كلية
الحاسبات والمعلومات
كلية معتمدة من الهيئة القومية
لضمان جودة التعليم والاعتماد

Assiut University

Course Title: Data Structures and Algorithms
Course Code: CS211

Prof. Dr. Khaled F. Hussain

Reference

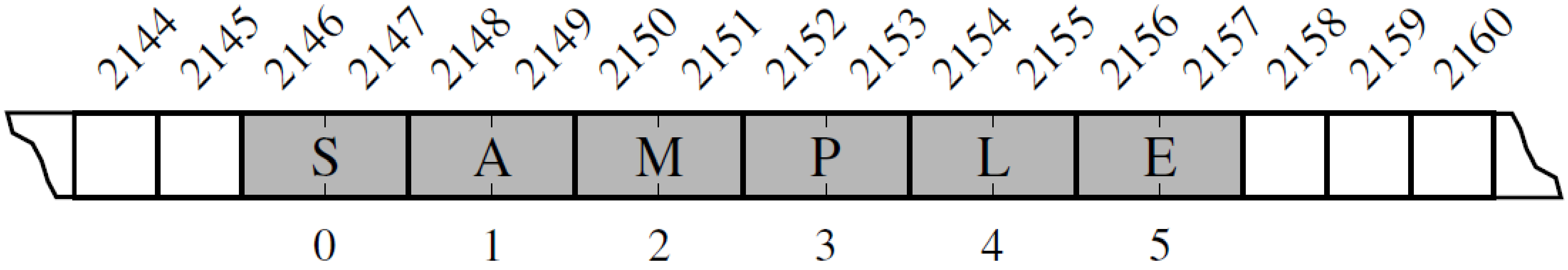
- Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, "Data Structures and Algorithms in Python"

Python’s Sequence Types

- We explore Python’s various “sequence” classes, namely the built-in `list`, `tuple`, and `str` classes.
- There is significant commonality between these classes: each supports indexing to access an individual element of a sequence, using a syntax such as `seq[k]`, and each uses a low-level concept known as an ***array*** to represent the sequence.

Python’s Sequence Types (Cont.)

- A group of related variables can be stored one after another in a contiguous portion of the computer’s memory. We will denote such a representation as an ***array***.
- As a tangible example, a text string is stored as an ordered sequence of individual characters.
- In Python, each character is represented using the Unicode character set, and on most computing systems, Python internally represents each Unicode character with 16 bits (i.e., 2 bytes).
- Therefore, a six-character string, such as ‘SAMPLE’ , would be stored in 12 consecutive bytes of memory

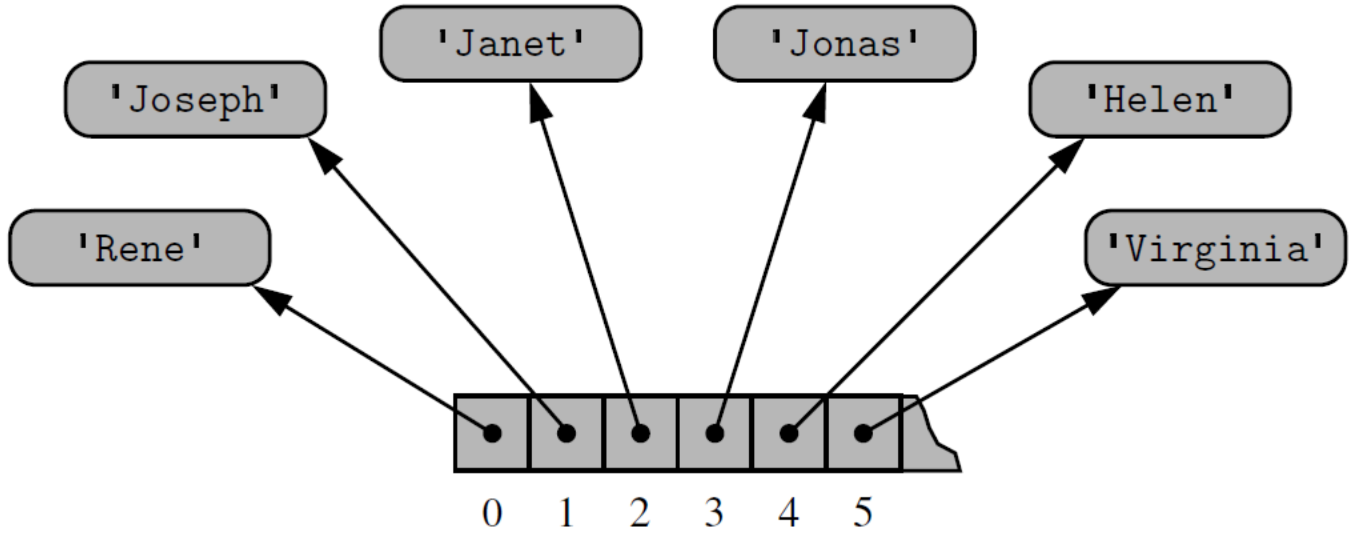


Python’s Sequence Types (Cont.)

- We will refer to each location within an array as a ***cell***, and will use an integer ***index*** to describe its location within the array, with cells numbered starting with 0, 1, 2, and so on. For example, the cell of the array with index 4 has contents L and is stored in bytes 2154 and 2155 of memory.
- The appropriate memory address can be computed using the calculation, start + cellsize*index.
- As an example, cell 4 begins at memory location $2146+2 \cdot 4 = 2146+8 = 2154$.

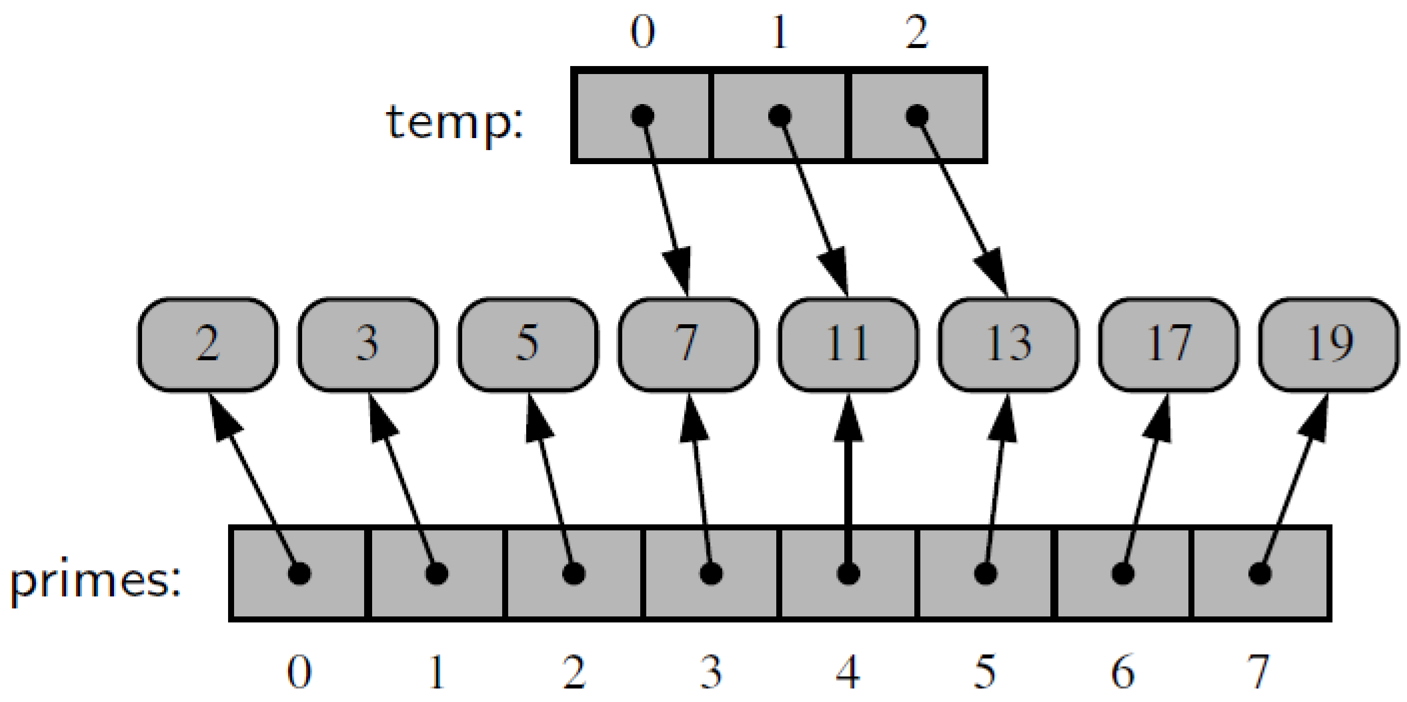
Python’s Sequence Types (Cont.)

- Python represents a list or tuple instance using an internal storage mechanism of an 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.
- Although the relative size of the individual elements may vary, the number of bits used to store the memory address of each element is fixed (e.g., 64-bits per address). In this way, Python can support constant-time access to a list or tuple element based on its index.



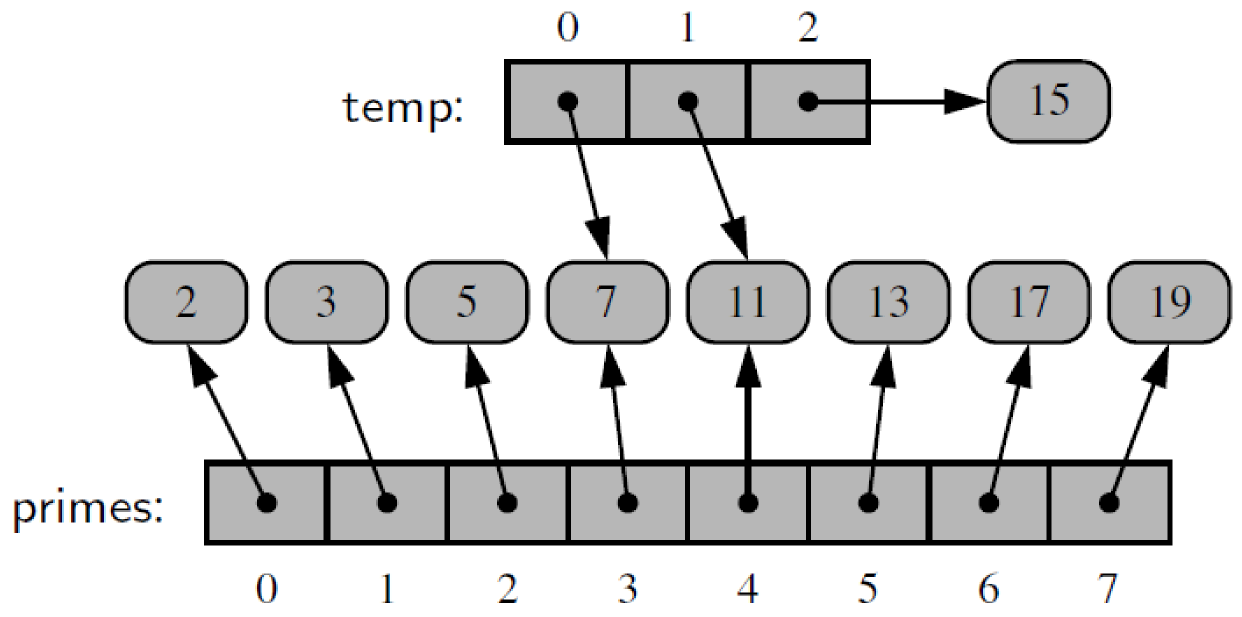
Python’s Sequence Types (Cont.)

- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
- As an example, when you compute a slice of a list `temp = primes[3:6]`, the result is a new list instance, but that new list has references to the same elements that are in the original list



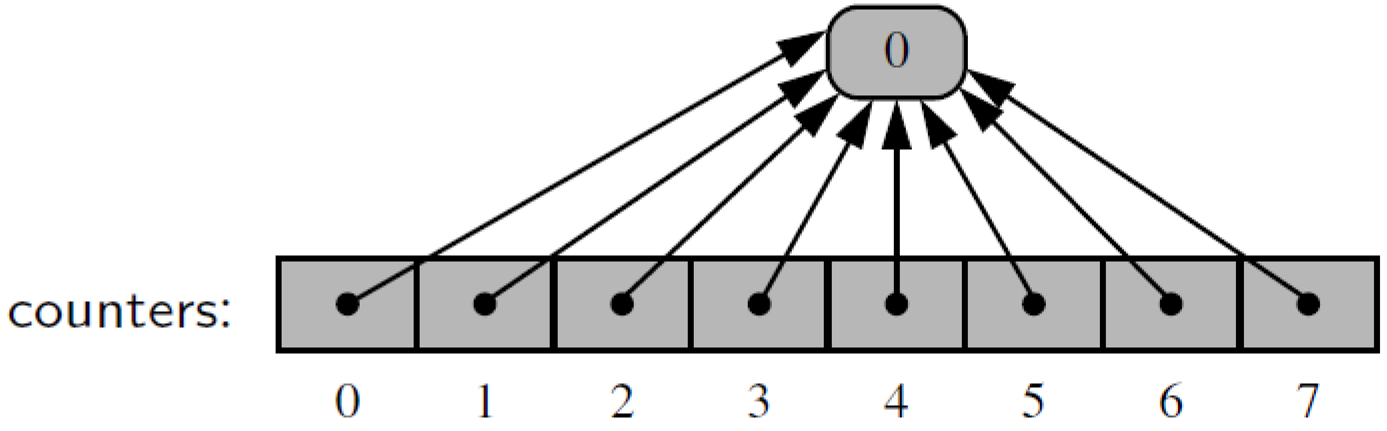
Python's Sequence Types (Cont.)

- When the elements of the list are immutable objects, as with the integer instances, for example, the command temp[2] = 15 were executed from this configuration, that does not change the existing integer object; it changes the reference in cell 2 of the temp list to reference a different object.



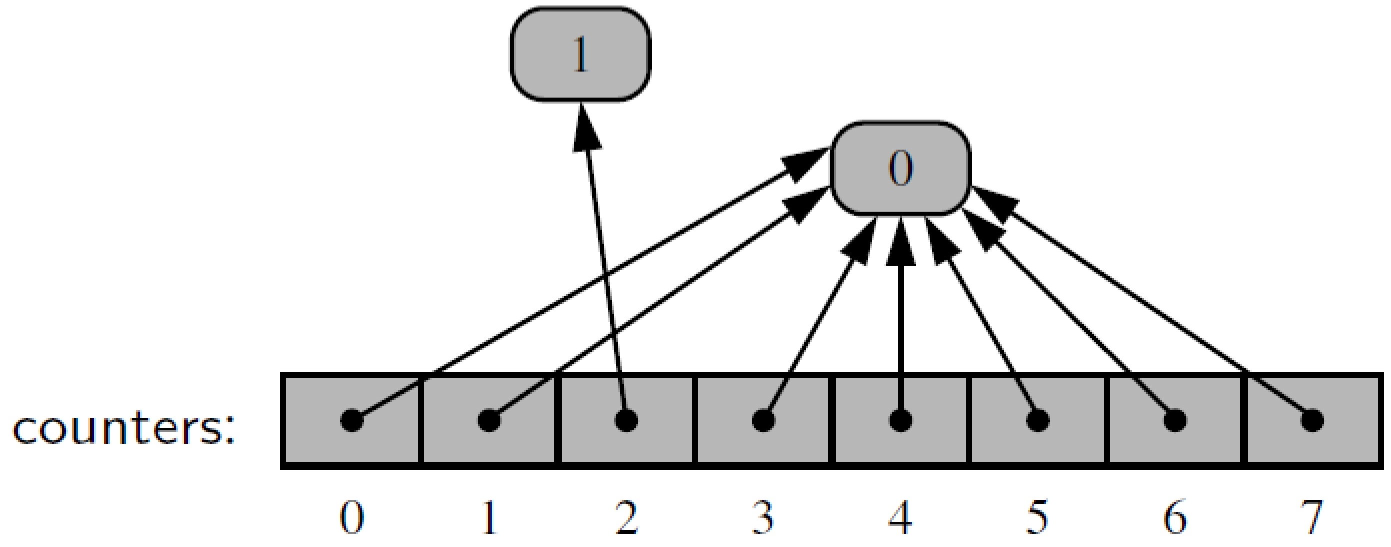
Python's Sequence Types (Cont.)

- It is a common practice in Python to initialize an array of integers using a syntax such as `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.



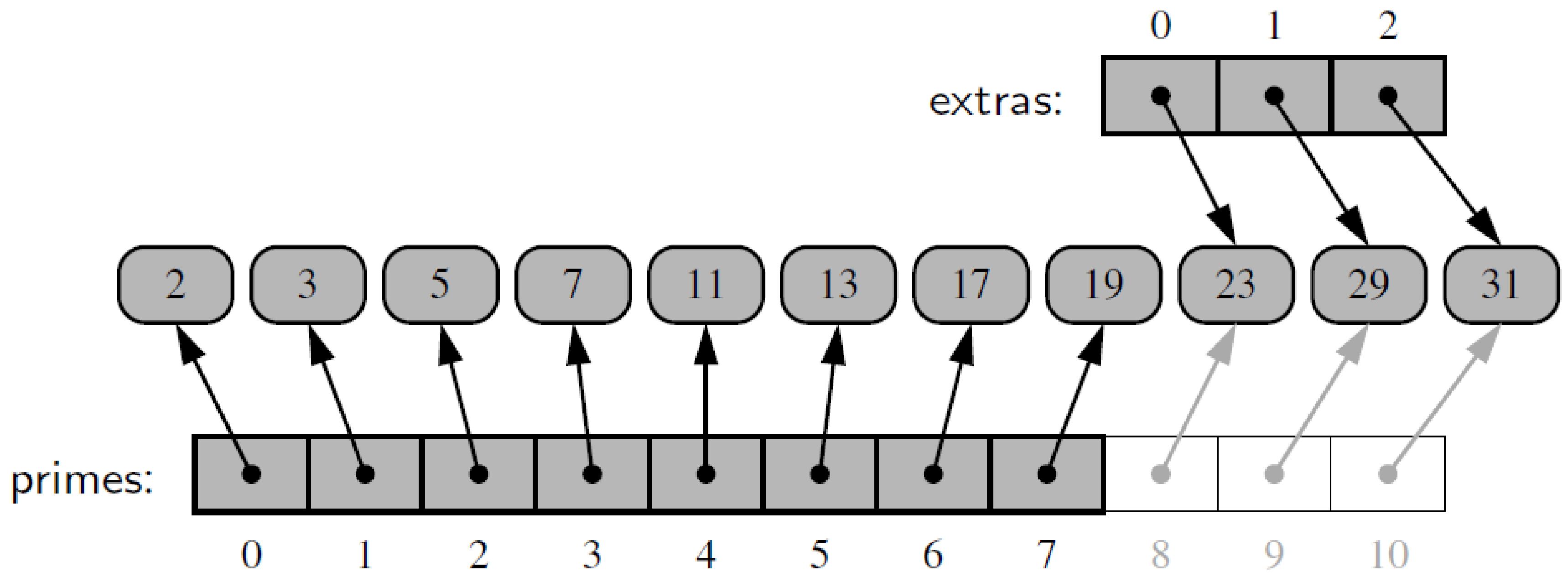
Python’s Sequence Types (Cont.)

- However, we rely on the fact that the referenced integer is immutable.
- Even a command such as `counters[2] += 1` does not technically change the value of the existing integer instance. This computes a new integer, with value 0+1, and sets cell 2 to reference the newly computed value.



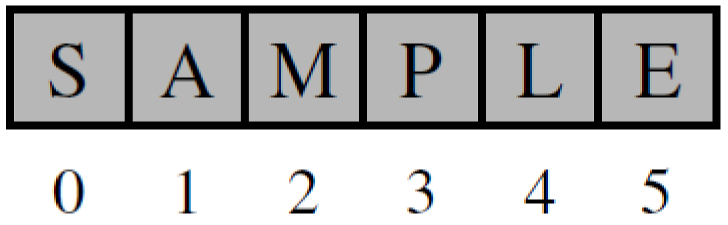
Python’s Sequence Types (Cont.)

- As a demonstration of the referential nature of lists, we note that the extend command is used to add all elements from one list to the end of another list. The extended list does not receive copies of those elements, it receives references to those elements.
- The effect of command primes.extend(extras)



Compact Arrays in Python

- strings are represented using an array of characters (not an array of references). We will refer to this more direct representation as a ***compact array.***
- Compact arrays have several advantages over referential structures in terms of computing performance. The overall memory usage will be much lower for a compact structure



Compact Arrays in Python (Cont.)

- Suppose we wish to store a sequence of one million, 64-bit integers. In theory, we might hope to use only 64 million bits. However, we estimate that a Python list will use *four to five times as much memory*.
- Each element of the list will result in a 64-bit memory address being stored in the primary array, and an `int` instance being stored elsewhere in memory.
- Python allows you to query the actual number of bytes being used for the primary storage of any object. This is done using the `getsizeof` function of the `sys` module.

Compact Arrays in Python (Cont.)

Python integers are objects, and they manage their own memory. They will use as many bytes as needed to hold the number you want.

```
import sys

x = 0

sys.getsizeof(x)
```

- 28 (bytes)

```
x = 100_000_000_000_000

sys.getsizeof(x)
```

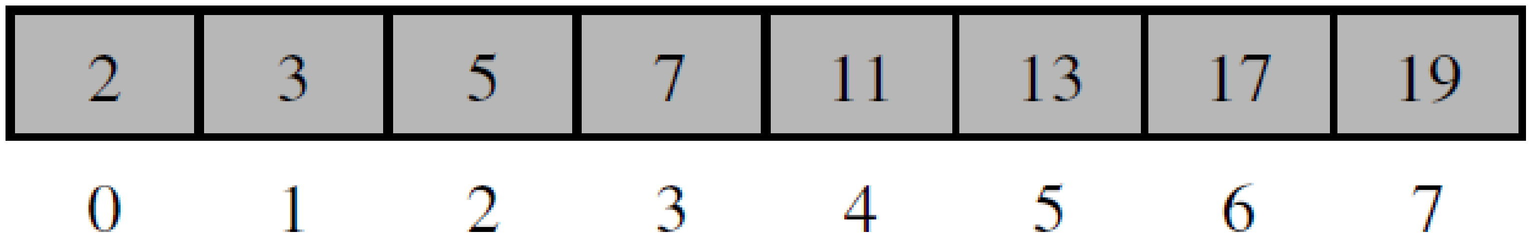
- 32 (bytes)

```
x = x ** 1000

sys.getsizeof(x)
```

- 6228 (bytes)

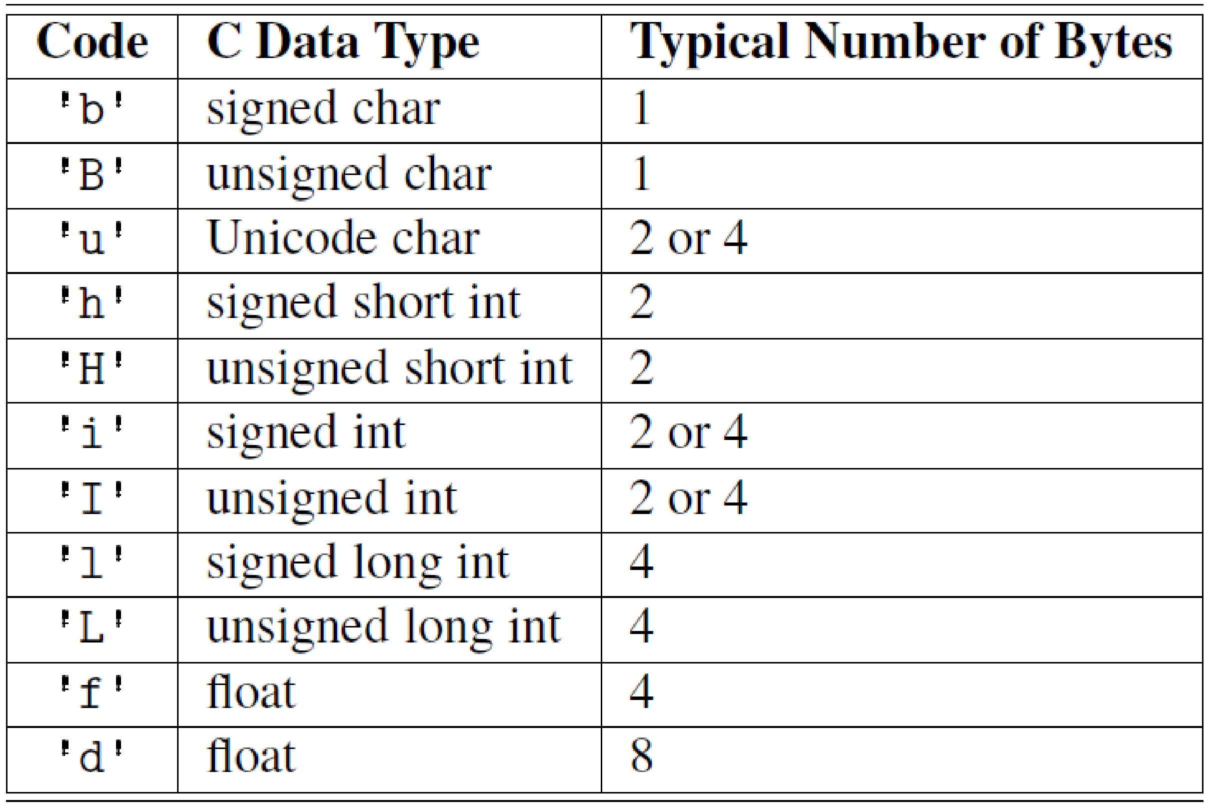
Compact Arrays in Python (Cont.)

- Primary support for compact arrays is in a module named array.
- The public interface for the array class conforms mostly to that of a Python list.
- However, the constructor for the array class requires a ***type code*** as a first parameter, which is a character that designates the type of data that will be stored in the array.
- As a tangible example, the type code, i , designates an array of (signed) integers.
- primes = array(i , [2, 3, 5, 7, 11, 13, 17, 19])



Compact Arrays in Python (Cont.)

- Type codes supported by the array module.



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

Dynamic Arrays

- Python’s list class presents a more interesting abstraction. Although a list has a particular length when constructed, the class allows us to add elements to the list, with no apparent limit on the overall capacity of the list.
- To provide this abstraction, Python relies on an algorithm known as a ***dynamic array***.
- The first key to providing the semantics of a dynamic array is that a list instance maintains an underlying array that often has greater capacity than the current length of the list.
- If a user continues to append elements to a list, any reserved capacity will eventually be exhausted. In that case, the class requests a new, larger array from the system, and initializes the new array so that its prefix matches that of the existing smaller array. At that point in time, the old array is no longer needed.

Dynamic Arrays (Cont.)

```python
import sys        # provides getsizeof function
data = [ ]
n = 100
for k in range(n):
    a = len(data)          # number of elements
    b = sys.getsizeof(data)      # actual size in bytes
    print("Length: {0:3d}; Size in bytes: {1:4d}".format(a, b))
    data.append(None)
```

Dynamic Arrays (Cont.)

```
Length: 0; Size in bytes : 72
Length: 1; Size in bytes : 104
Length: 2; Size in bytes : 104
Length: 3; Size in bytes : 104
Length: 4; Size in bytes : 104
Length: 5; Size in bytes : 136
Length: 6; Size in bytes : 136
Length: 7; Size in bytes : 136
Length: 8; Size in bytes : 136
Length: 9; Size in bytes : 200
Length: 10; Size in bytes : 200
…
….
```

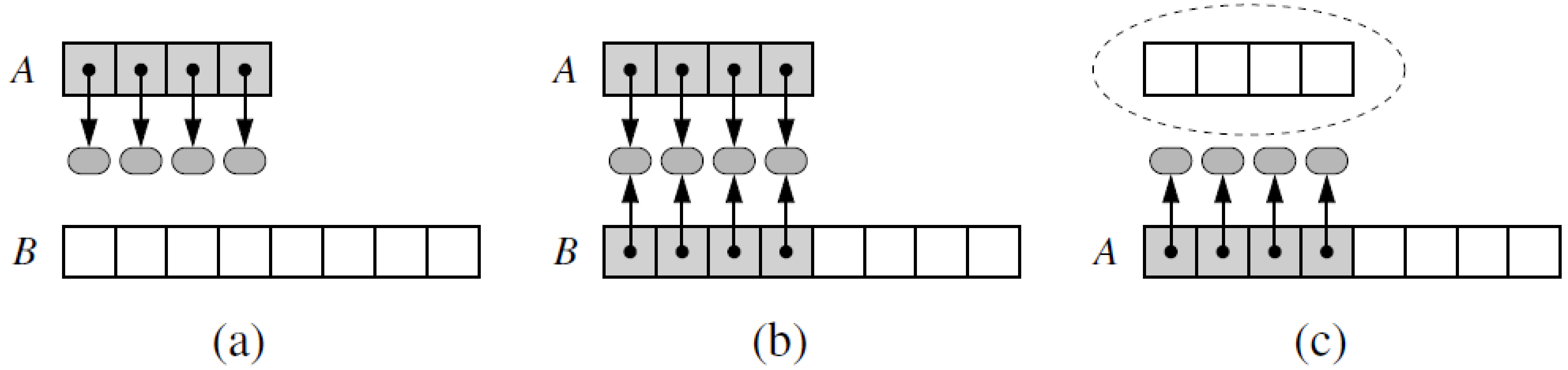
Dynamic Arrays (Cont.)

- Because a list is a referential structure, the result of getsizeof for a list instance only includes the size for representing its primary structure; it does not account for memory used by the objects that are elements of the list.
- We repeatedly append None to the list, because we do not care about the contents, but we could append any type of object without affecting the number of bytes reported by getsizeof(data).

Implementing a Dynamic Array

1. Allocate a new array *B* with larger capacity.
2. Set *B*[*i*] = *A*[*i*], for *i* = 0, . . . ,*n*−1, where *n* denotes current number of items.
3. Set *A* = *B*, that is, we henceforth use *B* as the array supporting the list.
4. Insert the new element in the new array.

- The remaining issue to consider is how large of a new array to create. A commonly used rule is for the new array to have twice the capacity of the existing array that has been filled.



```
import ctypes                                          # provides low-level arrays

 class DynamicArray:

   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 __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]                         # retrieve from 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
```

```
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


  def _make_array(self, c):                    # nonpublic utitity
    """Return new array with capacity c."""
    return (c * ctypes.py_object)()            # see ctypes documentation
```

```
def insert(self, k, value):
    """Insert value at index k, shifting subsequent values rightward."""
    # (for simplicity, we assume 0 <= k <= n in this verion)
    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
```

```
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 ValueError('value not found')   # only reached if no match
```

Composing Strings

- Assume that we have a large string named document, and the goal is to produce a new string, that contains only the alphabetic characters of the original string (e.g., with spaces, numbers, and punctuation removed).
- It may be tempting to compose a result through repeated concatenation, as follows:

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

- While the preceding code fragment accomplishes the goal, it is inefficient, because strings are immutable.
- Constructing that new string would require time proportional to its length. If the final result has n characters, the series of concatenations would take time proportional to the familiar sum $1+2+3+\cdots+n$, and therefore $O(n^2)$ time.

Composing Strings (Cont.)

- To guarantee linear time composition of a string is to use a temporary list to store individual pieces, and then to rely on the join method of the str class to compose the final result.

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

- This approach is guaranteed to run in $O(n)$ time.

Composing Strings (Cont.)

- We can further improve the practical execution time by using a list comprehension syntax to build up the temporary list, rather than by repeated calls to append.
- That solution appears as,

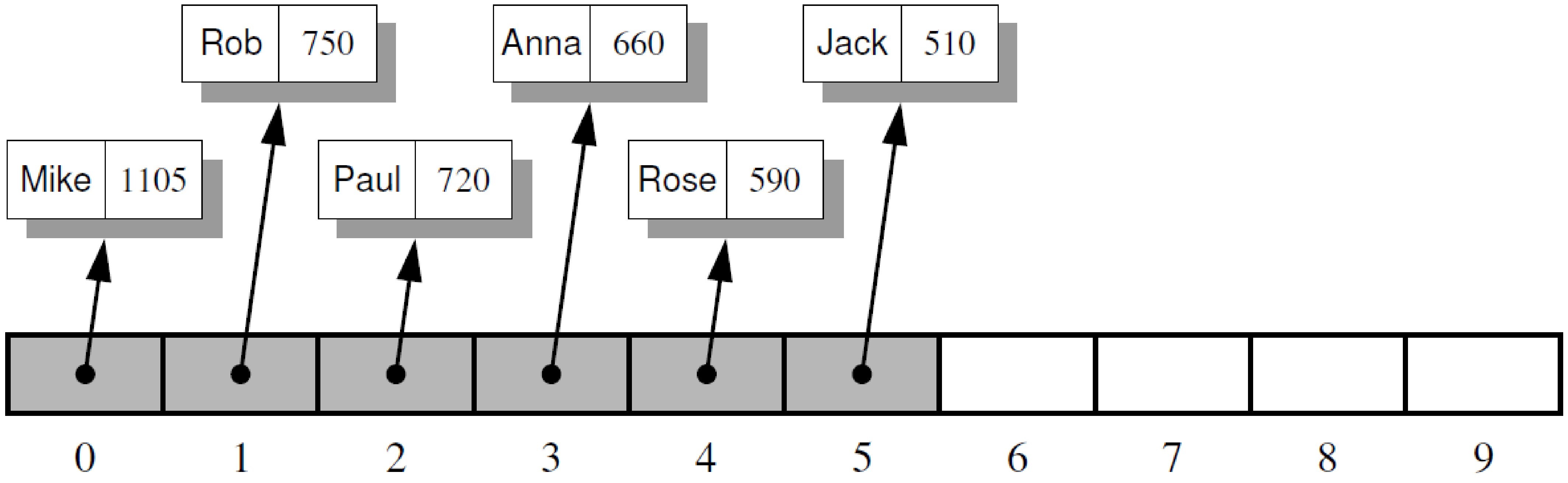
```
letters = ‘ ‘.join([c for c in document if c.isalpha( )])
```

- Better yet, we can entirely avoid the temporary list with a generator comprehension:

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

Storing High Scores for a Game

- The first application we study is storing a sequence of high score entries for a video game. This is representative of many applications in which a sequence of objects must be stored.
- The class supports the __getitem__ method to retrieve an entry at a given index with a syntax board[i] (or None if no such entry exists), and it supports a simple __str__ method that returns a string representation of the entire scoreboard, with one entry per line.



```
class GameEntry:
  """Represents one entry of a list of high scores."""

  def __init__(self, name, score):
    """Create an entry with given name and score."""
    self._name = name
    self._score = score

  def get_name(self):
    """Return the name of the person for this entry."""
    return self._name

  def get_score(self):
    """Return the score of this entry."""
    return self._score

 def __str__(self):
    """Return string representation of the entry."""
    return '({0}, {1})'.format(self._name, self._score) # e.g., '(Bob, 98)'
```

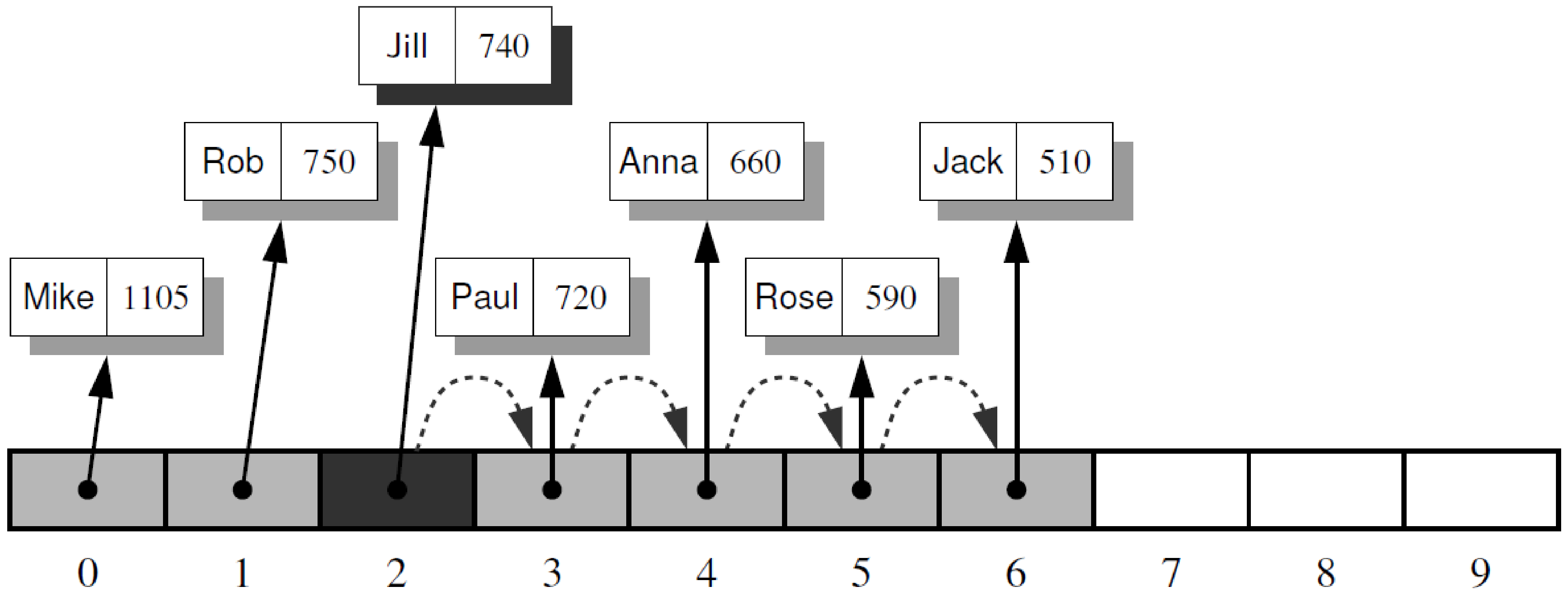
```
class Scoreboard:
  """Fixed-length sequence of high scores in nondecreasing order."""

  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):
   """Consider adding entry 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 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
```

```
if __name__ == '__main__':
  board = Scoreboard(5)
  for e in (
    ('Rob', 750), ('Mike',1105), ('Rose', 590), ('Jill', 740),
    ('Jack', 510), ('Anna', 660), ('Paul', 720), ('Bob', 400),
    ):
    ge = GameEntry(e[0], e[1])
    board.add(ge)
    print('After considering {0}, scoreboard is:'.format(ge))
    print(board)
    print()
```

Adding a new GameEntry

- Add Jill to the scoreboard. In order to make room for the new reference, we have to shift the references for game entries with smaller scores than the new one to the right by one cell. Then we can insert the new entry with index 2.



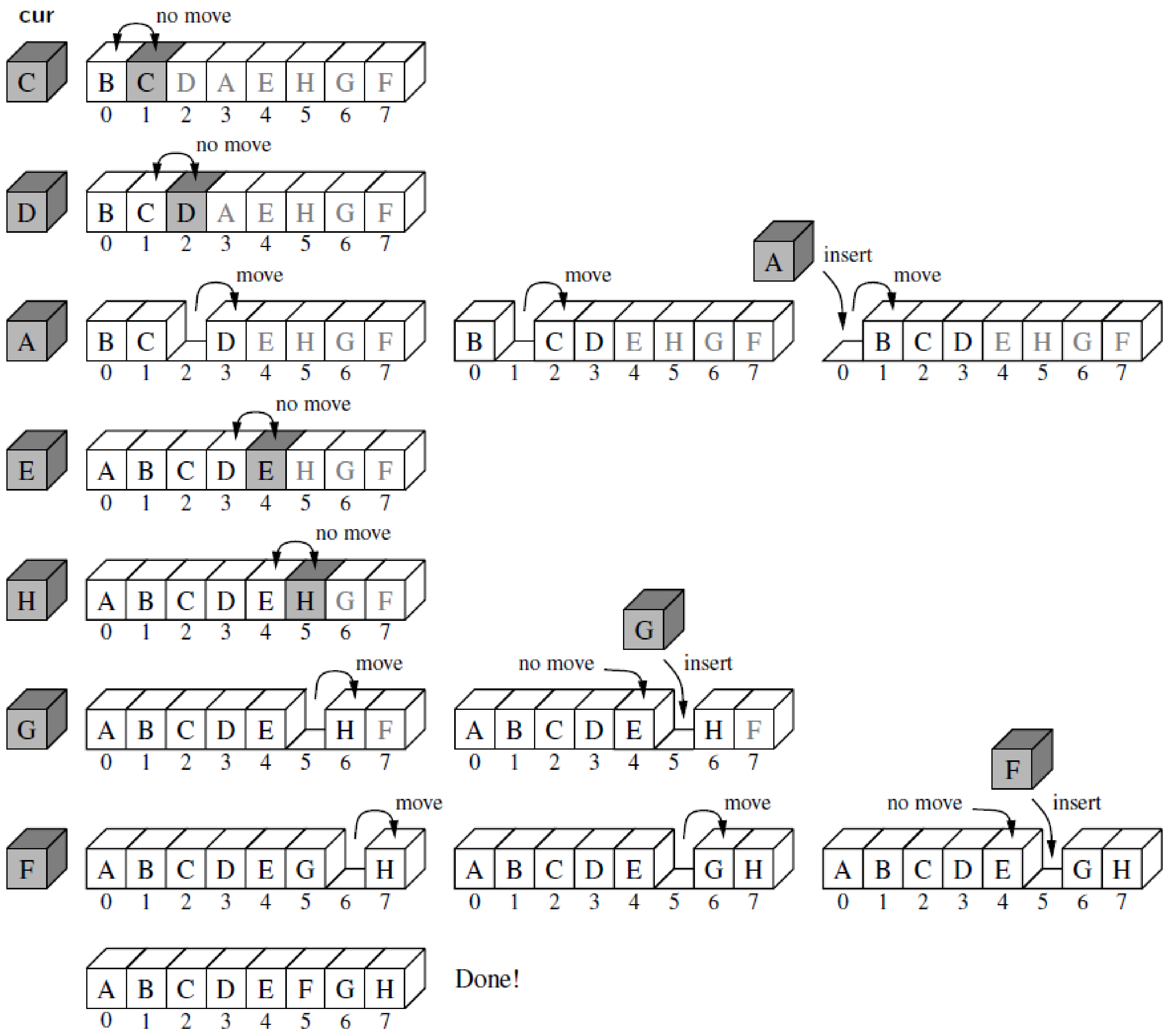
The Insertion-Sort Algorithm

```
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].
```

- The nested loops of insertion-sort lead to an $O(n^2)$ running time in the worst case. The most work is done if the array is initially in reverse order.
- On the other hand, 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.

The Insertion-Sort Algorithm (Cont.)

```
def insertion_sort(A):
  """Sort 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 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
```


cur
no move
move
insert
Done!

Multidimensional Data Sets

- A common representation for a two-dimensional data set in Python is as a list of lists.
- In particular, we can represent a two-dimensional array as a list of rows, with each row itself being a list of values.
- For example, the two-dimensional data

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

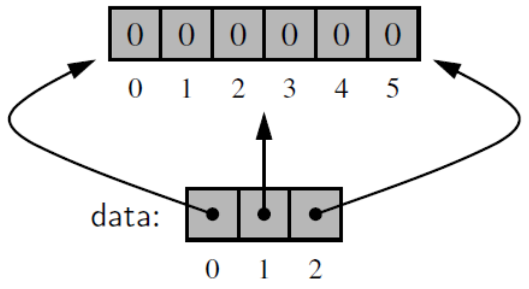
- might be stored in Python as follows.

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

- An advantage of this representation is that we can naturally use a syntax such as `data[1][3]` to represent the value that has row index 1 and column index 3

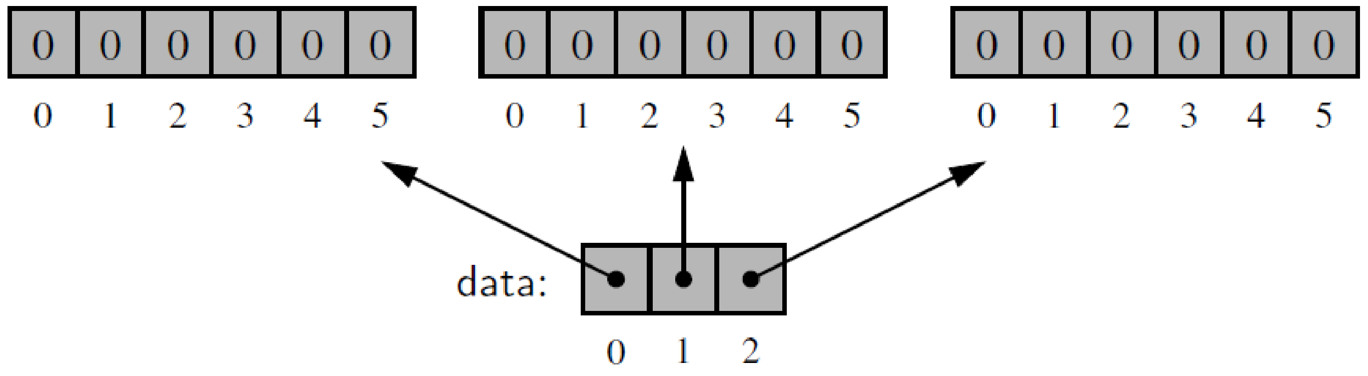
Multidimensional Data Sets (Cont.)

- To quickly initialize a one-dimensional list, we generally rely on a syntax such as `data = [0]*n` to create a list of *n* zeros.
- `data = ([0]*c)*r                # Warning: this is a mistake`
- While`([0]*c)` is indeed a list of *c* zeros, multiplying that list by *r* unfortunately creates a single list with length $r \cdot c$, just as `[2,4,6]*2` results in list `[2, 4, 6, 2, 4, 6]`.
- A better, yet still flawed attempt is to make a list that contains the list of *c* zeros as its only element, and then to multiply that list by *r*.
- `data = [ [0]*c ] *r             # Warning: still a mistake`
- This is much closer, as we actually do have a structure that is formally a list of lists.
- 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.



Multidimensional Data Sets (Cont.)

- This is truly a problem. Setting an entry such as `data[2][0] = 100` would change the first entry of the secondary list to reference a new value, 100. Yet that cell of the secondary list also represents the value `data[0][0]`, because "row" `data[0]` and "row" `data[2]` refer to the same secondary list.
- To properly initialize a two-dimensional list, we must ensure that each cell of the primary list refers to an *independent* instance of a secondary list. This can be accomplished through the use of Python's list comprehension syntax.
- `data = [ [0] * c for j in range(r) ]`
- This command produces a valid configuration. By using list comprehension, the expression `[0]*c` is reevaluated for each pass of the embedded for loop. Therefore, we get *r* distinct secondary lists, as desired.


0 0 0 0 0 0
0 1 2 3 4 5
0 0 0 0 0 0
0 1 2 3 4 5
0 0 0 0 0 0
0 1 2 3 4 5
data:
0 1 2