CSc 120
Introduction to Computer Programming II

Adapted from slides
by Dr. Saumya Debray

09: Python lists
some performance puzzlers
Example 1: insert vs append

**insert**: adds an element into the middle of a list

```python
>>> list0 = [1,2,3,4]
>>> list0
[1, 2, 3, 4]
>>> list0.insert(2, 'aaa')
>>> list0
[1, 2, 'aaa', 3, 4]
>>> list0.insert(3, 'bbb')
>>> list0
[1, 2, 'aaa', 'bbb', 3, 4]
```

**append**: adds an element at the end of a list

```python
>>> list0 = [1,2,3,4]
>>> list0
[1, 2, 3, 4]
>>> list0.append('aaa')
>>> list0
[1, 2, 3, 4, 'aaa']
>>> list0.append('bbb')
>>> list0
[1, 2, 3, 4, 'aaa', 'bbb']
```
Example 1: insert vs append

**insert**: adds an element into the middle of a list

```python
list0 = mklist(n)  # length of list0 == n
list0.insert(n//2, 0)  # insert at midpoint
```

**append**: adds an element at the end of a list

```python
list0 = mklist(n)
list0.append(0)  # add at the end
```

Why this difference?
Example 2: mk_nxn_list

# given a number \( n > 0 \), return an \( n \times n \) list-of-lists of \( n \)

## Version 1

```python
def mk_nxn_list(n):
    return [[n]*n]*n
```

```python
given n = 2:
>>> mk_nxn_list(2)
[[2, 2], [2, 2]]
given n = 3:
>>> mk_nxn_list(3)
[[3, 3, 3], [3, 3, 3], [3, 3, 3]]
given n = 4:
>>> mk_nxn_list(4)
[[4, 4, 4, 4], [4, 4, 4, 4], [4, 4, 4, 4], [4, 4, 4, 4]]
```

## Version 2

```python
def mk_nxn_list(n):
    outlist = []
    for i in range(n):
        row_i = []
        for j in range(n):
            row_i.append(n)
        outlist.append(row_i)
    return outlist
```

```python
given n = 2:
>>> mk_nxn_list(2)
[[2, 2], [2, 2]]
given n = 3:
>>> mk_nxn_list(3)
[[3, 3, 3], [3, 3, 3], [3, 3, 3]]
given n = 4:
>>> mk_nxn_list(4)
[[4, 4, 4, 4], [4, 4, 4, 4], [4, 4, 4, 4], [4, 4, 4, 4]]
```
mk_nxn_list(n) version 1
mk_nxn_list(n) version 2

![Graph showing the relationship between input size n and run time.](image-url)
mk_nxn_list(n) both versions

Why this difference?
data organization in memory
Data organization in memory

**insert:** adds an element into the middle of a list

```python
>>> list0 = [1, 2, 3, 4]
>>> list0
[1, 2, 3, 4]
>>> list0.insert(2, 'aaa')
>>> list0
[1, 2, 'aaa', 3, 4]
>>> list0.insert(3, 'bbb')
>>> list0
[1, 2, 'aaa', 'bbb', 3, 4]
```

**append:** adds an element at the end of a list

```python
>>> list0 = [1, 2, 3, 4]
>>> list0
[1, 2, 3, 4]
>>> list0.append('aaa')
>>> list0
[1, 2, 3, 4, 'aaa']
>>> list0.append('bbb')
>>> list0
[1, 2, 3, 4, 'aaa', 'bbb']
```

- The organization of lists in memory affects the implementation of primitive operations
Data organization in memory

Consider list insertion

```python
>>> list0 = [1, 2, 3, 4]
>>> list0
[1, 2, 3, 4]
>>> list0.insert(2, 'aaa')
>>> list0
[1, 2, 'aaa', 3, 4]
>>> list0.insert(3, 'bbb')
>>> list0
[1, 2, 'aaa', 'bbb', 3, 4]
```
Data organization in memory

- Computer memory is organized as a sequence of *locations*
  - each location is identified by its *address* (a number)
  - a location typically consists of 8 bits (a "byte")
  - bytes are often grouped into "words" (32 or 64 bits)

→ A location (or word) can only hold a limited amount of data
Data organization in memory

- A memory location can hold only a limited amount of data
- An object typically spans multiple memory locations
- Data are organized as follows:
  - objects and values are placed where memory is available
  - the object's memory address is used as a reference to it

E.g.: for \( L = ["aaa", "bbb"] \)
Data organization in memory

We typically write these data structures in a way that abstracts away actual address values.
insert vs. append
List (array) organization in Python

• (References to) the list elements are kept in a contiguous sequence of memory words
  – there is a little extra space at the end to give it some room to grow

• The following operations are O(1):
  – len()
    o read off length info from the header
  – accessing the i\(^\text{th}\) element of the list
    o compute its address using the value of i
    o access memory location at that address
Appending to a list

L

info about the list
“aaa”
“bbb”
“ccc”
“ddd”
“eee”
“fff”
“ggg”
“hhh”

L.append(‘qqq’)

info about the list
“aaa”
“bbb”
“ccc”
“ddd”
“eee”
“fff”
“ggg”
“hhh”
“qqq”

O(1)
Inserting into a list

L

<table>
<thead>
<tr>
<th>info about the list</th>
</tr>
</thead>
<tbody>
<tr>
<td>“aaa”</td>
</tr>
<tr>
<td>“bbb”</td>
</tr>
<tr>
<td>“ccc”</td>
</tr>
<tr>
<td>“ddd”</td>
</tr>
<tr>
<td>“eee”</td>
</tr>
<tr>
<td>“fff”</td>
</tr>
<tr>
<td>“ggg”</td>
</tr>
<tr>
<td>“hhh”</td>
</tr>
</tbody>
</table>

L

<table>
<thead>
<tr>
<th>info about the list</th>
</tr>
</thead>
<tbody>
<tr>
<td>“aaa”</td>
</tr>
<tr>
<td>“bbb”</td>
</tr>
<tr>
<td>“ccc”</td>
</tr>
<tr>
<td>“ddd”</td>
</tr>
<tr>
<td>“eee”</td>
</tr>
<tr>
<td>“fff”</td>
</tr>
<tr>
<td>“ggg”</td>
</tr>
<tr>
<td>“hhh”</td>
</tr>
</tbody>
</table>

L.insert(2, ‘qqq’)

<table>
<thead>
<tr>
<th>info about the list</th>
</tr>
</thead>
<tbody>
<tr>
<td>“aaa”</td>
</tr>
<tr>
<td>“bbb”</td>
</tr>
<tr>
<td>“qqq”</td>
</tr>
<tr>
<td>“ccc”</td>
</tr>
<tr>
<td>“ddd”</td>
</tr>
<tr>
<td>“eee”</td>
</tr>
<tr>
<td>“fff”</td>
</tr>
<tr>
<td>“ggg”</td>
</tr>
<tr>
<td>“hhh”</td>
</tr>
</tbody>
</table>
Inserting into a list $O(n)$

- List elements:
  - “aaa”
  - “bbb”
  - “ccc”
  - “ddd”
  - “eee”
  - “fff”
  - “ggg”
  - “hhh”

- Inserting `qqq` into index 2:
  - L.insert(2, ‘qqq’)

- $O(n)$ list elements have to be moved over by one position.
## Python lists: complexity summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len</code></td>
<td>O(1)</td>
</tr>
<tr>
<td>Access an element's value</td>
<td>O(1)</td>
</tr>
<tr>
<td>Append</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insert, delete</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Q: Can we do insert in O(1) time?  
(The complexity of other operations may change)
Data organization in memory

Given references, what happens when we copy a list?
shallow vs. deep copying
Shallow vs. deep copying

For a *compound object* $A$ (i.e., an object that contains references to other objects, e.g., lists, dictionaries):

- **shallow copying:**
  - creates a copy $A'$ of $A$
  - inserts into $A'$ references to the objects in $A$

- **deep copying:**
  - creates a copy $A'$ of $A$
  - inserts into $A'$ references to deep copies of objects in $A$
References and “copy”

- **Shallow**
  - Copy references

- **Deep**
  - Copy everything
Shallow vs. deep copying

A: copied
Shallow vs. deep copying

def mk_nxn_list(n):
    return [[n]*n]*n
Shallow vs. deep copying

```python
def mk_nxn_list(n):
    return 
    
    n

time: O(1)
```
Shallow vs. deep copying

def mk_nxn_list(n):
    return [[n]*n]*n

time: O(1)
Shallow vs. deep copying

```python
def mk_nxn_list(n):
    return [[n]*n]*n

time: O(n)
```
Shallow vs. deep copying

def mk_nxn_list(n):
    return [[n]*n]*n

time: O(1)
Shallow vs. deep copying

def mk_nxn_list(n):
    return [[n]*n]*n

time: O(n)
Shallow vs. deep copying

```python
def mk_nxn_list(n):
    return [[n]*n]*n

time: O(n)
```
def mk_nxn_list(n):
    return [[n]*n]*n

Overall: O(n)
def mk_nxn_list(n):
    outlist = []
    for i in range(n):
        row_i = []
        for j in range(n):
            row_i.append(n)
        outlist.append(row_i)
    return outlist
Shallow vs. deep copying

def mk_nxn_list(n):
    outlist = []
    for i in range(n):
        row_i = []
        for j in range(n):
            row_i.append(n)
        outlist.append(row_i)
    return outlist

O(1)
Shallow vs. deep copying

```python
def mk_nxn_list(n):
    outlist = []
    for i in range(n):
        row_i = []
        for j in range(n):
            row_i.append(n)
        outlist.append(row_i)
    return outlist
```

$\mathcal{O}(n)$
def mk_nxn_list(n):
    outlist = []
    for i in range(n):
        row_i = []
        for j in range(n):
            row_i.append(n)
        outlist.append(row_i)
    return outlist
def mk_nxn_list(n):
    outlist = []
    for i in range(n):
        row_i = []
        for j in range(n):
            row_i.append(n)
        outlist.append(row_i)
    return outlist
Shallow vs. deep copying

def mk_nxn_list(n):
    outlist = []
    for i in range(n):
        row_i = []
        for j in range(n):
            row_i.append(n)
        outlist.append(row_i)
    return outlist

Overall: O(n^2)
## Shallow vs. Deep Copying: Howto

<table>
<thead>
<tr>
<th>Shallow Copy</th>
<th>Deep Copy</th>
</tr>
</thead>
</table>
| ```
>>> import copy
>>> x = [[1,2,3],[4,5,6]]
>>> y = copy.copy(x)
>>> y
[[1, 2, 3], [4, 5, 6]]
>>> x[0].append('abc')
>>> y
[[1, 2, 3, 'abc'], [4, 5, 6]]
``` | ```
>>> import copy
>>> x = [[1,2,3],[4,5,6]]
>>> y = copy.deepcopy(x)
>>> y
[[1, 2, 3], [4, 5, 6]]
>>> x[0].append('pqr')
>>> x
[[1, 2, 3, 'pqr'], [4, 5, 6]]
>>> y
[[1, 2, 3], [4, 5, 6]]
``` |
Summary

• Shallow vs. deep copying:
  – applies to any object, not just lists
  – how to:
    o import copy
    o copy.copy(obj), copy.deepcopy(obj)
  – shallow copying more efficient, but creates “aliases” that can cause weird behaviors
Aliases

• Aliasing occurs when more than one variable contains the same reference to the same object

• A reference is a location in memory

• Aliases are created when lists (and compound objects) are copied

• Aliases are also created through assignment
Data organization in memory

we will use the simplified diagram of a list for the next example
Example: aliasing

What is the content of each list after the following assignment statements?

```python
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append("foo")
list4[0] = None
```
**list1 = [2,3,5]**
**list2 = [7,11,13]**
**list3 = list1**
**list4 = list1+list2**

**list1[2] = -3**
**list3.append(“foo”)**
**list4[0] = None**

Remember, a list doesn't actually contain the values – it contains references to other objects.

But for simplicity, we often draw the object inside the list itself.
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append(“foo”)
list4[0] = None
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append("foo")
list4[0] = None

Example: aliasing

Assigning a list to a new variable creates an alias.
Example: aliasing

```python
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1+list2

list1[2] = -3
list3.append("foo")
list4[0] = None
```

List concatenation creates a brand-new list. It is a duplicate.
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append(“foo”)
list4[0] = None

Example: aliasing

Changing a value inside a list changes one element.

All of the aliases of the list see the same change.
Example: aliasing

In the same way, .append() modifies the list (and thus all aliases of it). It does **NOT** create a new list.

So .append() and + are different.
Example: aliasing

There is not an alias to `list4`.

The assignment does not affect the other lists.
Summary

• Python lists (“arrays”): properties
  – len : O(1)
  – accessing an element by index: O(1)
  – append: O(1)
  – insert, delete: O(n)

• Shallow vs. deep copying:
  – applies to any object, not just lists
  – how to:
    o import copy
    o copy.copy(obj), copy.deepcopy(obj)
  – shallow copying more efficient, but creates “aliases” that can cause weird behaviors
  – assignment also create aliases
timing programs
import time

def mklist(n):
    outlist = []
    for i in range(n):
        outlist.append(n)
    return outlist

# estimate overhead of timing loop
def get_overhead(niters):
    start_time = time.time()
    for i in range(niters):
        pass
    stop_time = time.time()
    overhead = stop_time - start_time
    return overhead

# do the timing
def do_time(listsz_start, listsz_stop, niters, overhead):
    listsz = listsz_start
    insertion_pt = listsz//2
    delta = 100
    while listsz <= listsz_stop:
        x = mklist(listsz)
        start_time = time.time()
        for i in range(niters):
            x.insert(insertion_pt, 0)
        stop_time = time.time()
        ave_time = (stop_time-start_time-overhead)/niters
        print("{:d}, {:f}".format(listsz, ave_time))
    if listsz >= 10*delta:
        delta *= 10
        listsz += delta     # next list size to try

def main():
    niters = 10000
    overhead = get_overhead(niters)
    do_time(100, 10000000, niters, overhead)

main()
# do the timing

def do_time(listsz_start, listsz_stop, niters, overhead):
    listsz = listsz_start
    insertion_pt = listsz//2
    delta = 100
    while listsz <= listsz_stop:
        x = mklist(listsz)
        start_time = time.time()
        for i in range(niters):
            x.insert(insertion_pt, 0)
        stop_time = time.time()
        ave_time = (stop_time-start_time-overhead)/niters

        print("{:d}, {:+f}".format(listsz, ave_time))

        if listsz >= 10*delta:
            delta *= 10
            listsz += delta  # next list size to try
# estimate overhead of timing loop

def get_overhead(niters):
    start_time = time.time()
    for i in range(niters):
        pass
    stop_time = time.time()
    overhead = stop_time - start_time
    return overhead

def main():
    niters = 10000
    overhead = get_overhead(niters)
    do_time(100, 100000000, niters, overhead)