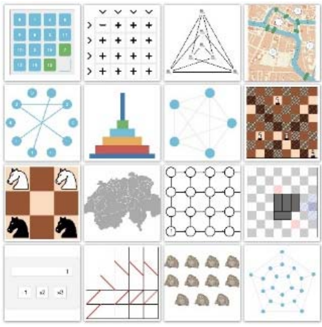
# Data Structures

## Lec 3: Array-Based Sequences



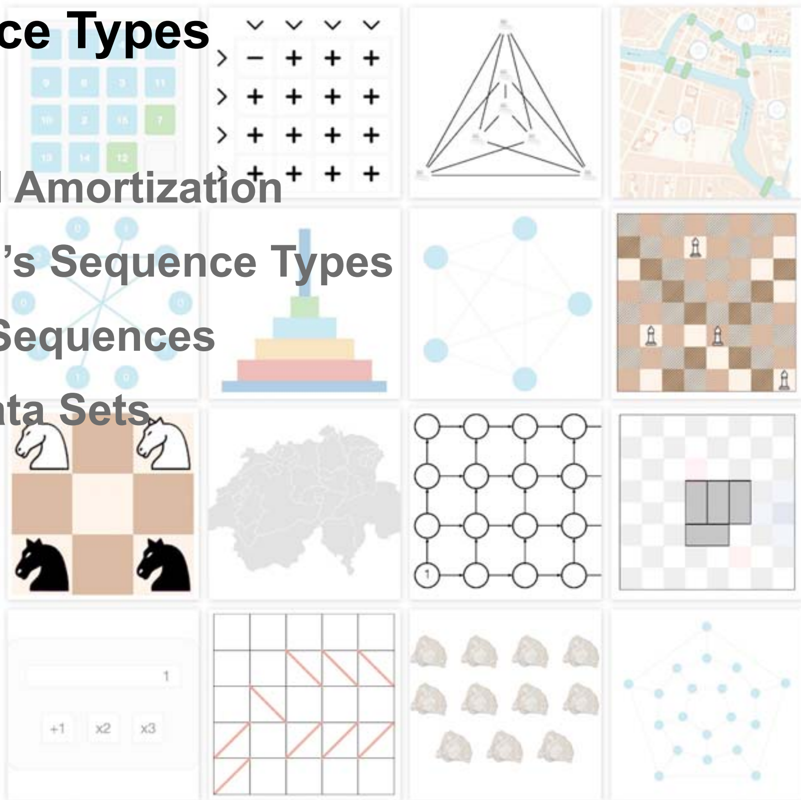

**Halûk Gümüşkaya**

Professor of Computer Engineering

**web:** http://www.gumuskaya.com

**e-mail:** haluk@gumuskaya.com, halukgumuskaya@atlas.edu.tr

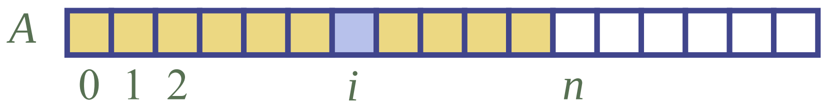

# Array-Based Sequences



1. **Python's Sequence Types**
2. Low-Level Arrays
3. Dynamic Arrays and Amortization
4. Efficiency of Python's Sequence Types
5. Using Array Based Sequences
6. Multidimensional Data Sets

# Python Sequence Classes

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



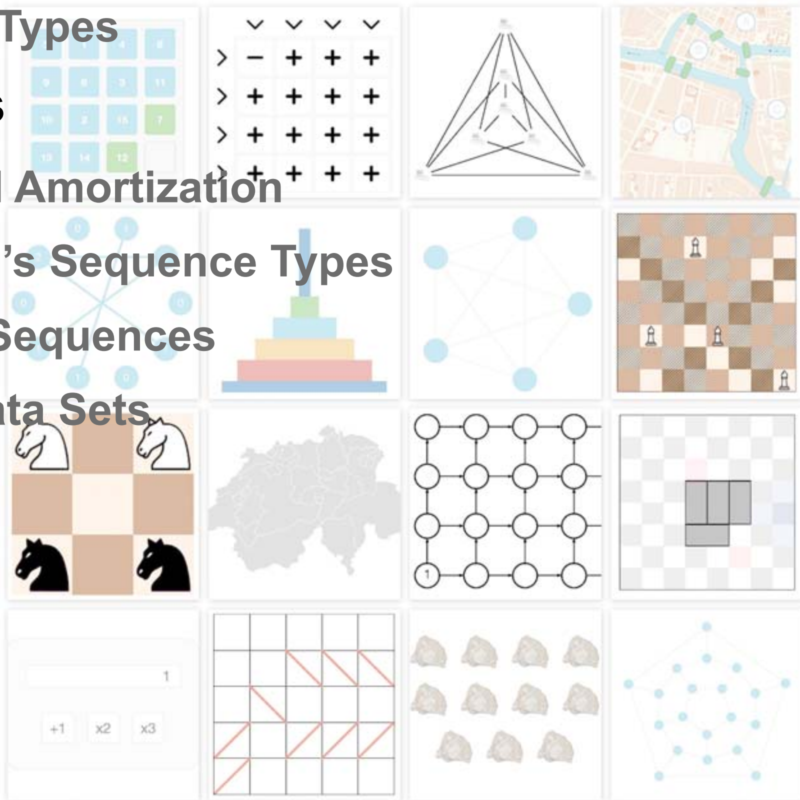



# Python Sequence Classes: `list`, `tuple`, and `str`

- There are significant differences:
  - in the abstractions that these classes represent, and
  - in the way that instances of these classes are represented internally by Python.
- Used so widely in Python programs, and they will become building blocks upon which we will develop more complex data structures,
- It is imperative that we establish a clear understanding of both the public behavior and inner workings of these classes.

# Array-Based Sequences

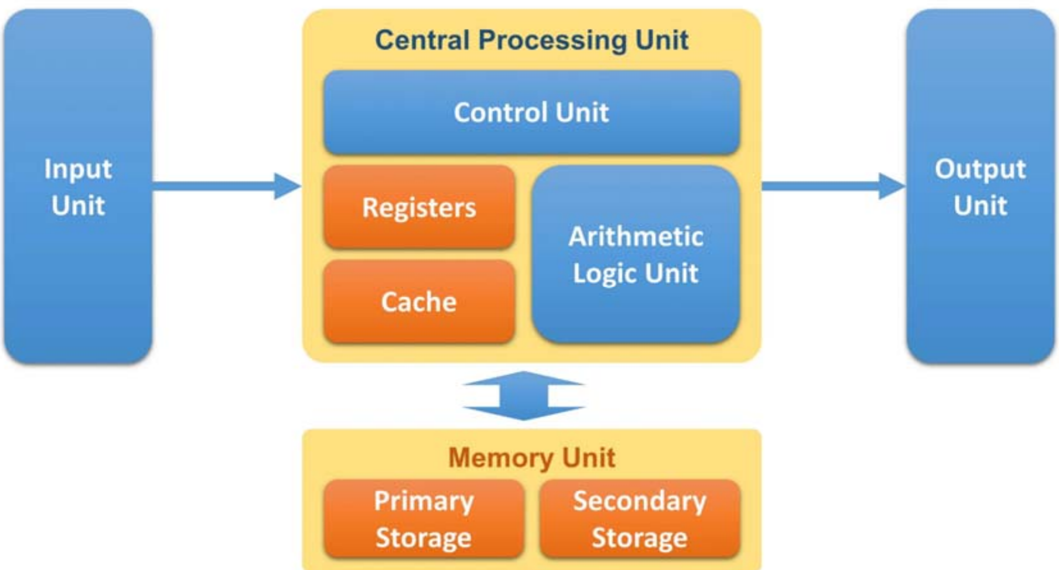
1. Python’s Sequence Types
2. **Low-Level Arrays**
3. Dynamic Arrays and Amortization
4. Efficiency of Python’s Sequence Types
5. Using Array Based Sequences
6. Multidimensional Data Sets



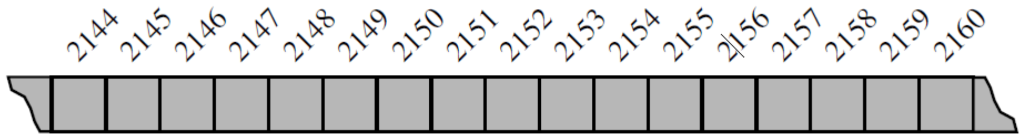


# Low Level Computer Architecture

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



# Memory, Address, Byte




Memory Addresses

Memory Locations (cells)

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

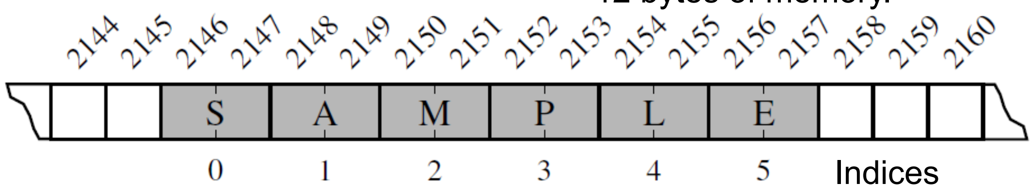
- A computer's main memory performs as Random Access Memory (RAM).
- That is, it is just as easy to retrieve byte #8675309 as it is to retrieve byte #309.
- Using the notation for asymptotic analysis, we say that any individual byte of memory can be stored or retrieved in $O(1)$ time.



# Array of Characters in the Computer's Memory

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

An *array of 6 characters* requiring 12 bytes of memory.



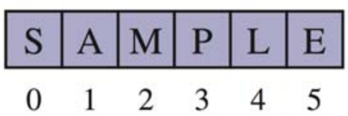

Memory Addresses

Memory Locations (cells)

A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires 2 bytes of memory. The numbers below the entries are indices into the string.

# A Higher-Level Abstraction for Array of Characters

- If one knows:
  - the memory address at which an array starts (e.g., 2146),
  - the number of bytes per element (e.g., 2 for a Unicode character), and
  - a desired index within the array,
- the appropriate memory address can be computed using the calculation, `start + cellsize * index`.
- The arithmetic for calculating memory addresses within an array can be handled automatically.
- Therefore, a programmer can envision a more typical high-level abstraction of an array of characters as diagrammed:



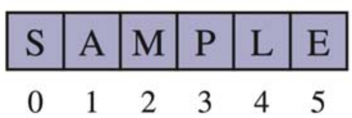

An array can store **primitive elements**, such as characters, giving us a **compact array**.

A higher-level abstraction for the string given before.

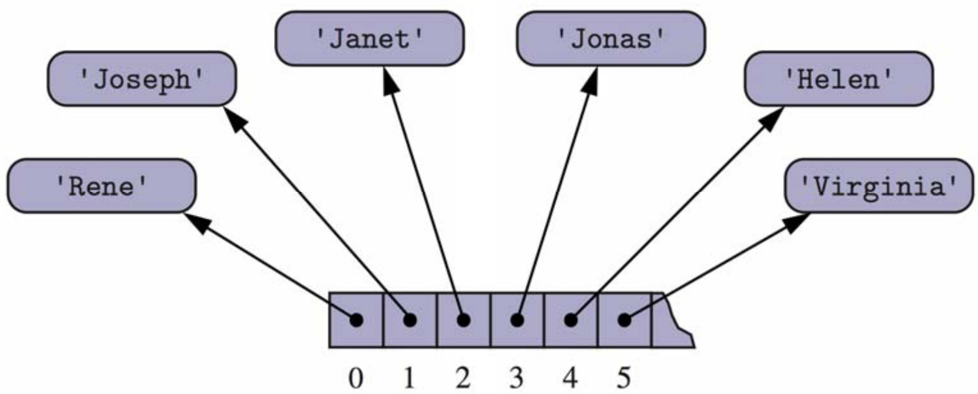


# Arrays of Elements or Object References

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

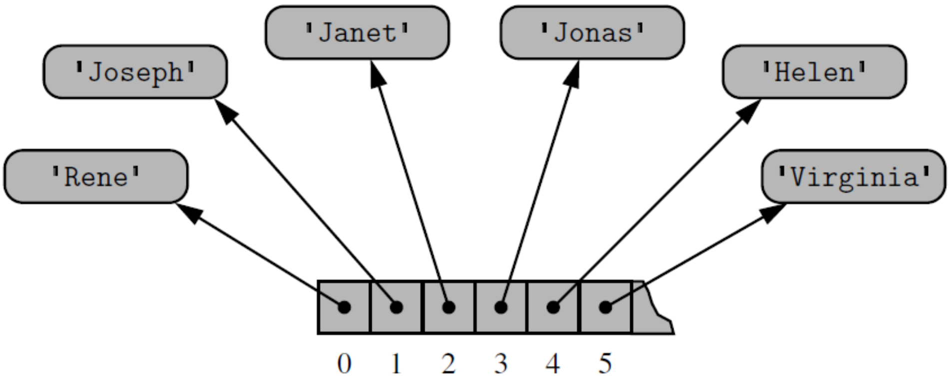



- An array can also store references to objects.



# Referential Arrays

- An array can store **references to objects**.
- Example: `['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', 'Virginia', ... ]`
- 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.
- A high-level diagram of such a list is shown below.



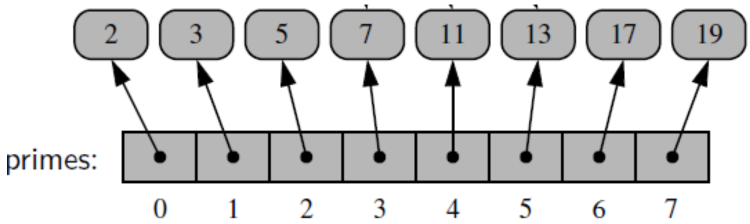



# Referential Arrays

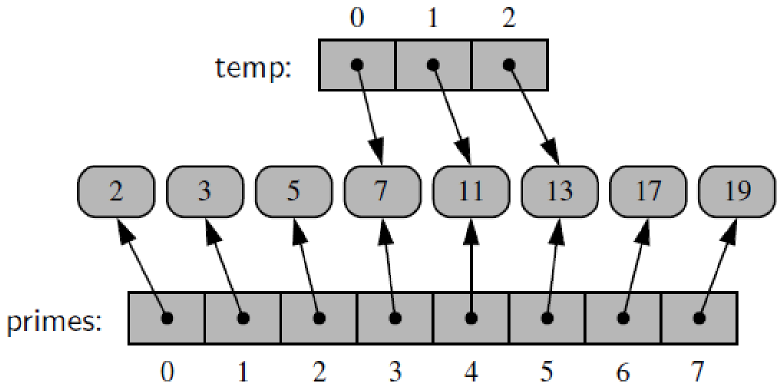
- The fact that lists and tuples are referential structures is significant to the semantics of these classes.
- A single list instance may include multiple references to the same object as elements of the list, and it is possible for a single object to be an element of two or more lists.

# An Object can be an Element of 2 or more Lists

- Example:




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

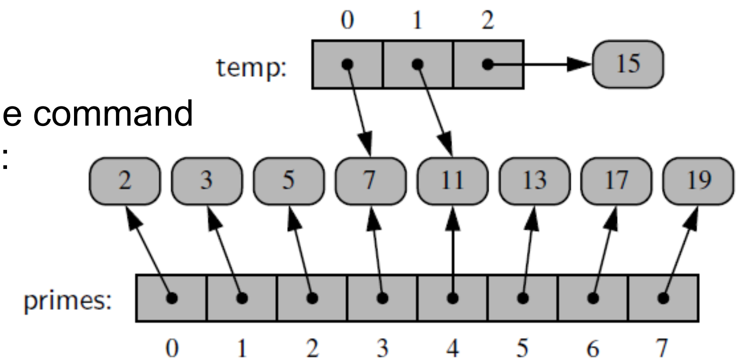





# If the Elements of the Lists are Immutable Objects

- When the elements of the list are immutable objects, as with the integer instances in Figure 5.5, the fact that the 2 lists share elements is not that significant, as neither of the lists can cause a change to the shared object.
- If, 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.

The result of the command
**`temp[2] = 15`**:



# Shallow Copy and Deep Copy

- The same semantics is demonstrated when making a new list as a copy of an existing one, with a syntax such as

```
backup = list(primes)
```

- This produces a new list that is a shallow copy (see Section 2.6), in that it references the same elements as in the first list.

- With immutable elements, this point is moot.

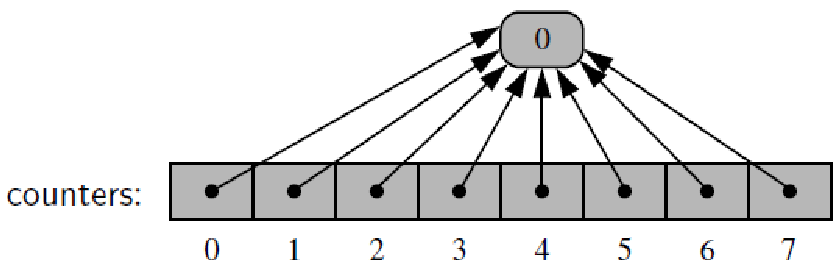
- If the contents of the list were of a mutable type, a deep copy, meaning a new list with new elements, can be produced by using the **deepcopy** function from the **copy** module.



# Initializing an Array of Integers

- As a more striking example, it is a common practice in Python to initialize an array of integers using a syntax such as **counters = [0]*8**.

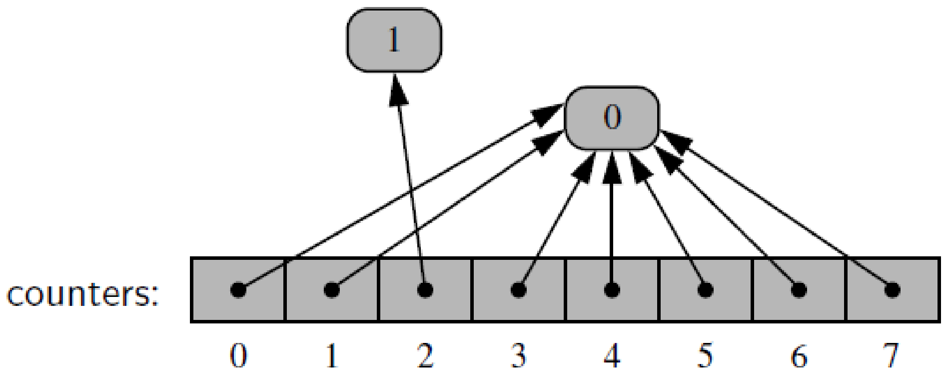
- This syntax produces a list of length 8, with all 8 elements being the value zero.
- All 8 cells of the list reference the same object:




The result of the command **counters = [0]*8**.

# Initializing an Array of Integers: Increment

- 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.



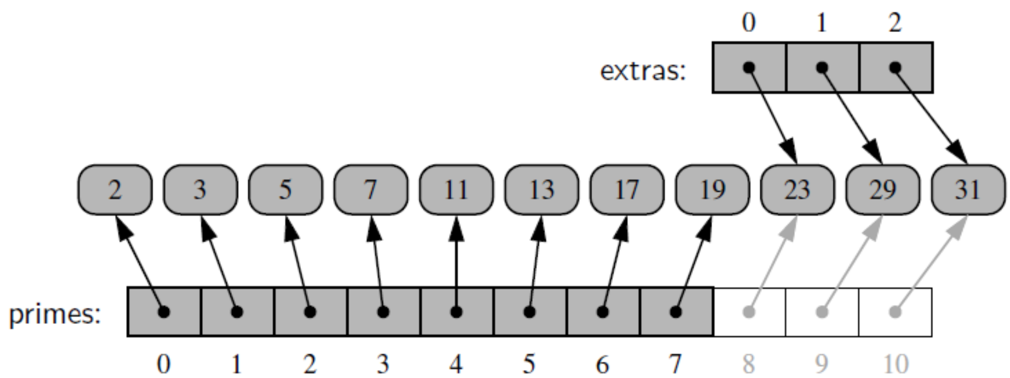

The result of command **`counters[2] += 1`**.



# Extending Array of Integers

Adding 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)`**, shown in light gray.

## Compact Arrays

- Primary support for compact arrays is in a **module** named `array`.
  - That module defines a **class**, also named **`array`**, providing **compact storage** for arrays of primitive data types.
  - Compact arrays have several advantages over referential structures in terms of computing performance.

- The **constructor** for the array class requires a **type code** *as a first parameter*, which is a character that designates the type of data that will be stored in the array.

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

Type code, i , designates an array of (signed) integers, typically represented using at least 16-bits each.



## Type Codes in the `array` Class

- Python's array class has the following type codes:

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

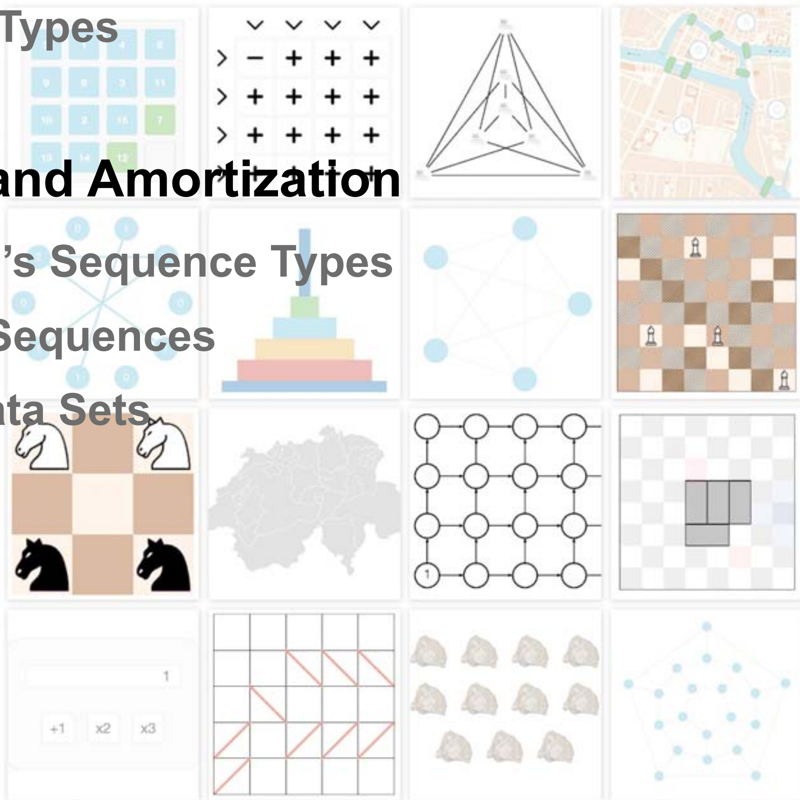
# User Defined Data Types: ctypes

- The array module does not provide support for making compact arrays of user defined data types.
- Compact arrays of such structures can be created with the lower level support of a module named **`ctypes`**. (See Section 5.3.1 for more discussion of the **`ctypes`** module.)



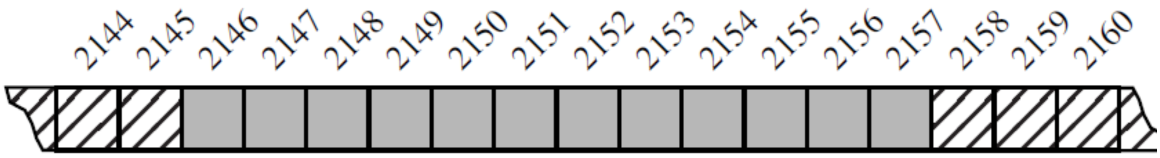
# Array-Based Sequences

1. Python’s Sequence Types
2. Low-Level Arrays
3. **Dynamic Arrays and Amortization**
4. Efficiency of Python’s Sequence Types
5. Using Array Based Sequences
6. Multidimensional Data Sets



# Low Level Static Array

- Example:




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

- Because the system might dedicate neighboring memory locations to store other data, the capacity of an array cannot trivially be increased by expanding into subsequent cells.



# Dynamic Arrays

- In the context of representing a Python **`tuple`** or **`str`** instance, constraint is no problem.
- Instances of those classes are immutable, so the correct size for an underlying array can be fixed when the object is instantiated.

- 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 algorithmic sleight of hand known as a dynamic array.

## Example: Exploring the Relationship Between a List's Length and its Underlying Size in Python

```
import sys      # provides getsizeof function

data = [ ]

for k in range(10):            # NOTE: must fix choice of n
    a = len(data)              # number of elements
    b = sys.getsizeof(data)    # actual the of bytes used to store an object
    print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
    data.append(None)          # increase length by one
```

```
Length:   0; Size in bytes:   56
Length:   1; Size in bytes:   88
Length:   2; Size in bytes:   88
Length:   3; Size in bytes:   88
Length:   4; Size in bytes:   88
Length:   5; Size in bytes:  120
Length:   6; Size in bytes:  120
Length:   7; Size in bytes:  120
Length:   8; Size in bytes:  120
Length:   9; Size in bytes:  184
```

An empty list instance already requires a certain number of bytes of memory (56 on my system).

It reports the number of bytes devoted to the array and other instance variables of the list, but *not* any space devoted to elements referenced by the list.



## The Test Program in Debug Mode Execution

```
import sys # provides getsizeof function

data = [ ]   data: [None, None, None, None, None, None, None, None, None]

for k in range(10):  # NOTE: must fix choice of n   k: 9
    a = len(data) # number of elements   a: 9
    b = sys.getsizeof(data) # actual size in bytes   b: 184
    print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
    data.append(None) # increase length by one
```

# The Test Program in Debug Mode Execution

```
01 a = {int} 9
01 b = {int} 184
data = {list: 9} [None, None, None, None, None, None, None, None, None]
   01 0 = {NoneType} None
   01 1 = {NoneType} None
   01 2 = {NoneType} None
   01 3 = {NoneType} None
   01 4 = {NoneType} None
   01 5 = {NoneType} None
   01 6 = {NoneType} None
   01 7 = {NoneType} None
   01 8 = {NoneType} None
   01 __len__ = {int} 9
```



# Analysis of the Program Output

Dynamic Memory Allocation

```
Length:   0; Size in bytes:   56
Length:   1; Size in bytes:   88
Length:   2; Size in bytes:   88
Length:   3; Size in bytes:   88
Length:   4; Size in bytes:   88
Length:   5; Size in bytes:  120
Length:   6; Size in bytes:  120
Length:   7; Size in bytes:  120
Length:   8; Size in bytes:  120
Length:   9; Size in bytes:  184
```

32 bytes increase

32 bytes

64 bytes

My experiment was run on a 64-bit machine architecture, meaning that each memory address is a 64-bit number (i.e., 8 bytes).

We speculate that the increase of 32 bytes reflects the allocation of an underlying array capable of storing 4 object references.

- No any underlying change in the memory usage after inserting the 2nd, 3rd, or 4th element into the list.
- After the 5th element has been added to the list, the memory usage jumps from 88 bytes to 120 bytes.
- The original base usage of 56 bytes for the list, the total of 120 suggests an additional 64 = 8×8 bytes that provide capacity for up to 8 object references.
- The memory usage does not increase again until the 9th insertion.
- At that point, the 184 bytes can be viewed as the original 56 plus an additional 128-byte array to store 16 object references.

# Appending `None` to the `List`

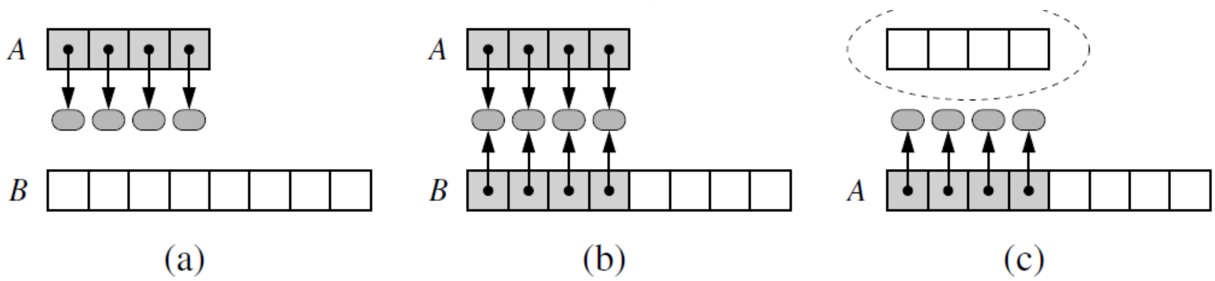
- Because a list is a referential structure, the result of **`getsizeof`** for a list instance only includes the size for representing its **primary structure**.
- It does not account for memory used by the objects that are elements of the list.

- In our experiment, we repeatedly append **`None`** to the list, because we do not care about the contents.
- 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 $i = 0, \ldots, n-1$, 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.




An illustration of the 3 steps for "growing" a dynamic array:
(a) create new array *B*;
(b) store elements of *A* in *B*;
(c) reassign reference *A* to the new array.

Not shown is the future garbage collection of the old array, or the insertion of the new element.

## An Implementation of a DynamicArray class, using a raw array from the ctypes module as storage.

```python
import ctypes                                      # provides low-level arrays

class DynamicArray:
  """A dynamic array class akin to a simplified Python list."""

  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
```



## An Implementation of a DynamicArray class, using a raw array from the ctypes module as storage.

```python

  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
```

# Amortized Analysis of Dynamic Arrays

- A detailed analysis of the running time of operations on dynamic arrays is presented.
- We use the big-Omega notation introduced in Section 3.3.1 to give an asymptotic lower bound on the running time of an algorithm or step within it.

- The strategy of replacing an array with a new, larger array might at first seem slow.
- Because a single append operation may require $\Omega(n)$ time to perform, where n is the current number of elements in the array.
- Using an algorithmic design pattern called amortization, we can show that performing a sequence of such append operations on a dynamic array is actually quite efficient.



# Amortized Analysis of Dynamic Arrays

- 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.
- In this way, there are many simple append operations for each expensive one (see Figure 5.13 give).

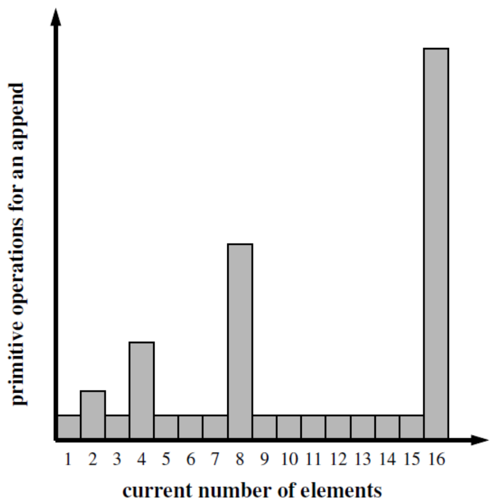



**Figure 5.13:** Running times of a series of append operations on a dynamic array.

## Amortized Analysis of Dynamic Arrays

- **Proposition 5.1:** *Let S be a sequence implemented by means of a dynamic array* ***with initial capacity one****, using the strategy of* ***doubling the array size*** *when full.*

  *The total time to perform a series of n append operations in S, starting from S being empty, is* $O(n)$.

- Read the justification given in the book.



## An Accounting Technique

- 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.
- Thus, the total amount of cyberdollars spent for any computation will be proportional to the total time spent on that computation.
- The beauty of using this analysis method is that we can overcharge some operations in order to save up cyber-dollars to pay for others.

# Illustration of a Series of Append Operations

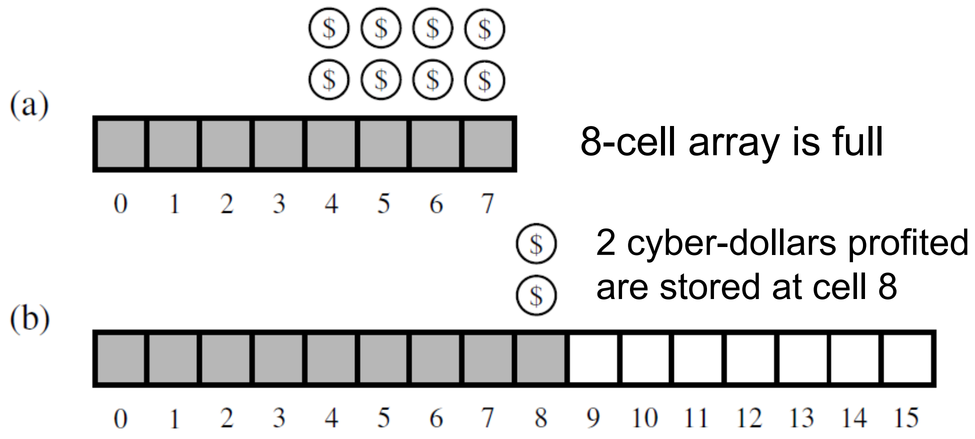



Figure 5.14: Illustration of a series of append operations on a dynamic array: (a) an 8-cell array is full, with 2 cyber-dollars "stored" at cells 4 through 7;

(b) an append operation causes an overflow and a doubling of capacity. Copying the 8 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 2 cyber-dollars profited are stored at cell 8.



# Other Ways to Increase Capacity

- **Geometric Increase in Capacity**
  - When choosing the geometric base, there exists a tradeoff between run-time efficiency and memory usage.
  - With a base of 2 (i.e., doubling the array), if the last insertion causes a resize event, the array essentially ends up twice as large as it needs to be.
  - If we increase the array by only 25% of its current size (i.e., a geometric base of 1.25), we do not risk wasting as much memory in the end, but there will be more intermediate resize events along the way.

# Other Ways to Increase Capacity

- **Beware of Arithmetic Progression**
  - A **constant number of additional cells** are reserved each time an array is resized.
  - Unfortunately, the overall performance of such a strategy is significantly worse.

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



# Other Ways to Increase Capacity

- **Beware of Arithmetic Progression**
  - At an extreme, an increase of only one cell causes each append operation to resize the array, leading to a familiar $1+2+3+\cdots+n$ summation and $\Omega(n^2)$ overall cost.
  - Using increases of 2 or 3 at a time is slightly better, as portrayed in Figure 5.13, but the overall cost remains quadratic.

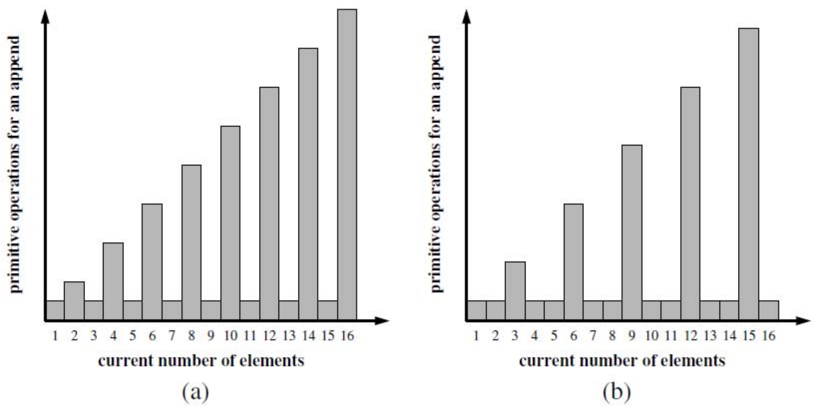



**Figure 5.15:** Running times of a series of append operations on a dynamic array using arithmetic progression of sizes. (a) Assumes increase of 2 in size of the array, while (b) assumes increase of 3.

# Growable Array-based Array List

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

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

- ……



# Comparison of the Strategies

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

# Incremental Strategy Analysis

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

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

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



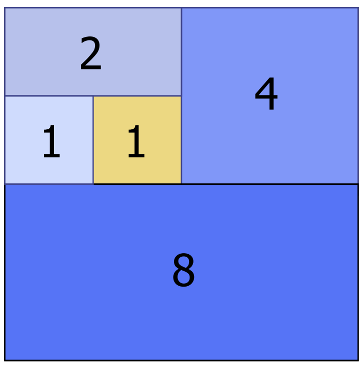
# Doubling Strategy Analysis

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

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

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

geometric series



# Python's List Class

- Python's list class is using a form of dynamic arrays for its storage.
- Yet, a careful examination of the intermediate array capacities (see Exercises R-5.2 and C-5.13) suggests that Python is not using a pure geometric progression, nor is it using an arithmetic progression.



# Measuring the Amortized Cost of Append

```
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 = [ ]
  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
```

**Code Fragment 5.4:** Measuring the amortized cost of append for Python's list class.

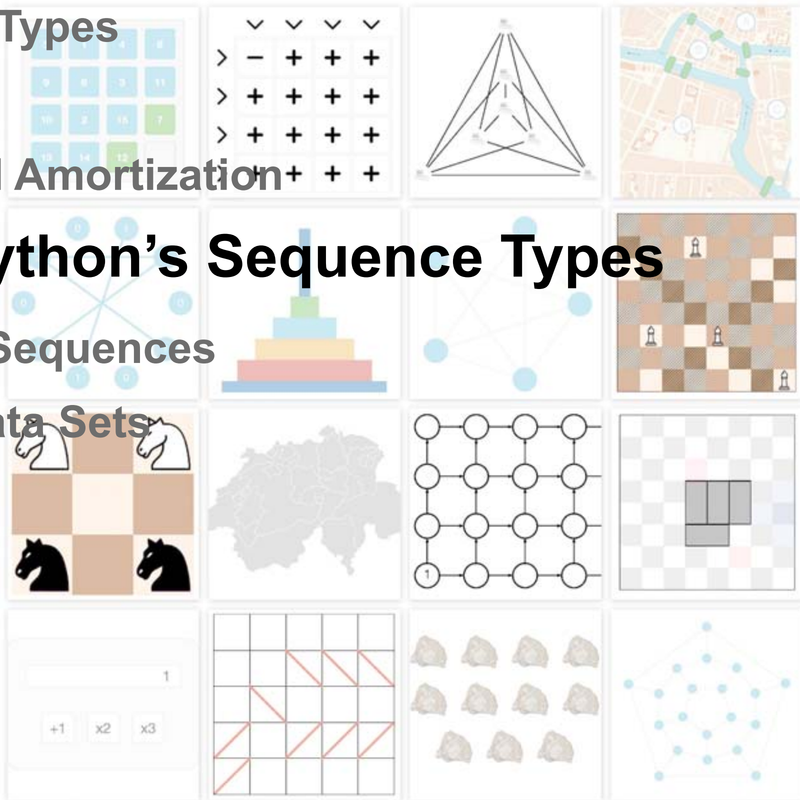
Taken as a whole, there seems clear evidence that the amortized time for each append is **independent of *n***.

| **n** | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 | 10,000,000 | 100,000,000 |
|---|---|---|---|---|---|---|---|
| $\mu s$ | 0.219 | 0.158 | 0.164 | 0.151 | 0.147 | 0.147 | 0.149 |

**Table 5.2:** Average running time of append, measured in microseconds, as observed over a sequence of *n* calls, starting with an empty list.

# Array-Based Sequences

1. Python’s Sequence Types
2. Low-Level Arrays
3. Dynamic Arrays and Amortization
4. **Efficiency of Python’s Sequence Types**
5. Using Array Based Sequences
6. Multidimensional Data Sets





# Python Collections (Arrays)

1. **List** is ordered and changeable. Allows duplicate members.
2. **Tuple** is a is ordered and unchangeable. Allows duplicate members.

3. **Dictionary** is a collection which is ordered** and changeable. No duplicate members.
4. **Set** is a is unordered, unchangeable*, and unindexed. No duplicate members.

*Set items are unchangeable, but you can remove and/or add items whenever you like.

**As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

# Python Lists

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

- Lists are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data.
  - The other 3 are **`Tuple`**, **`Set`**, and **`Dictionary`**, all with different qualities and usage.
- A list is a collection which is ordered and changeable.
- Lists are written with square brackets.

https://www.w3schools.com/python/python_lists.asp



# Python Tuples

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

- Tuples are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data.
  - The other 3 are **`List`**, **`Set`**, and **`Dictionary`**, all with different qualities and usage.
- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets.

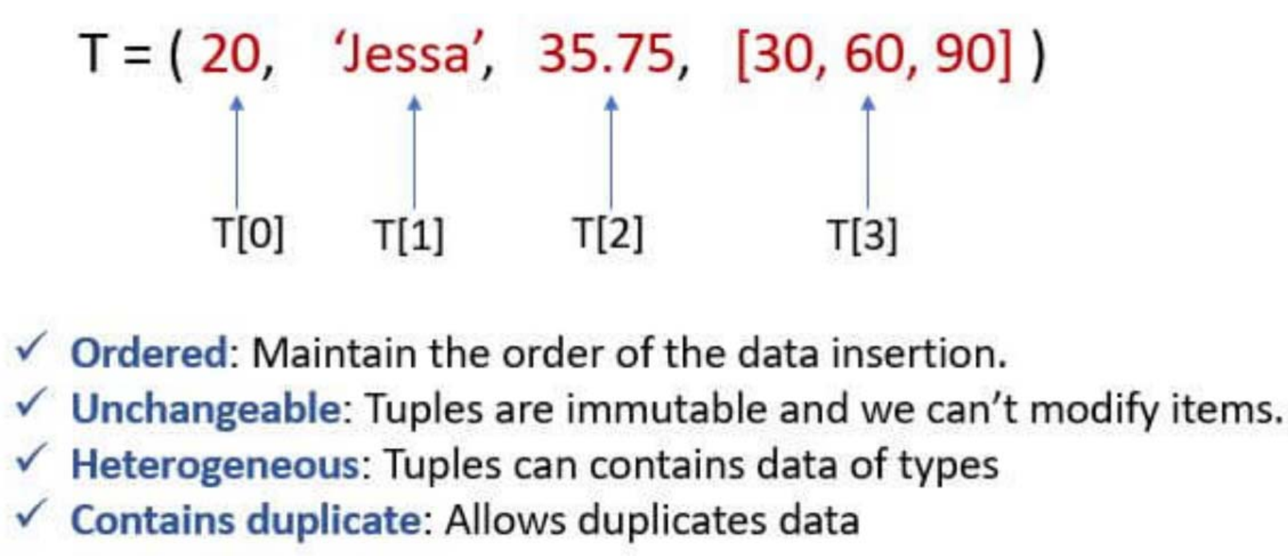
https://www.w3schools.com/python/python_tuples.asp

# Tuples: Immutable Data Structure

- The nonmutating (unchangeable) 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 because they are immutable;
- Therefore, there is no need for an underlying dynamic array with surplus capacity.



# Tuples: Example




https://pynative.com/python-tuples/

# Python's List and Tuple Classes

| 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 | $O(cn)$ |

**Table 5.3:** Asymptotic performance of the **nonmutating** behaviors of the list and tuple classes. Identifiers data, data1, and data2 designate instances of the list or tuple class, and *n*, *n*1, and *n*2 their respective lengths. 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, we let *k* denote the leftmost index at which they disagree or else $k = \min(n1, n2)$.



# Mutating Behaviors of the List Class

| 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)<br>**del** data[k] | $O(n-k)^*$ |
| data.remove(value) | $O(n)^*$ |
| data1.extend(data2)<br>data1 += data2 | $O(n_2)^*$ |
| data.reverse() | $O(n)$ |
| data.sort() | $O(n \log n)$ |

*amortized

**Table 5.4:** Asymptotic performance of the **mutating** behaviors of the list class. Identifiers data, data1, and data2 designate instances of the list class, and n, n1, and n2 their respective lengths.

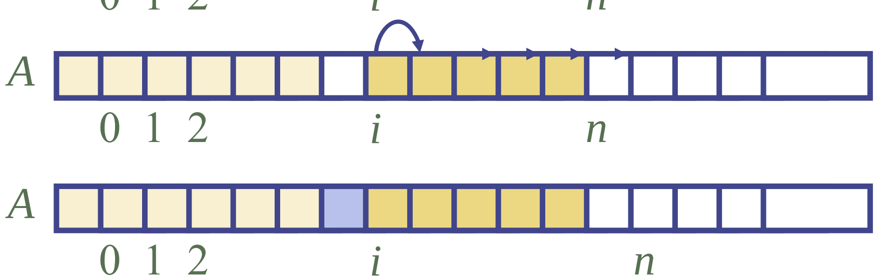
# Adding Elements to a List

- In Section 5.3 we fully explored the append method.
- In the worst case, it requires Ω(n) time because the underlying array is resized, but it uses O(1) time in the amortized sense.
- Lists also support a method, with signature **`insert(k, value)`**, that inserts a given value into the list at index 0 ≤ k ≤ n while shifting all subsequent elements back one slot to make room.



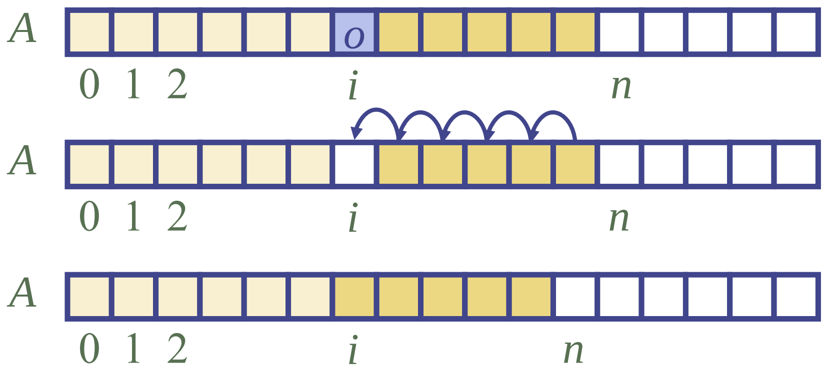
# Insertion

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



# Element Removal

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



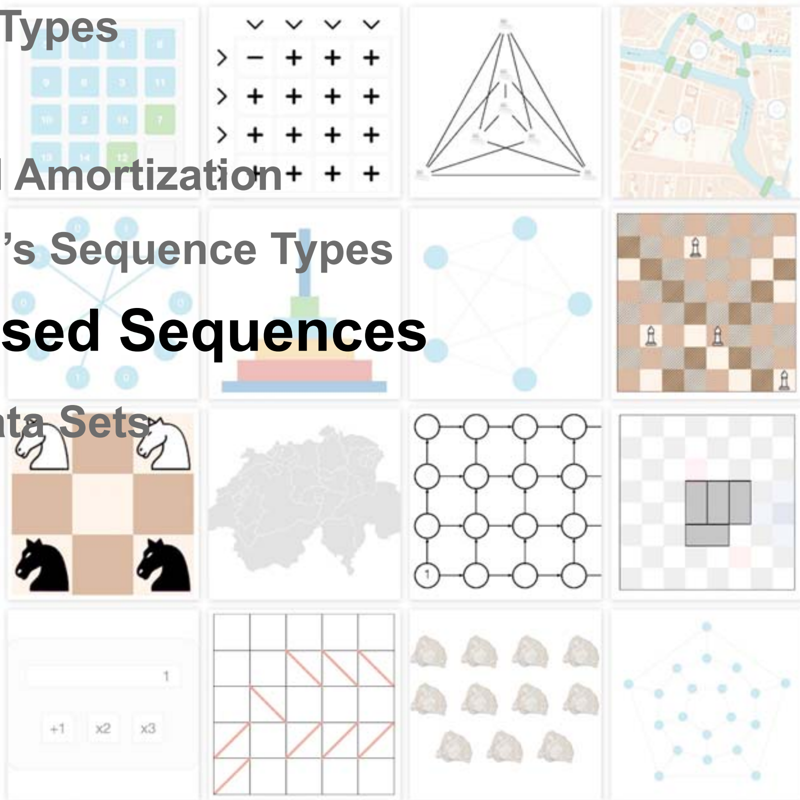



# Summary: Performance

- In an array based implementation of a **dynamic list**:
  - The space used by the data structure is ***O*(*n*)**.
  - Indexing the element at I takes ***O*(*1*)** time.
  - **`add`** and **`remove`** run in ***O*(*n*)** time in worst case.
- In an **`add`** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.

# Array-Based Sequences

1. Python's Sequence Types
2. Low-Level Arrays
3. Dynamic Arrays and Amortization
4. Efficiency of Python's Sequence Types
5. **Using Array Based Sequences**
6. Multidimensional Data Sets





# Storing High Scores for a Game

- This is representative of many applications in which a sequence of objects must be stored.

- We could just as easily have chosen to store
  - records for patients in a hospital or
  - the names of players on a football team.

- Nevertheless, let us focus on storing high score entries for a video game, which is a simple application that is already rich enough to present some important data-structuring concepts.

# GameEntry Class

```
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)'
```

What information to include in an object representing a high score entry?

**Code Fragment 5.7:** Python code for a simple GameEntry class. We include methods for returning the name and score for a game entry object, as well as a method for returning a string representation of this entry.



# Scoreboard

- To maintain a sequence of high scores, we develop a class named Scoreboard.
- A scoreboard is limited to a certain number of high scores that can be saved; once that limit is reached, a new score only qualifies for the scoreboard if it is strictly higher than the lowest "high score" on the board.

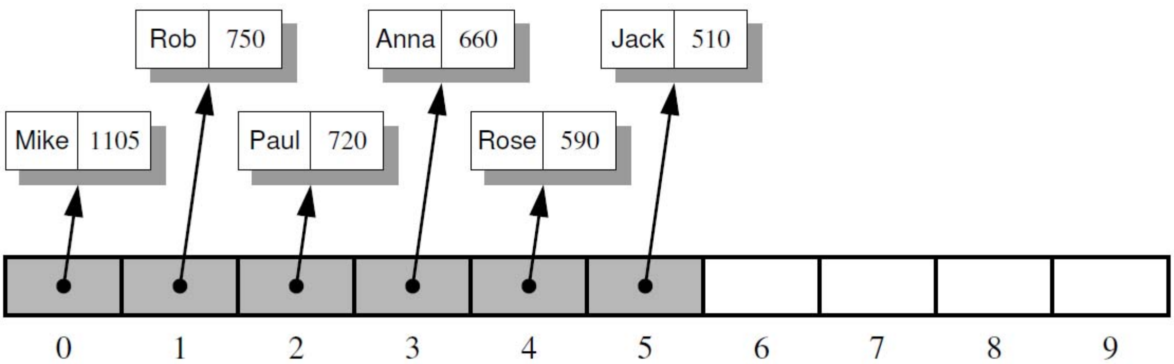



**Figure 5.18:** An illustration of an ordered list of length 10, storing references to 6 GameEntry objects in the cells from index 0 to 5, with the rest being None.

# Scoreboard Class

```
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))

```

List

the number of actual entries currently in our table.

initially set all entries to **None**.



# Scoreboard Class

```
  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
```

As entries are added, we will maintain them from highest to lowest score, starting at index 0 of the list.

# Adding an Entry

- This process is quite similar to the implementation of the insert method of the list class, as described on pages 204–205.

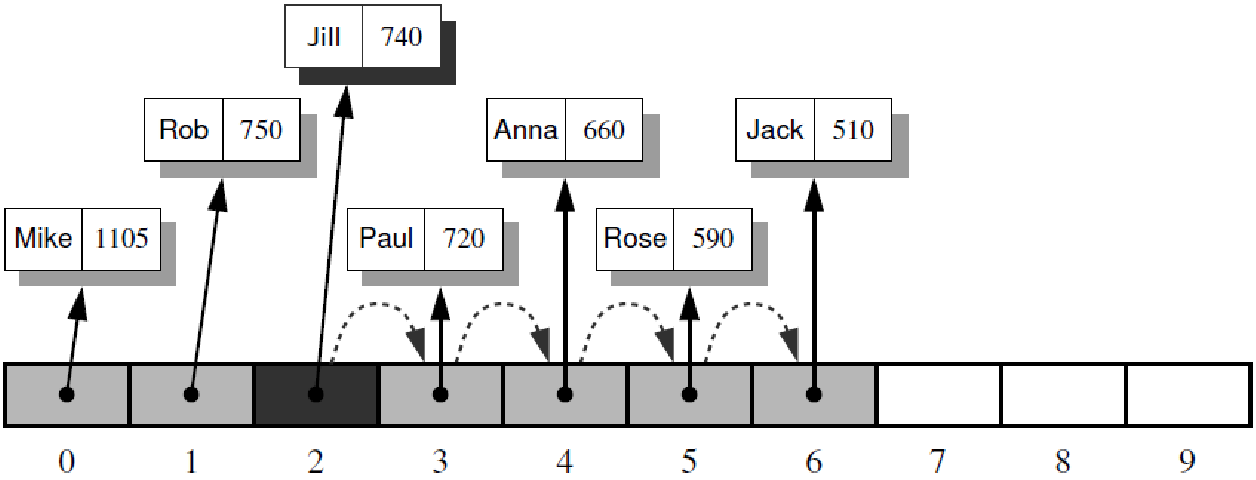



**Figure 5.19:** Adding a new GameEntry for Jill to the scoreboard. In order to make room for the new reference, we have to shift the references for game entries with smaller scores than the new one to the right by one cell. Then we can insert the new entry with index 2.



# Sorting a Sequence: Insertion Sort Algorithm

- In this section, we use a similar technique to solve the sorting problem, that is, starting with an unordered sequence of elements and rearranging them into nondecreasing order.……

```python
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
```

**Code Fragment 5.10:** Python code for performing insertion-sort on a list.

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.

# Execution of the Insertion-Sort Algorithm

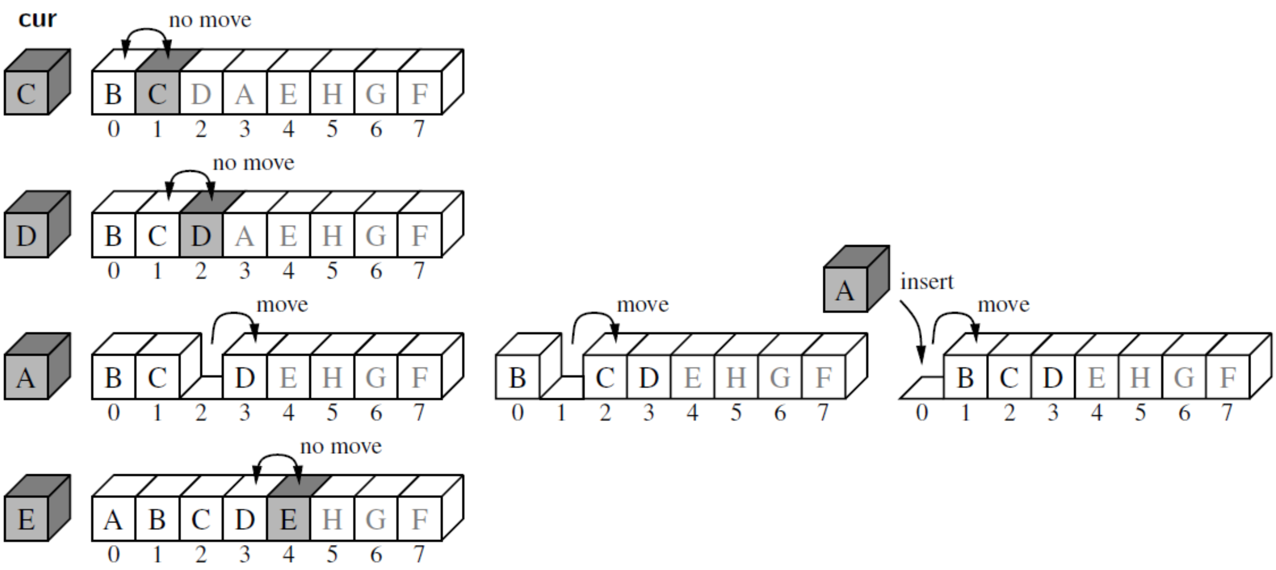



**Figure 5.20:** Execution of the insertion-sort algorithm on an array of 8 characters. Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array, and shown as the cur value.



# Execution of the Insertion-Sort Algorithm

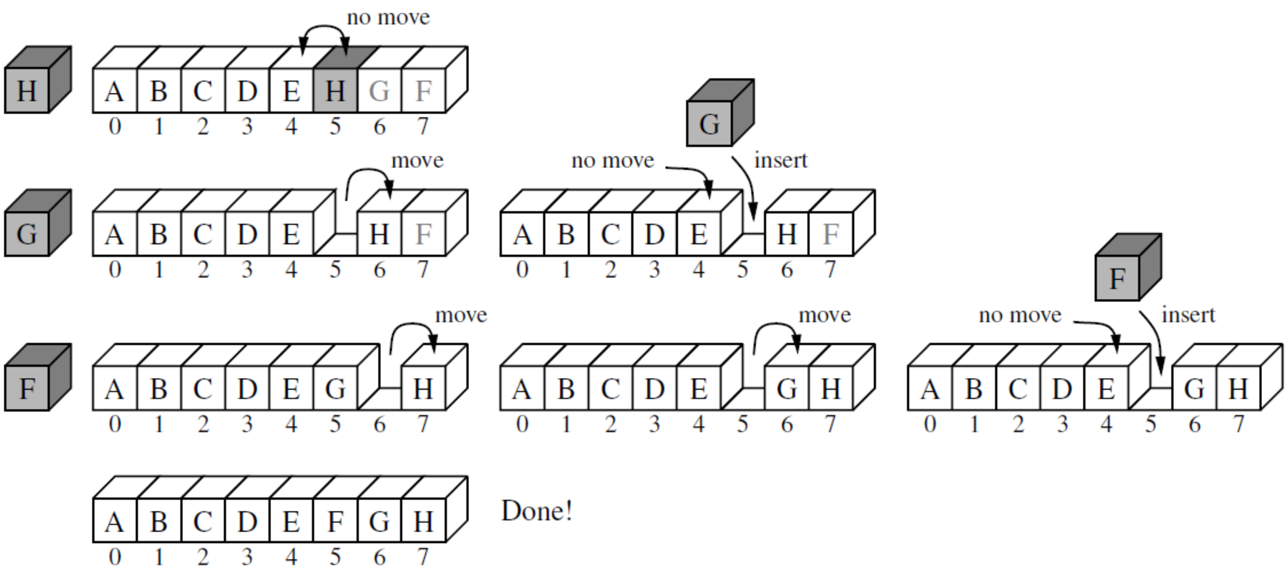



**Figure 5.20:** Execution of the insertion-sort algorithm on an array of 8 characters. Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array, and shown as the cur value.