and output.

- The built-in **str** class represents strings based upon the Unicode international character set (**16-bit**).

- The **str class** comes with multiple methods

Python’s String Class

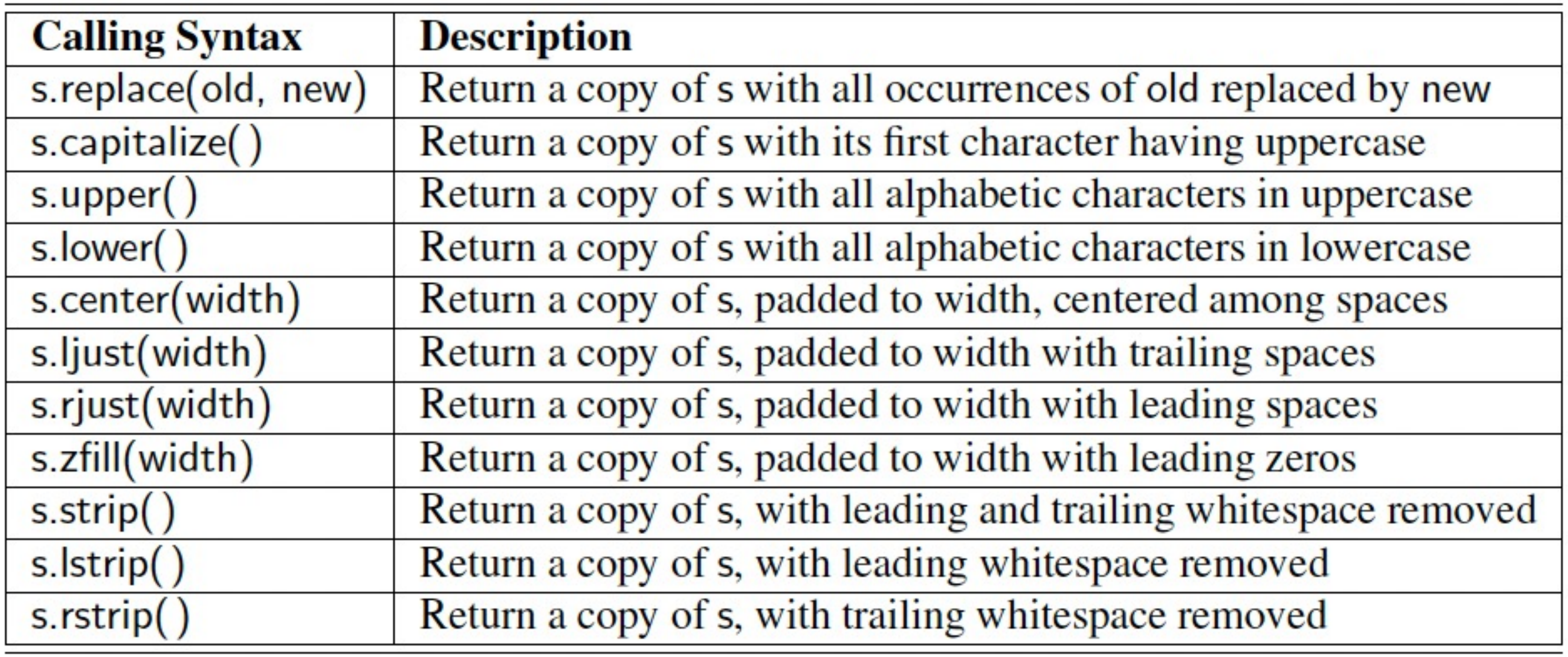
- Methods that search for Substrings



Calling Syntax	**Description**
s.count(pattern)	Return the number of non-overlapping occurrences of pattern
s.find(pattern)	Return the index starting the leftmost occurrence of pattern; else -1
s.index(pattern)	Similar to find, but raise ValueError if not found
s.rfind(pattern)	Return the index starting the rightmost occurrence of pattern; else -1
s.rindex(pattern)	Similar to rfind, but raise ValueError if not found

Python’s String Class

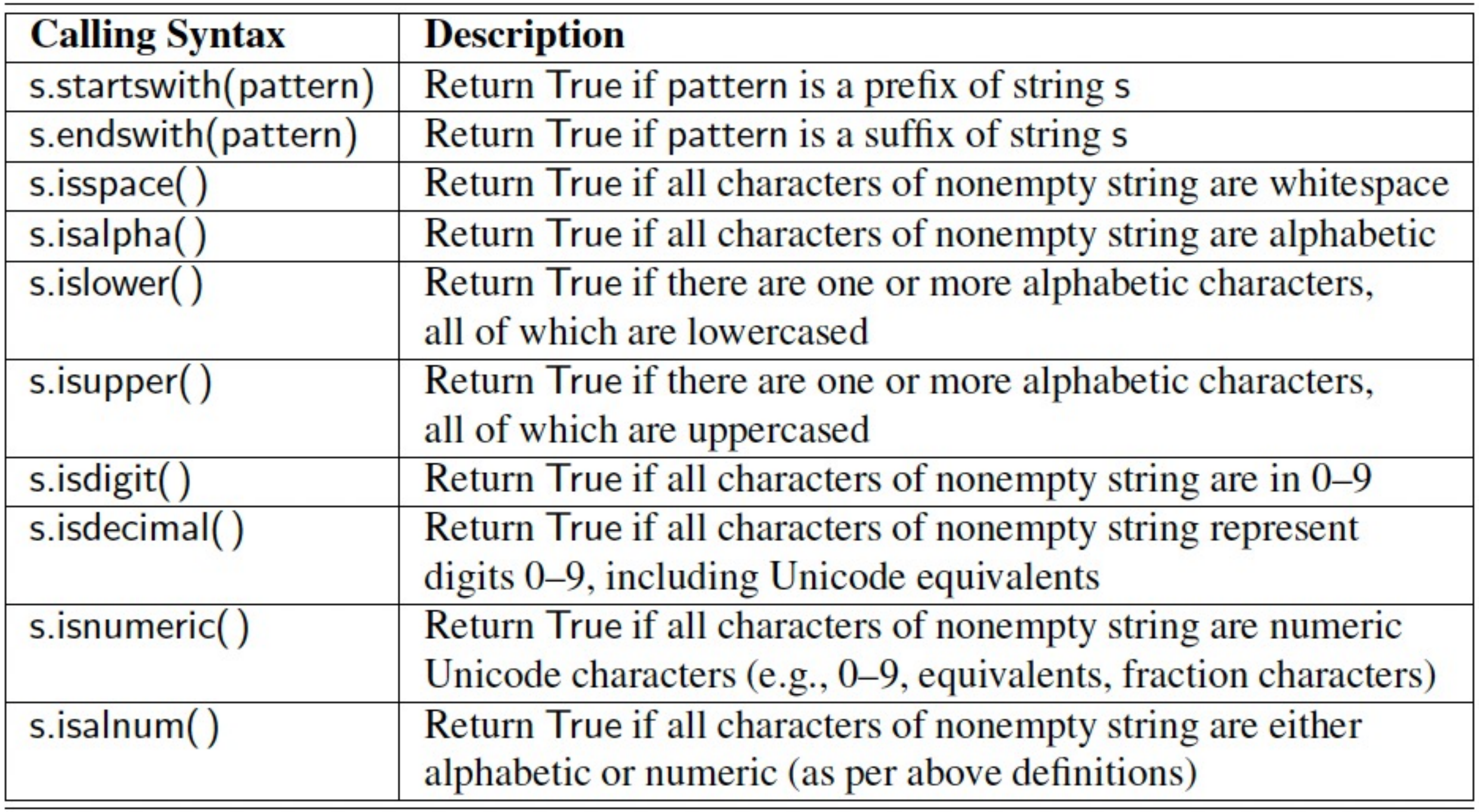
- Methods that constructs related Strings



Calling Syntax	Description
s.replace(old, new)	Return a copy of s with all occurrences of old replaced by new
s.capitalize( )	Return a copy of s with its first character having uppercase
s.upper( )	Return a copy of s with all alphabetic characters in uppercase
s.lower( )	Return a copy of s with all alphabetic characters in lowercase
s.center(width)	Return a copy of s, padded to width, centered among spaces
s.ljust(width)	Return a copy of s, padded to width with trailing spaces
s.rjust(width)	Return a copy of s, padded to width with leading spaces
s.zfill(width)	Return a copy of s, padded to width with leading zeros
s.strip( )	Return a copy of s, with leading and trailing whitespace removed
s.lstrip( )	Return a copy of s, with leading whitespace removed
s.rstrip( )	Return a copy of s, with trailing whitespace removed

Python’s String Class

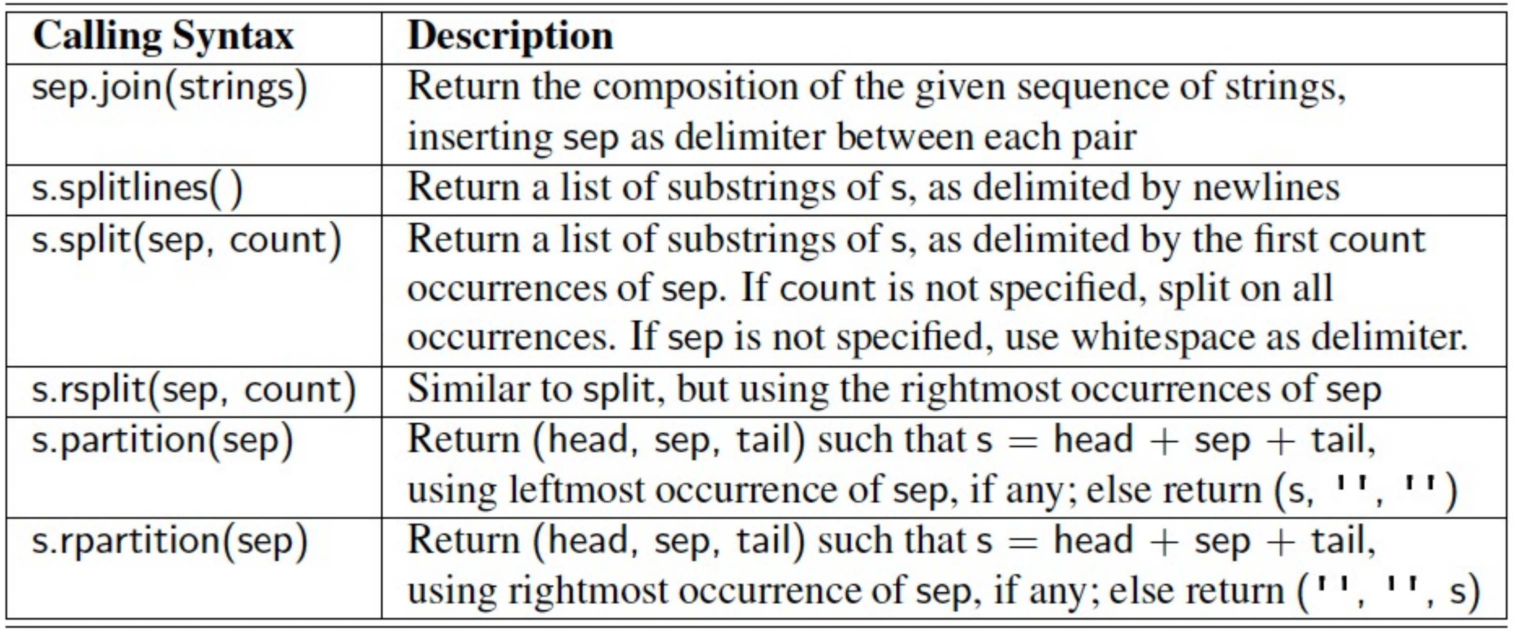
- Methods that tests Boolean conditions



Calling Syntax	Description
s.startswith(pattern)	Return True if pattern is a prefix of string s
s.endswith(pattern)	Return True if pattern is a suffix of string s
s.isspace( )	Return True if all characters of nonempty string are whitespace
s.isalpha( )	Return True if all characters of nonempty string are alphabetic
s.islower( )	Return True if there are one or more alphabetic characters, all of which are lowercased
s.isupper( )	Return True if there are one or more alphabetic characters, all of which are uppercased
s.isdigit( )	Return True if all characters of nonempty string are in 0–9
s.isdecimal( )	Return True if all characters of nonempty string represent digits 0–9, including Unicode equivalents
s.isnumeric( )	Return True if all characters of nonempty string are numeric Unicode characters (e.g., 0–9, equivalents, fraction characters)
s.isalnum( )	Return True if all characters of nonempty string are either alphabetic or numeric (as per above definitions)

Python’s String Class

- Splitting and Joining Strings



Calling Syntax	Description
sep.join(strings)	Return the composition of the given sequence of strings, inserting sep as delimiter between each pair
s.splitlines( )	Return a list of substrings of s, as delimited by newlines
s.split(sep, count)	Return a list of substrings of s, as delimited by the first count occurrences of sep. If count is not specified, split on all occurrences. If sep is not specified, use whitespace as delimiter.
s.rsplit(sep, count)	Similar to split, but using the rightmost occurrences of sep
s.partition(sep)	Return (head, sep, tail) such that s = head + sep + tail, using leftmost occurrence of sep, if any; else return (s, '', '')
s.rpartition(sep)	Return (head, sep, tail) such that s = head + sep + tail, using rightmost occurrence of sep, if any; else return ('', '', s)

Python’s String Class

- Comments on some String usage

- Notation:
  - n: The length of a String
  - m: The length of pattern (if any)

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

- Method that **tests Boolean conditions** of a String (e.g., islower) take $\boldsymbol{O(n)}$ **time**
  - Examine all n characters in the worst case
  - isLower can immediately return False if the 1st character is uppercased
  - Same for comparison operators (e.g., ==, <,..)

Python’s String Class

- Comments on some String usage

- Notation:
  - n: The length of a String
  - m: The length of pattern (if any)

- Pattern Matching:
  - **Aim**: Finding a string pattern within a larger string
  - **Methods**: __contains__, find, index, count, replace, split,…..
  - A naïve implementation:
    - Consider the $n - m + 1$ possible starting indices for the pattern, and
    - Spend $O(m)$ time at each starting position checking if the pattern matches
    - **=> $\boldsymbol{O(mn)}$ time**
  - There exists algorithms for finding pattern of length m within a longer string of length n is **$\boldsymbol{O(n)}$ time**

Python’s String Class

- Comments on some String usage

- **Composing Strings**:
    - There exists several approaches for composing large strings.

    - Exercise:
    - Define a function that given a large string, produces a new string that contains only the alphabetic characters of the original string (e.g., with spaces, numbers, and punctuation removed).
    - Note: The **isaplha()** method of the **str** class, check whether a character is alphabetic or not.

Python’s String Class

- Comments on some String usage

- **Composing Strings**:

```
letters = ‘’                      # start with empty string
for c in document:
      if c.isalpha():
             letters += c         # concatenate alphabetic character
```

  - This code fragment accomplishes the goal

  - It is terribly inefficient

Python’s String Class

- Comments on some String usage

- **Composing Strings**:

```
letters = ‘’                      # start with empty string
for c in document:
      if c.isalpha():
             letters += c         # concatenate alphabetic character
```

- Strings are immutable
  - Letters += c : computes the concatenation letters + c and construct a new string
  - If the result has n characters, the series of concatenations would take O(n^2) time (1 + 2 + ….+ n)

Python’s String Class

- Comments on some String usage

- **Composing Strings**:

```
letters = ‘’                       # start with empty string
for c in document:
       if c.isalpha():
              letters += c         # concatenate alphabetic character
```

- Python interpreter have developed an **optimization** to allow such code to complete in linear time:
  - Python check if there were no other references to the String in question, it could inmplement += , by **directly mutating the string** (as a dynamic array).
  - Note: Python interpreter already maintains what are known as ***reference counts*** for each object.

Python’s String Class

- Comments on some String usage

- **Composing Strings**:

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

- A more standrad Python idiom to guarantee linear time composition of a string
    - Use of a **temporary list** to store individual characters.
    - Then, rely of the **join method of str class** to compose the final result.
    - => $\boldsymbol{O(n)}$ **time**

Python’s String Class

- Comments on some String usage

- **Composing Strings**:

- Improvements of the practical execution time:

- **Using list comprehension**

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

- **Using generator comprehension**

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

Using Array-based Sequences

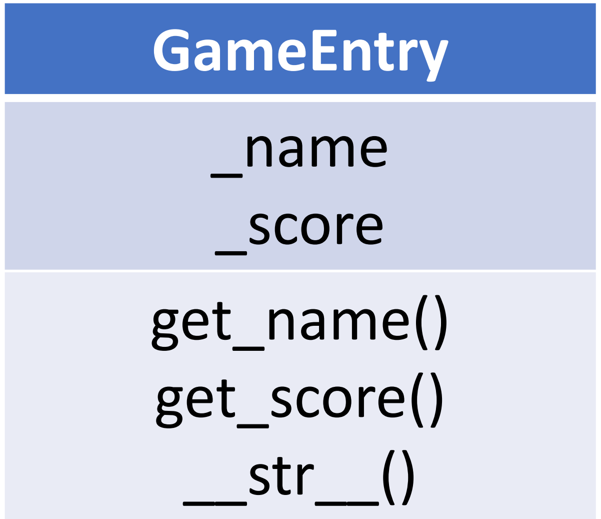
- We will study 3 use cases of array-based sequences:
    - Storing High Scores for a Game
    - Sorting a Sequence
    - Simple Cryptography

Using Array-based Sequences

- Storing High Scores for a Game:

- Objectif:
  - Store a sequence of high score entries for a video game.
  - Each entry is defined by the score and the name of the person earning this score

  - To do: Define a GameEntry class



GameEntry

_name
_score

get_name()
get_score()
__str__()

```
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) # e.g., (Bob, 98)
```

Using Array-based Sequences

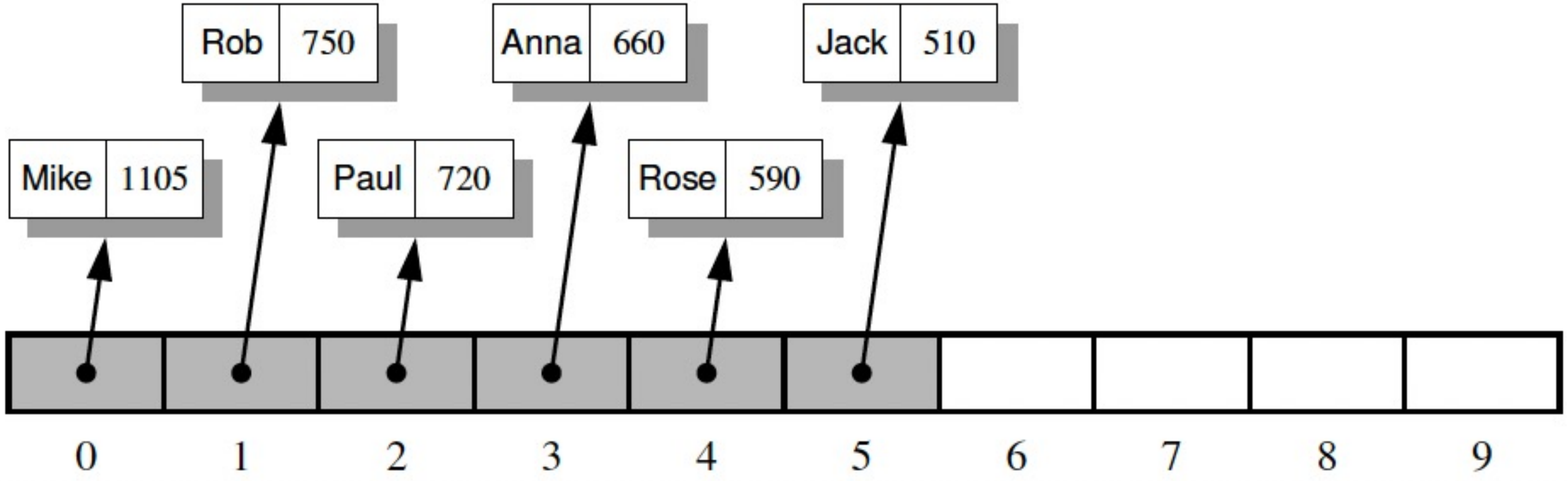
- Storing High Scores for a Game:

- Objectif:
  - Store a sequence of high score entries for a video game.
  - Each entry is defined by the score and the name of the person earning this score.

  - To do:
    - Define a GameEntry class
    - Define a ScoreBoard class that maintains the list of n GameEntry with the highest scores
      - The entries are ordered from highest to lowest score, starting at index 0 of the list.

Using Array-based Sequences

- Storing High Scores for a Game:
    - To do:
        - Define a GameEntry class
        - Define a ScoreBoard class that maintains the list of GameEntry with the highest scores (up to a maximum capacity)
            - The entries are ordered from highest to lowest score, starting at index 0 of the list.



Using Array-based Sequences

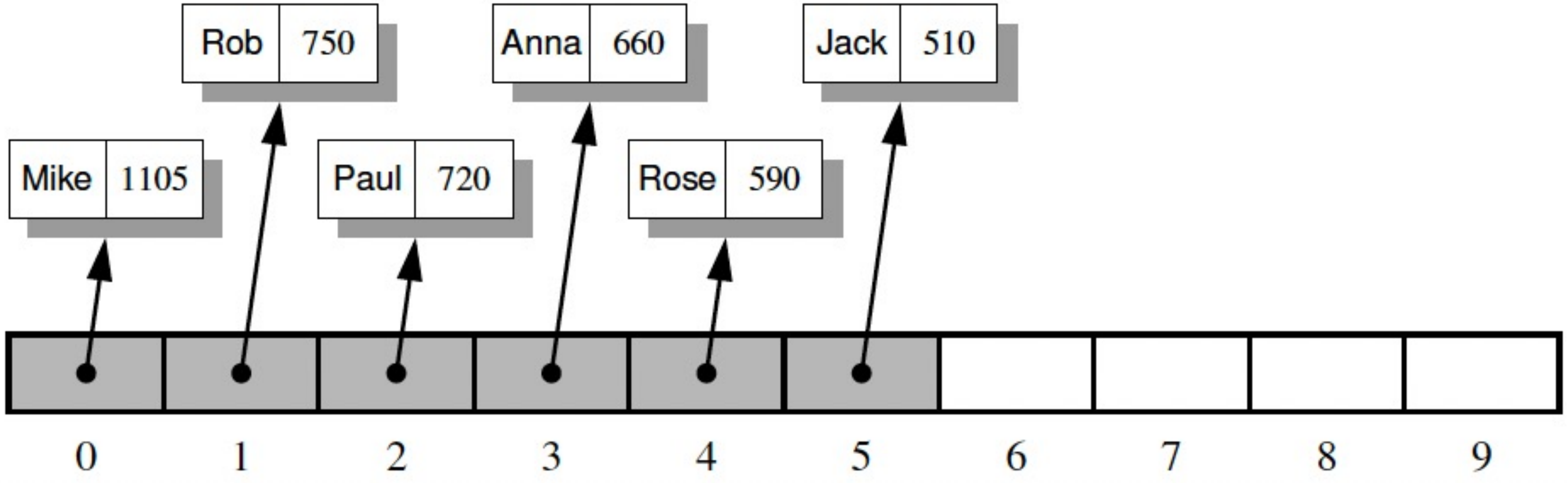
- Storing High Scores for a Game:
  - To do:
    - Define a GameEntry class
    - Define a ScoreBoard class that maintains the list of GameEntry with the highest scores (up to a maximum capacity)

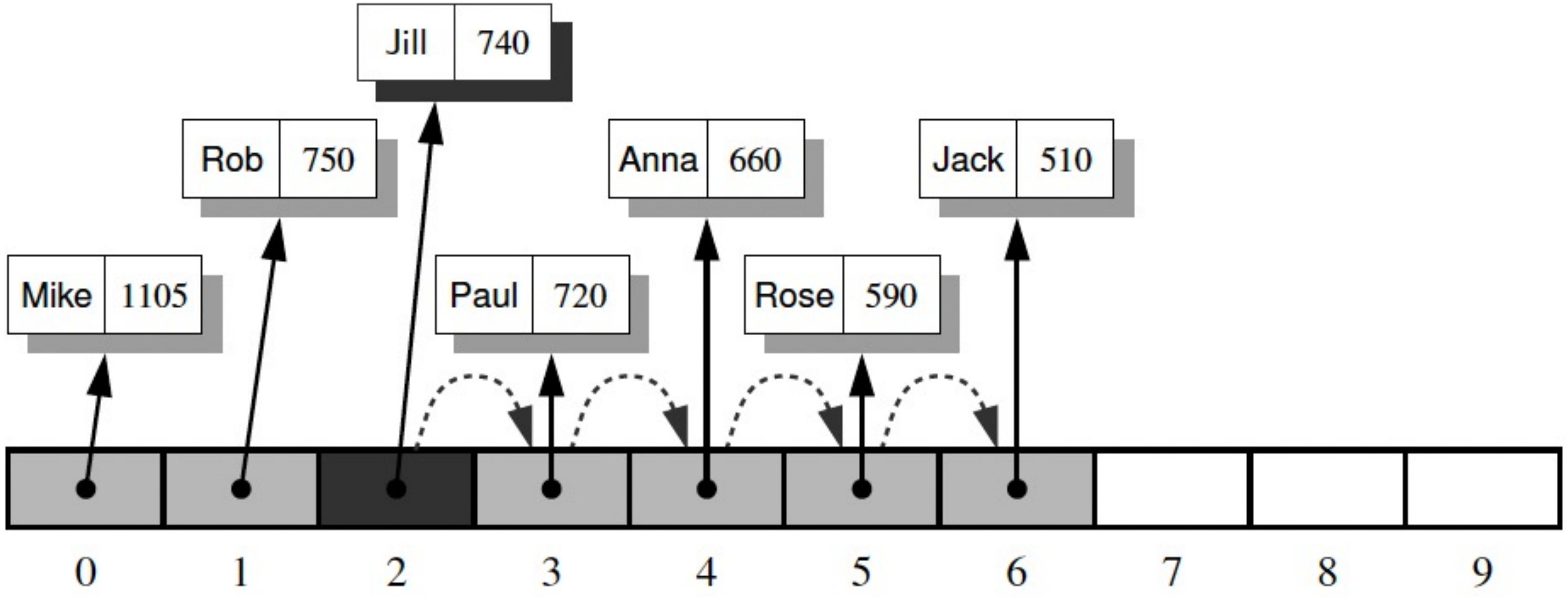
ScoreBoard
_board _n

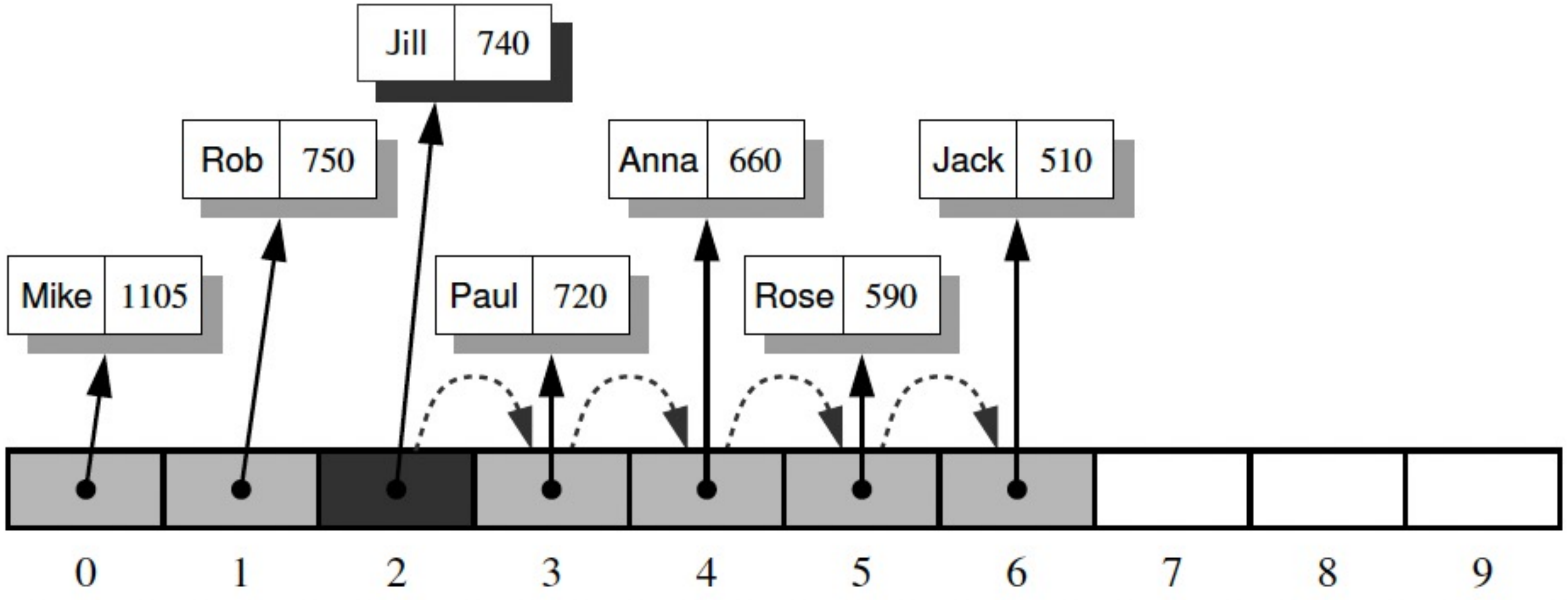
add(entry)
__getitem__(k)
__str__()

- **_board:** a Python list in order to manage the GameEntry instances that represent the high scores
- **_n :** Number of actual entries currently in the list
- **__getitem__(k) :** to retrieve an entry at a given index k (board[k])
- **add(entry):** Add a new entry to the scoreboard

```
class Scoreboard:

    def __init__(self, capacity=10):
    ”””Initialize scoreboard with given maximum capacity. All entries are 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))
```


Rob 750
Anna 660
Jack 510
Mike 1105
Paul 720
Rose 590
0
1
2
3
4
5
6
7
8
9


Jill 740
Rob 750
Anna 660
Jack 510
Mike 1105
Paul 720
Rose 590
0
1
2
3
4
5
6
7
8
9



<u>If the board is not full</u>

Any new entry will be retained

<u>Once the board is full</u>

A new entry is only retained if it is strictly better than one of the other scores

<u>if a new score is considered</u>

Increase the count of active scores (_n) unless the board is at full capacity

Correctly place a new entry within the list by:

Shifting any inferior score on spot lower (with the least score being dropped entirely when the scoreboard is full)

```
def add(self, entry):
       score = entry.get_score()

       # Does new entry qualify as a high score?
       # answer is yes if board not full or score is higher than 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                       # so overall number 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
```

Using Array-based Sequences

- We will study 3 use cases of array-based sequences:
  - Storing High Scores for a Game
  - Sorting a Sequence

Using Array-based Sequences

- Sorting a sequence:

- **Objectif** : Given an Array of n comparable elements, rearrange the array in nondecreasing order of its elements.

- There exists several sorting algorithms

- We will study on of the simplest one: **Insertion Sort**

Using Array-based Sequences

- Sorting a sequence:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.

Values from the unsorted part are picked and placed at the correct position in the sorted part.

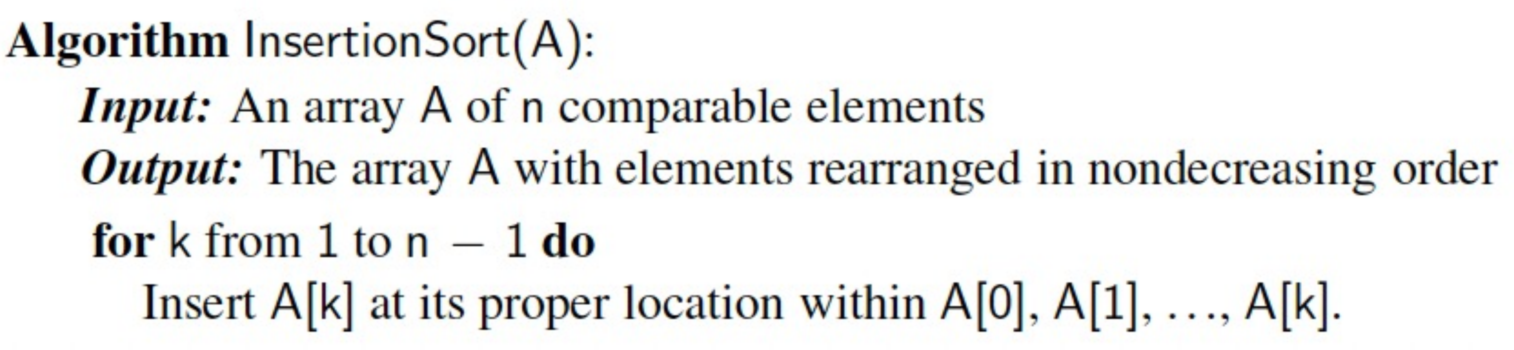
The array is virtually split into a sorted and an unsorted part.

Using Array-based Sequences

- Sorting a sequence:
    - We start with the first element in the array
        - One element by itself is already sorted.
    - Then, we consider the next element in the array
        - If it is smaller than the first, we swap them
    - Next, we consider the third element
        - We swap if leftward until it is in its proper order with the first two elements
    - Then, the fourth element
    - …….
    - Until the whole array is sorted

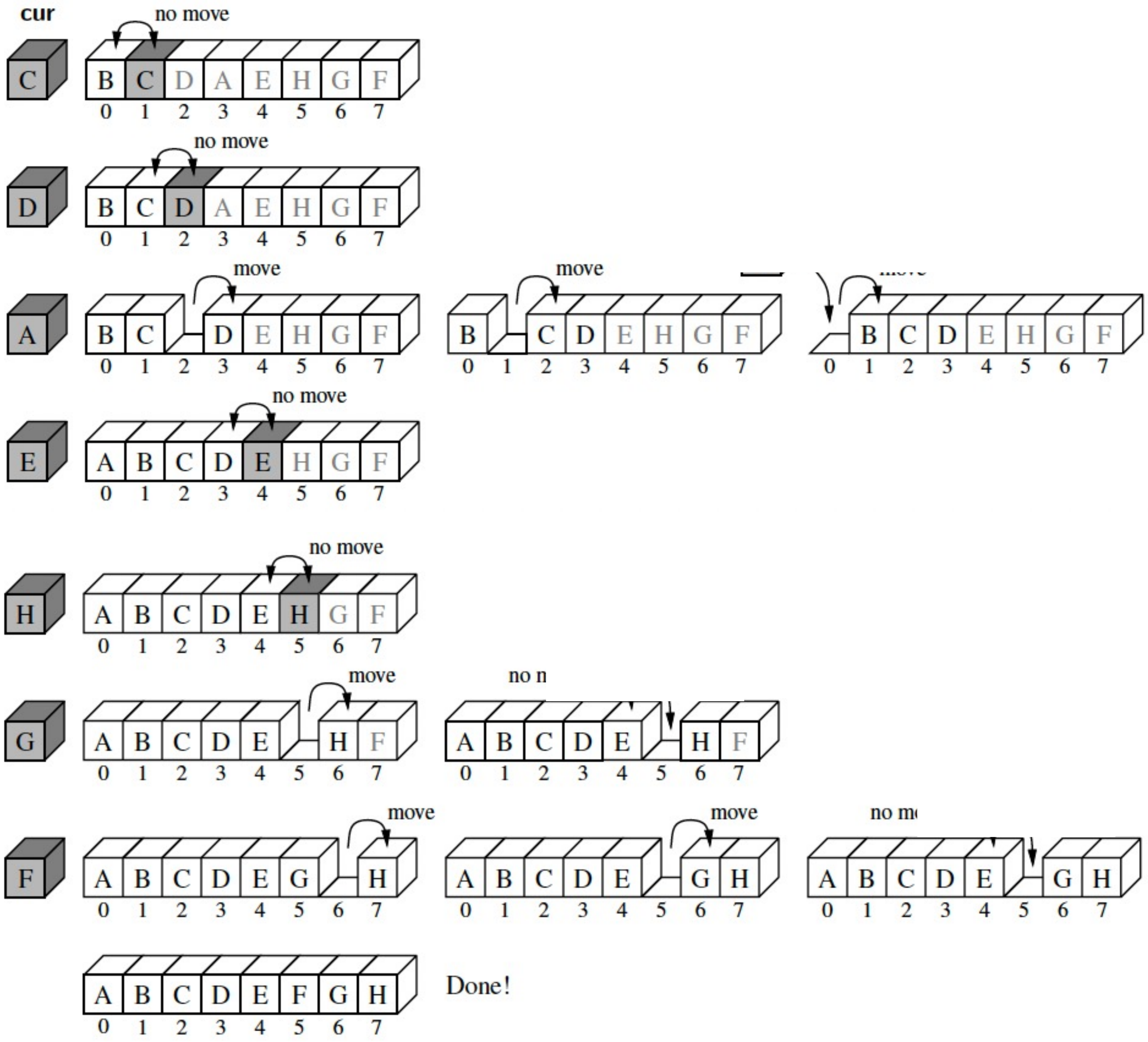
Using Array-based Sequences

- Sorting a sequence:



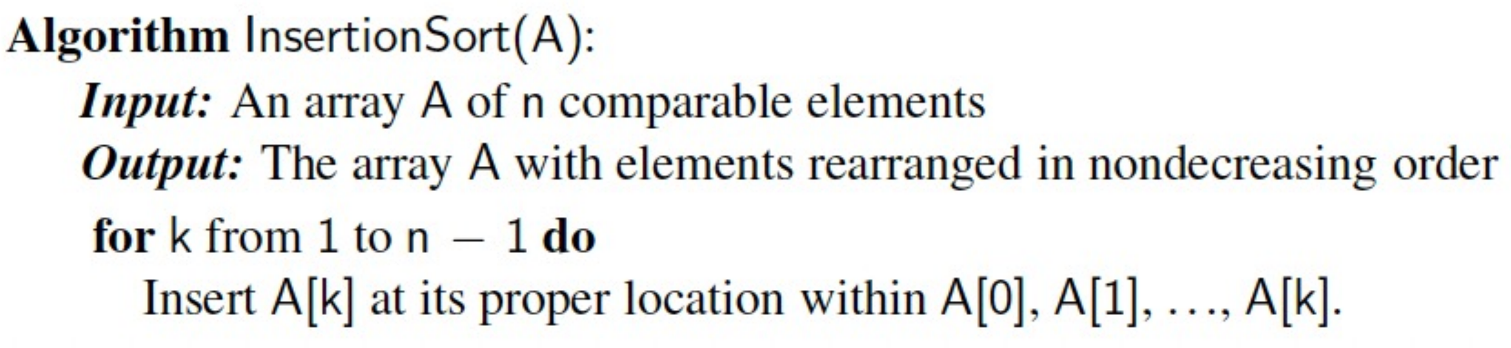
```
Algorithm InsertionSort(A):
  Input: An array A of n comparable elements
  Output: The array A with elements rearranged in nondecreasing order
  for k from 1 to n − 1 do
    Insert A[k] at its proper location within A[0], A[1], ..., A[k].
```

High-level description of the insertion-sort algorithm.


cur
no move
B C D A E H G F
0 1 2 3 4 5 6 7
C
no move
D
move
A
move
move
no move
E
no move
H
move
G
no n
move
F
move
no m
Done!

Using Array-based Sequences

- Sorting a sequence:
  - Exercise:
  - Give a Python implementation of the insertion-sort algorithm



```
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].
```

Using Array-based Sequences

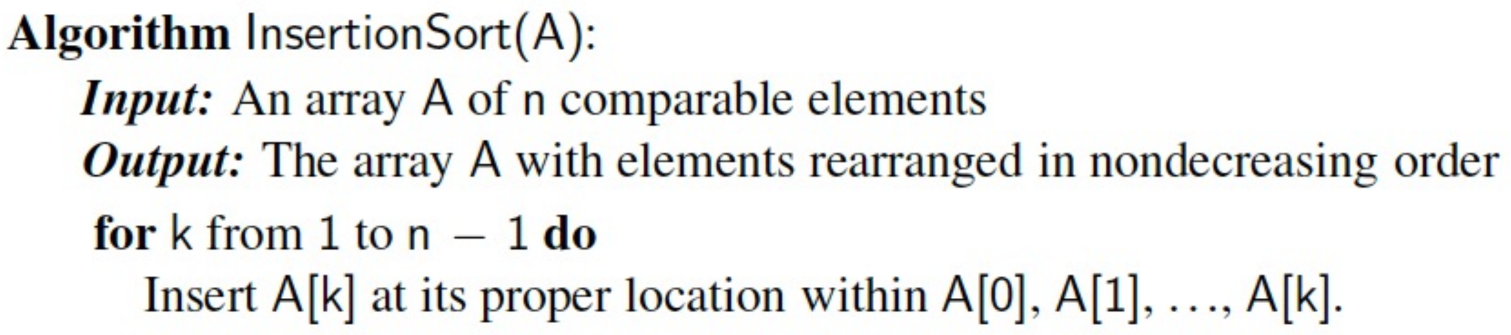
- Sorting a sequence:

  - Exercise:
  - Give a Python implementation of the insertion-sort algorithm

```
def insertion_sort(A):
      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 current
            while j > 0 and A[j−1] > cur: # element A[j−1] must be after current
                  A[j] = A[j−1]
                  j −= 1
            A[j] = cur                          # cur is now in the right place
```

Using Array-based Sequences

- Sorting a sequence:
    - The Nested loop of the insertion-sort lesad to $\boldsymbol{O(n^2)}$ running time in the worst case
    - **Worst case**: if the array is initially in reverse order
    - **Best case:** If the initial array is nearly sorted or perfectly sorted
        - Insertion-sort runs in $O(n)$ time because there are few or no iterations of the inner loop



```
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].
```

Using Array-based Sequences

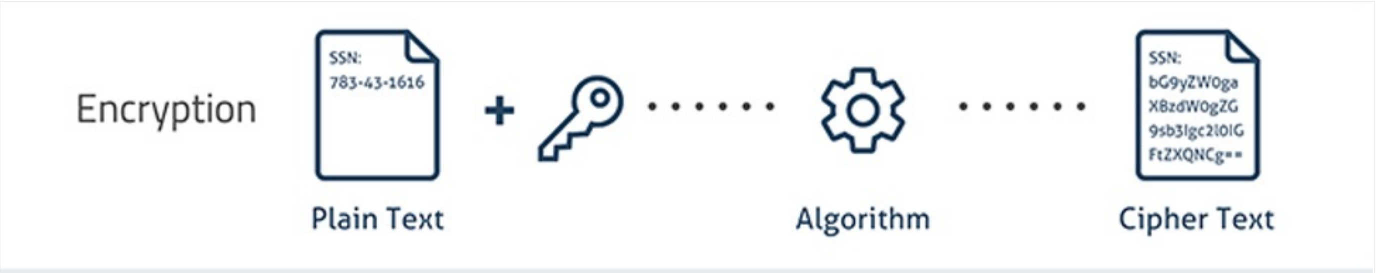
- We will study 3 use cases of array-based sequences:
    - Storing High Scores for a Game
    - Sorting a Sequence
    - Simple Cryptography

Using Array-based Sequences

- Simple Cryptography

- Cryptography is the practice and study of techniques for secure communication in the presence of adversarial behavior.

- More generally, cryptography is about constructing and analyzing protocols that prevent third parties or the public from reading private messages.



Src. : https://www..venafi.com

Using Array-based Sequences

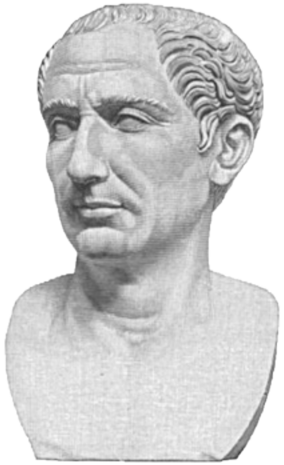
- Simple Cryptography

- **Caesar cipher** :

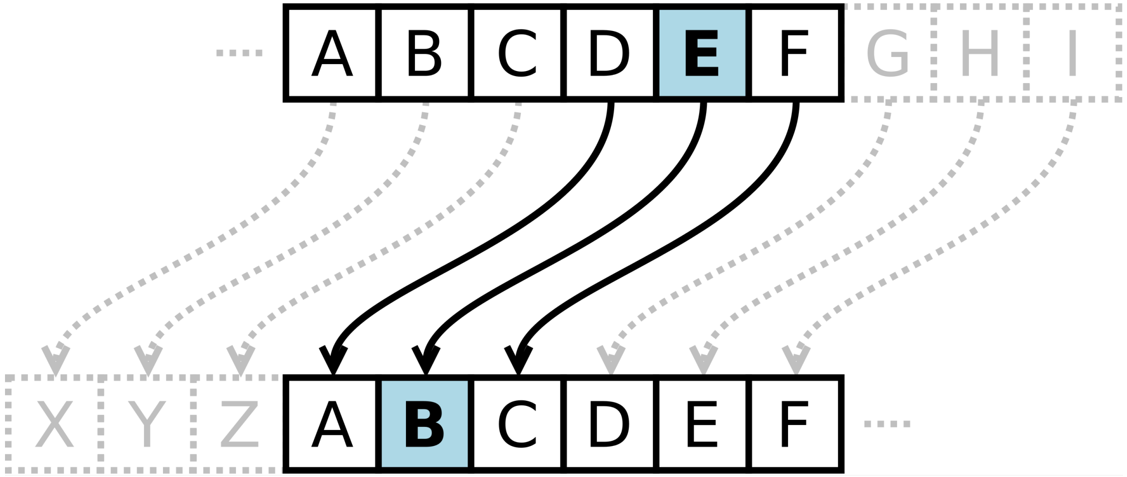
- Known as being the earliest encryption scheme.

- Used to protect important military messages.

- **Idea**: The Caesar cipher involves replacing each letter in a message with the letter that is a certain number of letters after it in the alphabet.



Juilius Caesar (100 BC – 44 BC)
a Roman general and statesman



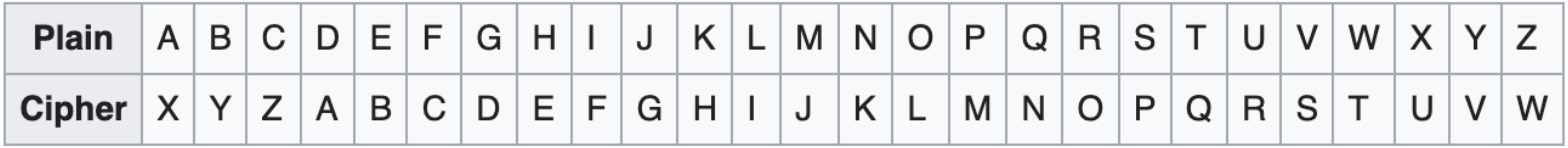
*Caesar used an alphabet where decrypting would shift **three letters to the left***

Using Array-based Sequences

- Simple Cryptography

- **Caesar cipher** :

using a left rotation of three places



Plain	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cipher	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

When encrypting, a person looks up each letter of the message in the "plain" line and writes down the corresponding letter in the "cipher" line.

Plaintext: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD

Using Array-based Sequences

- Simple Cryptography

- **Caesar cipher** :

The encryption can also be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, $A \rightarrow 0, B \rightarrow 1, \ldots, Z \rightarrow 25$

Encryption of a letter *x* by a shift *n* can be described mathematically as

$$E_n(x) = (x + n) \mod 26$$

Decryption is performed similarly,

$$D_n(x) = (x - n) \mod 26$$

Using Array-based Sequences

- Simple Cryptography

- **Caesar cipher** :

- **Frequency analysis:**
- Invented in the 9th century by Al-kindi
- Described in his book “A Manuscript on Deciphering Cryptographic Messages”, رسالة في استخراج الكتب المعماة.

- It examines the **letters frequency** in an encrypt message
- Al-Kindi used it to break the Cesar cryptosystem

أبو يوسف يعقوب بن إسحاق الصبّاح الكندي

Al-Kindi
(801 alkoufa – 872 Bagdad)

a prominent figure in
the House of Wisdom

Using Array-based Sequences

- Simple Cryptography

One way to solve an encrypted message, if we know its language, is to find a different plaintext of the same language long enough to fill one sheet or so, and then we count the occurrences of each letter. We call the most frequently occurring letter the "first", the next most occurring letter the "second", the following most occurring letter the "third", and so on, until we account for all the different letters in the plaintext sample.

Then we look at the cipher text we want to solve and we also classify its symbols. We find the most occurring symbol and change it to the form of the "first" letter of the plaintext sample, the next most common symbol is changed to the form of the "second" letter, and the following most common symbol is changed to the form of the "third" letter, and so on, until we acount for all symbols of the cryptogram we want to solve

أبو يوسف يعقوب بن إسحاق الصبّاح الكندي

Al-Kindi
(801 alkoufa – 872 Bagdad)

a prominent figure in
the House of Wisdom

Using Array-based Sequences

- Simple Cryptography

- **Caesar cipher** :

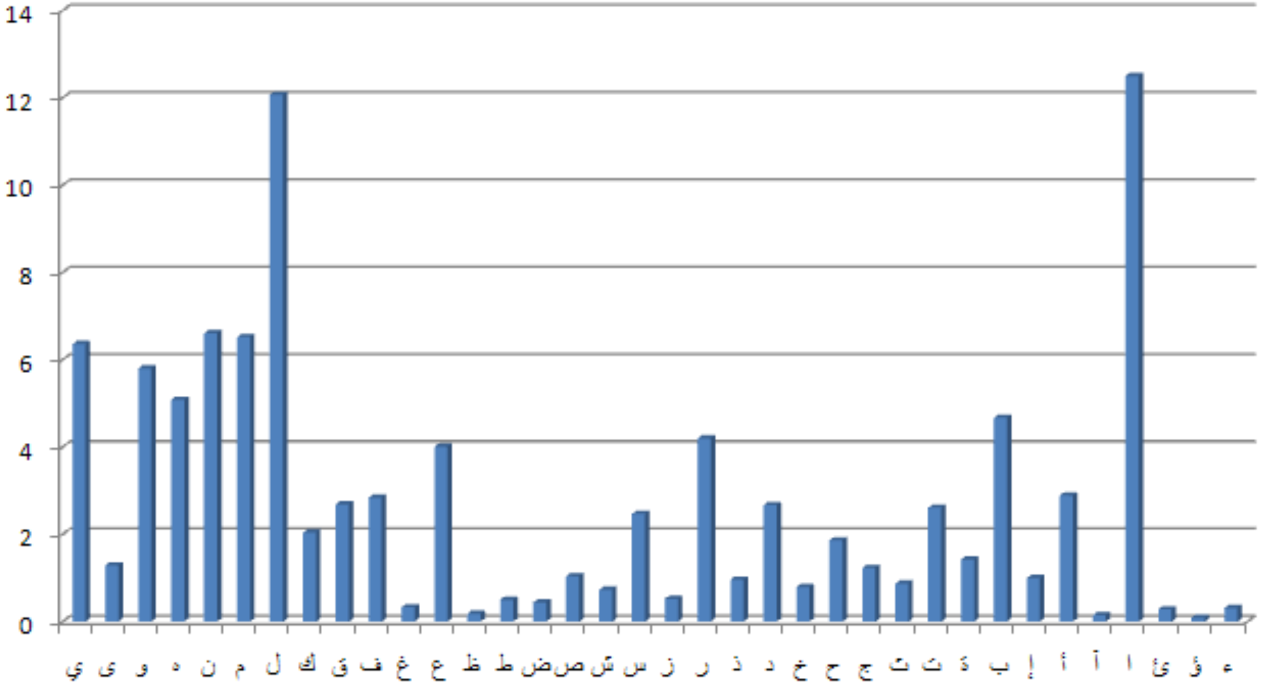
- **Frequency analysis:**
- It is based on the fact that, in any given stretch of written language, **certain letters and combinations of letters occur with varying frequencies**.
- Moreover, there is a characteristic distribution of letters that is roughly the same for almost all samples of that language.

- E.g. :
  - Given a section of English language, E, T, A and O are the most common letters, while Z, Q, X and J are rare.

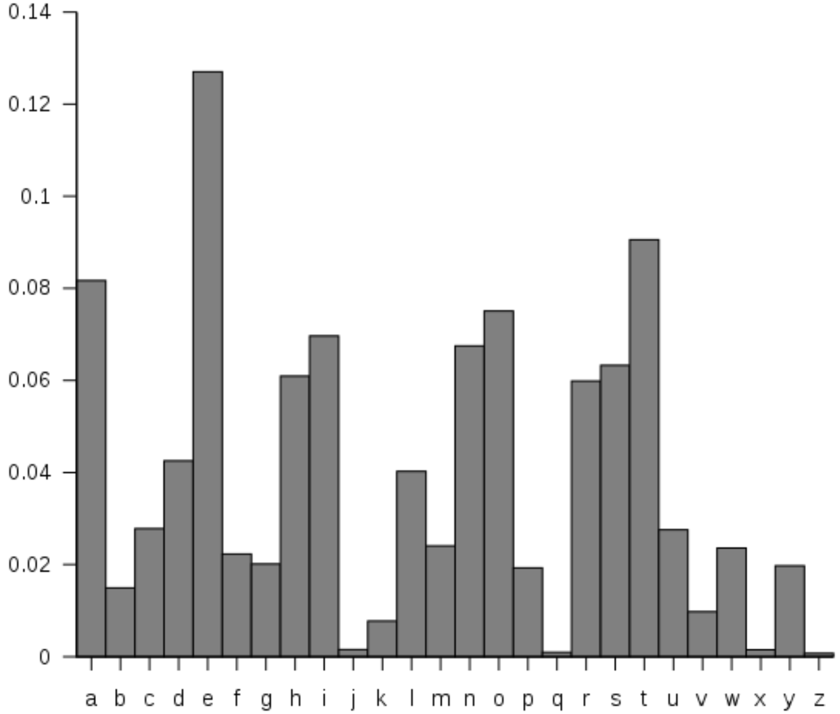
أبو يوسف يعقوب بن إسحاق الصبّاح الكندي

Al-Kindi
(801 alkoufa – 872 Bagdad)

a prominent figure in
the House of Wisdom



Arabic letter frequency

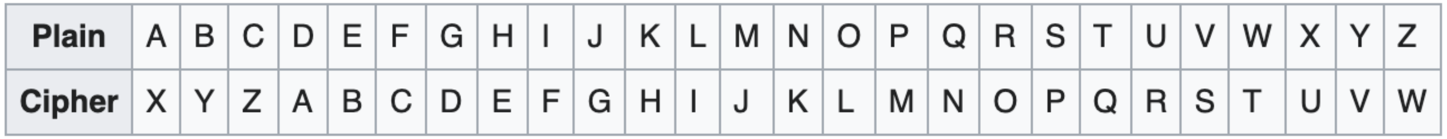


English letter frequency

Using Array-based Sequences

- Simple Cryptography

- **Caesar cipher** :
- **To do** : Implement the CaeserCipher class, such that:
    - It is characterized by the **shift** value
    - It maintains 2 precomputed strings (as the encryption and the decryption **keys**)
    - It has an **encrypt(message)** and a **decrypt(secret)** methods



Plain	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cipher	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

$$E_n(x) = (x + n) \mod 26 \qquad D_n(x) = (x - n) \mod 26$$

Using Array-based Sequences

CaesarCipher

_encoderKey
_decoderKey

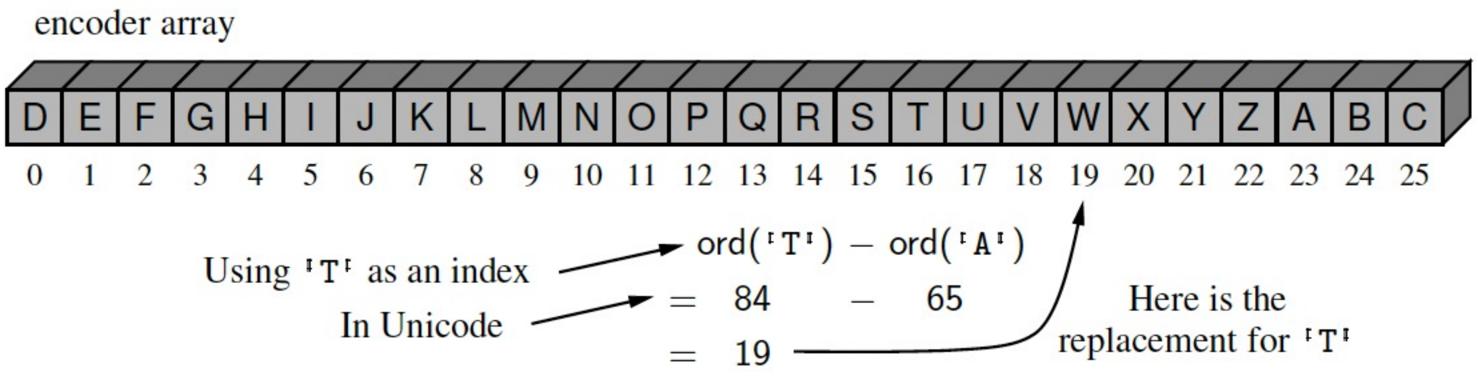
encrypt
decrypt

- Simple Cryptography

- **Caesar cipher** :
- **To do** : Implement the CaeserCipher class:
  - **Note:**
  - Python supports functions that convert between integer code point and one character strings.
  - Function **ord(c)** takes a one character string as parameter and returns the integer code point for that character
  - Function **chr(j)** takes an integer and returns its associated one-character string

To map the characters A to Z to the respective numbers 0 to 25.

j = ord(c) – ord('A ')



```
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._encoderKey = ‘’.join(encoder)     # will store as string
        self._decoderKey = ‘’.join(decoder)     # since fixed
```

```
def encrypt(self, message):
    ”””Return string representing encrypted message.”””
    return self._transform(message, self._encoderKey)

def decrypt(self, secret):
    ”””Return decrypted message given encrypted secret.”””
    return self._transform(secret, self._decoderKey)

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)
```

Multidimensional Data Sets

- Lists, tuples, and strings in Python are **one-dimensional**

- Many computer applications involve **multidimensional** data sets.

- For example:
    - Computer graphics are often modeled in either two or three dimensions.
    - Geographic information may be naturally represented in two dimensions.
    - Medical imaging may provide three-dimensional scans of a patient.

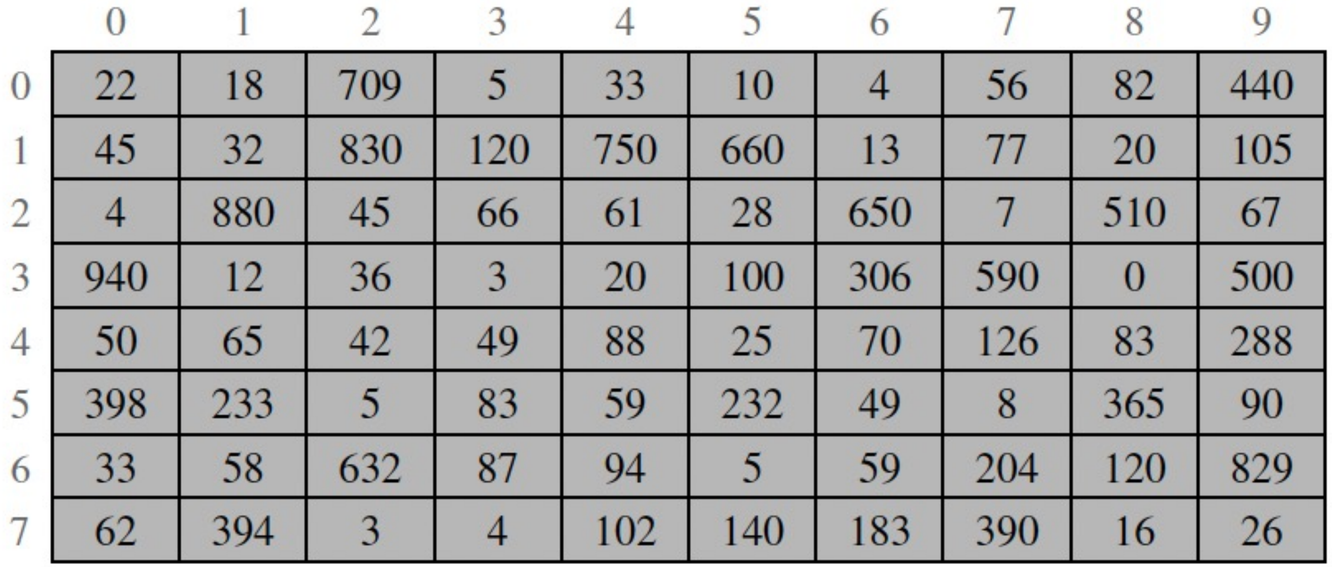
- A two-dimensional array is sometimes also called a **matrix**.

Multidimensional Data Sets

- A two-dimensional array is sometimes also called a **matrix**.

- We use two **indices**, say i and j, to refer to the **cells** in the matrix.
  - i refers to rows
  - j refers to columns

stores[3][5] is 100
Stores[6][2] is 632

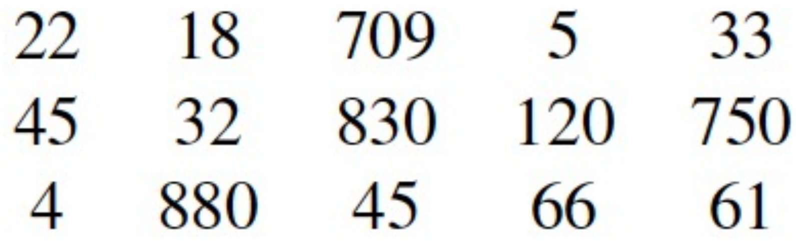


	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Multidimensional Data Sets

- A common representation for a two-dimensional data set in Python is as a list of lists.

- E.g.: data = [[22, 18, 709, 5, 33], [45, 32, 830, 120, 750], [4, 880, 45, 66, 61]]



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

Multidimensional Data Sets

- **Constructing a Multidimensional List**

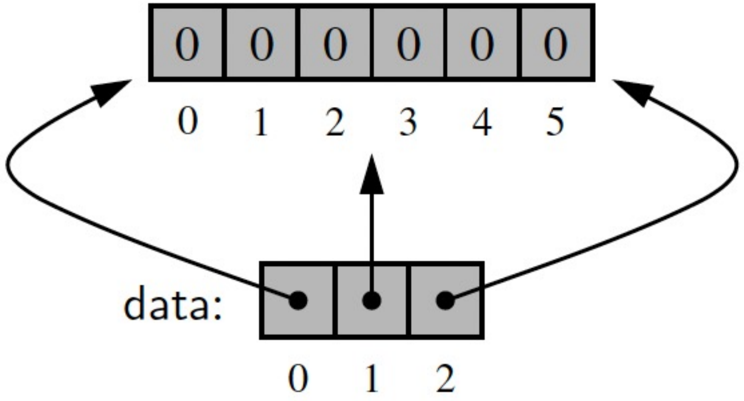
- For one-dimensional list, we rely generally on the syntax: $\boldsymbol{data = [0] * n}$
  (To create a list of n zeros)
  Remember that all entries references the same integer instance

- **What about list of lists?**

Multidimensional Data Sets

- **Constructing a Multidimensional List**

- $data = ([0] * c) * r$:
  - A mistake
  - $([0] * c)$: a list of c zeros
  - $([0] * c) * r$: a single list with length $r.c$
  - [2,4,6] * 2 results in list [2, 4, 6, 2, 4, 6]



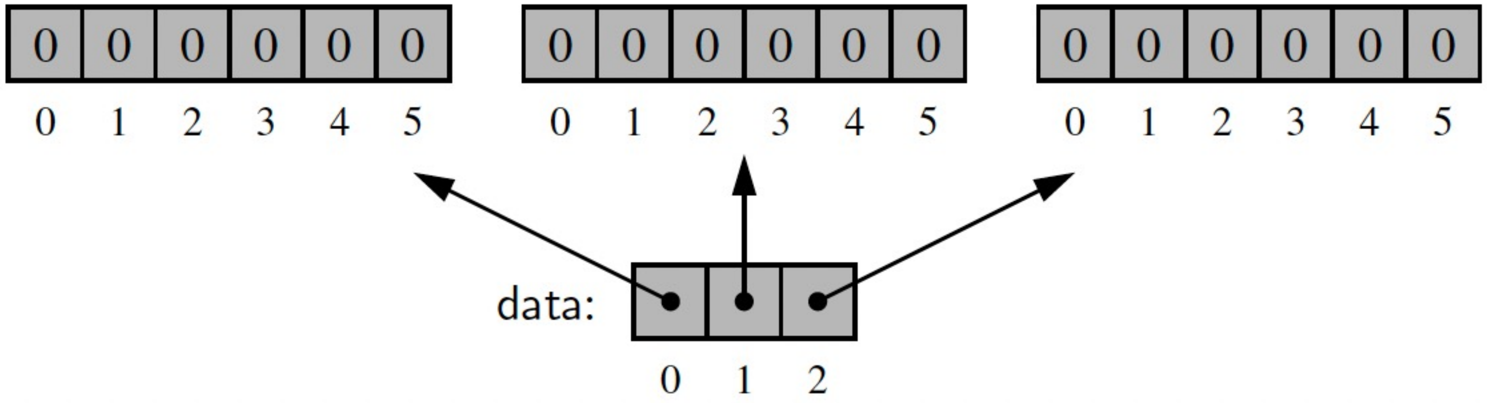
$data = [[0] * 6] * 3$

Updating data[2][0] updates also data[0][0] and data [1][0]

- $data = [[0] * c] * r$:
  - A mistake
  - It is formally a list of lists.
  - All r entries of the list known as data are references to the same instance of a list of c zeros.

Multidimensional Data Sets

- **Constructing a Multidimensional List**



A valid representation of a 3 x 6 data set as a list of lists.

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

r distinct secondary lists

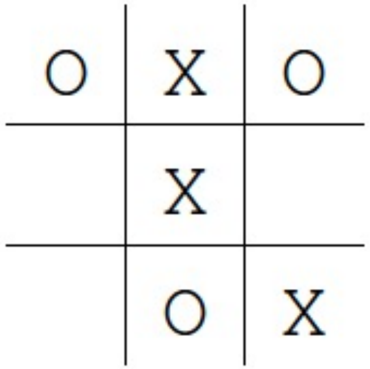
Multidimensional Data Sets

- **Positional Game (Tic - Tac - Toe):**

- Tic-Tac-Toe is a game played in a three-by-three board.

- Two players—X and O—alternate in placing their respective marks in the cells of this board, starting with player X.

- The winner: The one who succeeds in getting three of his or her marks in a row, column, or diagonal.



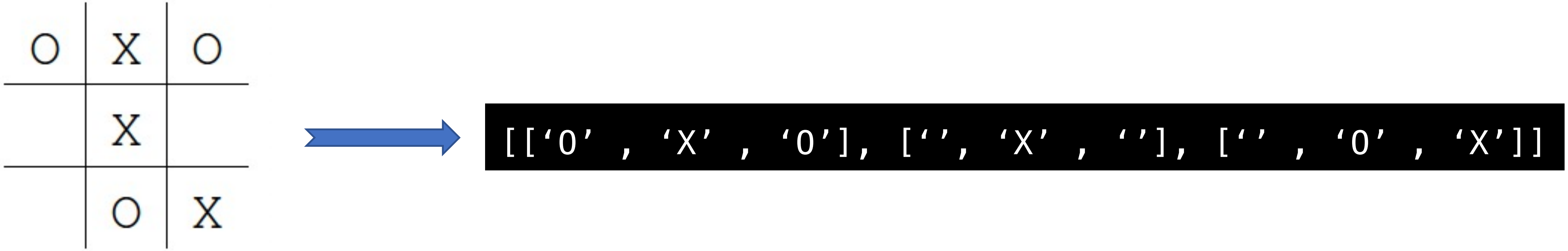
O	X	O
	X	
	O	X

Multidimensional Data Sets

- **Positional Game (Tic - Tac - Toe):**

- Our representation of a 3 x 3 board will be a list of lists of characters.

- ‘X’ or ‘O’ designating a player’s move, or ‘’ designating an empty space.



O	X	O
	X	
	O	X

```
[[‘O’ , ‘X’ , ‘O’], [‘’, ‘X’ , ‘’], [‘’ , ‘O’ , ‘X’]]
```

- **Aim:** Develop a class that will keep track of the moves of 2 players and report a winner. (*it does not allow playing against the computer*)

Multidimensional Data Sets

- **Positional Game (Tic - Tac - Toe):**

- **Aim:** Develop a class that will keep track of the moves of 2 players and report a winner. (it does not allow playing against the computer)

```
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’)
```

mark(i, j) method adds a mark at the given position for the current player

Multidimensional Data Sets

- **Positional Game (Tic - Tac - Toe):**
- **Aim:** Develop a class that will keep track of the moves of 2 players and report a winner. (it does not allow playing against the computer)

```
class TicTacToe:
      def __init__(self):
            ”””Start a new game.”””
            self._board = [[‘’] * 3 for j in range(3)]
            self._player = ‘X’

      def __str__ (self):
            rows = [‘|’ .join(self._board[r]) for r in range(3)]
            return ‘\n-----\n’.join(rows)
```

```
def mark(self, i, j):
        ”””Put an 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):
       board = self._board                     # local variable for shorthand
       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):
       for mark in ‘XO’ :
             if self._is_win(mark):
                    return mark                # Return mark of winning player
       return None
```