6.0001 Final review session
Carrie & Amy

Topics:
aliasing/mutation/cloning,
lists/dictionaries/sets,
order of growth,
sorting
Aliasing

Two variables point to the same object

Analogy: two nicknames for the same person.

\[
a = \{"Boston": "MA", 
     "San Jose": "CA"\}
\]

\[
b = a
\]

Make \(b\) point to whatever object \(a\) is pointing to.
(Re)-Assignment

Associate an object with a variable.
We are just changing the association between a variable and an object.
We are not changing the object itself!

\[
a = \{"Boston": "MA"
   , "San Jose": "CA"
\}
\]

\[
b = a
\]

\[
a = "hi"
\]

Assign “hi” to a by moving a’s arrow.
Notice that b is still the dictionary
(because b’s arrow points to the dictionary still!)
Mutation

Change the object itself!

a = {"Boston": "MA"
     "San Jose": "CA"}
b = a

a["Cambridge"] = "MA"

Find the object that a is pointing to, and then mutate it!
A and B are now twins (they are different objects, but with same DNA)

Aliasing

Cloning
Lists

```python
>>> letters = ['a', 'b', 'd']

# basic functions
>>> len(letters)
3
>>> letters.append('e')
['a', 'b', 'd', 'e']
>>> letters.insert(2, 'c')
['a', 'b', 'c', 'd', 'e']
>>> letters.remove('a')
['b', 'c', 'd', 'e']
>>> letters.extend(['f', 'g'])
['b', 'c', 'd', 'e', 'f', 'g']

# more interesting functions
>>> letters.reverse()
['g', 'f', 'e', 'd', 'c', 'b']
>>> letters.pop()
'b'

>>> letters
['g', 'f', 'e', 'd', 'c']

# this does not mutate either list
>>> letters + ['a', 'b']
['g', 'f', 'e', 'd', 'c', 'a', 'b']
```
Practice problem

x = [1,2,3]
y = x
y.append(0)
z = x[1:3]
z.remove(2)
x.append(y)
x = "hi"

What are the values of x, y, and z after the code fully executes?
Practice problem

```python
x = [1,2,3]
y = x
x and y are [1,2,3]
y.append(0)
x and y are [1,2,3,0]
z = x[1:3]
z is [2,3] and x and y are [1,2,3,0]
z.remove(2)
z is [3] and x and y are [1,2,3,0]
x.append(y)
z is [3] and x & y are both [1,2,3,0,[1,2,3,0,[1,2,3,0]...]] which in python is [1,2,3,0,[...]]
x = "hi"
z is [3] and x is "hi" and y is [1,2,3,0,[...]]
```
Practice problem

x = [1, 2, 3]  # What are the values of x, y, and z after the code fully executes?
y = x
y.append(0)
z = x[1:3]
z.remove(2)
x.append(y)
x = "hi"

x is "hi"
y is [1, 2, 3, 0, [...]]
z is [3]
# Dictionaries

## Initialize using curly braces

```python
>>> zoo = {'elephant': 3, 'giraffe': 4}
```

# Find value given key

```python
>>> zoo['elephant']
3
```

## Look up value of non-existent key

```python
>>> zoo['cheetah']
KeyError: 'frog'
```

## Check if key exists

```python
>>> if 'cheetah' not in zoo:
    print("no cheetah")
```

## Provide default value if the key does not exist

```python
>>> zoo['cheetah', 0]
```

# Add key/value pair

```python
>>> zoo['cheetah'] = 5
```

## Change the value given key

```python
>>> zoo['cheetah'] = 10
```

## Delete key/value pair

```python
>>> del zoo['elephant']
```

## Delete non-existent k/v pair

```python
>>> del zoo['elephant']
KeyError: 'elephant'
```

## Lookup the keys

```python
>>> zoo.keys()
['cheetah', 'giraffe']
```

## Lookup the values

```python
>>> zoo.values()
[10, 4]
```
Sets

• An **unordered** collection of **unique** elements
• Elements in the set must be **immutable**
• Unordered means:
  – No positions or indexing
  – No way to tell the order of insertion
Sets

#initialize using curly braces
>>> zoo = set(['elephant','giraffe'])

# add element to set (won’t error even if already exists)
>>> zoo.add('cheetah')
3

# remove element
>>> zoo.remove('cheetah')

# remove non-existent element
>>> zoo.remove('cheetah')
KeyError: 'cheetah'

# check if element exists
>>> if 'cheetah' not in zoo:
    print “no cheetah”
Instantiation

• List: [element1, element2, element3]
• Tuple: (element1, element2, element3)
• Dictionary: {key1: val1, key2:val2, key3:val3}
• Set: set([element1, element2, element3])
So many [] usages!

Lists:
1) Instantiating a list: 
   ```python
   a = [‘a’,’b’,’c’]
   ```
2) Index into a list: 
   ```python
   a[0] = ‘z’
   ```

Dictionaries: use [] to get to the value given a key

1) Adding a new key/value pair to a dictionary 
   ```python
   locations = {“Boston”: “MA”} 
   locations[“San Jose”] = “CA”
   ```

2) Mutating an existing key/value pair: 
   ```python
   locations[“San Jose”] = “some other state”
   ```

3) Finding the value given a key 
   ```python
   print locations[“Boston”] → this prints “MA”
   ```
Indexing practice

cities = {'boston': {'population': 10000,
                     'landmark': ['charles']},
          'san francisco': {'population': 20000,
                           'landmark': ['goldengate']}}

Write code to print ‘charles’
print cities['boston']['landmark'][0]

Write code to add 500 to san francisco’s population
cities['san francisco']['population'] += 500

Write code to add ‘state’ and ‘MA’ as a key/value pair inside boston’s information
cities['boston']['state'] = 'MA'
Practice problem

```python
zoo = {'fish': 0, 'elephant': 3, 'giraffe': 4, 'gorilla': 5, 'frog': 1}

tally = {}
for animal in zoo:
    code = animal[0]
    if code not in tally:
        tally[code] = {'amount': zoo[animal], 'members': [animal]}
    else:
        tally[code]['amount'] += zoo[animal]
        tally[code]['members'].append(animal)

tallykeys = tally.keys()
tallykeys.sort()
for tallykey in tallykeys:
    print(tallykey)
    print(tally[tallykey]['amount'])
    print(tally[tallykey]['members'])
    print("-----")

tally[\'g\'][\'members\'][0][1] = \'d\'
```

What is the value of `tally` at this point?

What is printed?
Practice problem (solution)

What is the value of `tally` at this point? (order of appearance of k/v pair doesn’t matter)

```json
{ 'e': { 'amount': 3, 'members': [ 'elephant' ] },
'g': { 'amount': 9, 'members': [ 'gorilla', 'giraffe' ] },
'f': { 'amount': 1, 'members': [ 'fish', 'frog' ] }}
```

What is printed?

```
  e
  3
  [ 'elephant' ]
  
  f
  1
  [ 'fish', 'frog' ]  # don’t worry about the order of appearance these list elements
  
  g
  9
  [ 'gorilla', 'giraffe' ]  # don’t worry about the order of appearance these list elements
  
  Error
```
Orders of Growth

• big O refers to the upper bound (or worst case) scenario. Actual runtime can be faster.

• Definition: f(n) is “in” O(g(n)) if for sufficiently large n, f(n) \( \leq cg(n) \), where c is a positive real number

• Many dictionary operations takes constant time, e.g.
  ```python
  some_dictionary[some_key]  # get value for key
  if some_key in some_dictionary:  # check if key exists
  ```

• Ignore lower-order terms & constants
  \[ O(n^3 + n^2 + n + 1) \rightarrow O(n^3) \]
  \[ O(100000000n) \rightarrow O(n) \]
Order of Growth: strategies

• Look at loops
  – How many loops happen?
  – Inside each loop, how much work is done?
  – # loops x work/loop = total work

• Look at recursion
  – How many times is the recursive call made?
  – In each recursive call, how much work is done?
  – # recursive calls x work/call = total work
What are the orders of growth?

```python
def iter(n):
    while i < n:
        i += 1
    return i

def fib_iter(n):
    prev, curr, i = 0, 1, 1
    while i < n:
        prev, curr = curr, prev + curr
        i+=1
    return curr
```

\( \mathcal{O}(n) \)

\( \mathcal{O}(n) \): notice that, with respect to complexity, \( \text{fib\_iter} \) is similar to \( \text{iter} \), just with extra fluff.
What is the order of growth?

```
def iter(x, n):
    for i in range(x):
        print n
    return i
```

\(O(x)\)

Note that the order of growth in this case has no dependence on \(n\).
What is the order of growth?

def mystery(n):
    if n == 0:
        return
    for i in range(n):
        print "hi"
    mystery(n-1)

\[ O(n^2) \]
What is the order of growth?

```
def fibonacci(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

$O(2^n)$
What is the order of growth?

```python
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

$O(\log n)$
What is the order of growth?

def mystery(n):
    if n == 0:
        return

    mydict = {}
    for i in range(n):
        mydict[i] = 0
    mystery(n-1)

O(n^2)
What is the order of growth?

def mystery2(n):
    if n==0:
        return

    for i in range(n):
        for j in range(n):
            print "hi"
    mystery2(n-1)

O(n^3)
What is the order of growth?

def mystery3(n, x):
    for i in range(n):
        j = 0
        while j < x:
            j += 1

O(nx)
What is the order of growth?
(warning: very hard problem)

```python
def foo(n):
    if n < 1:
        return 2
    if n % 2 == 0:
        return foo(n - 1) + foo(n - 2)
    else:
        return 1 + foo(n - 2)
```

If n is odd \(\rightarrow O(n)\)
If n is even \(\rightarrow O(n^2)\)
Search algorithms

• Linear search
  – Go through each element and see if it matches the target element
  – Worst case, you look at each element once, so total work is $O(\text{len(numberlist)})$

• Binary search
  – if target > node, take right branch. If target < node, take left branch.
  – Recursively do this until you reach a node with no more child-nodes
  – Runtime is just the number of levels in the tree.
    How many levels of the tree? $O(\log(\text{num_nodes}))$
Selection Sort: $O(n^2)$

- At each step, the prefix is sorted, the suffix is unsorted.
- Keep lengthening the prefix and shortening the suffix.
- Append the min element from suffix to end of prefix.

```python
def selSort(L):
    """Assumes that L is a list of elements that can be compared using >. Sorts L in ascending order""
    prefixEnd = 0
    print 'List to be sorted =', L
    while prefixEnd != len(L):
        # look at each element in suffix
        for i in range(prefixEnd, len(L)):
            if L[i] < L[prefixEnd]:
                # swap position of elements
                L[prefixEnd], L[i] = L[i], L[prefixEnd]
        print 'Partially sorted list =', L
        prefixEnd += 1
```

This loop is $O(len(L))$.
This loop is $O(len(L)/2)$ on average.
Merge Sort  $O(n\log n)$

Break list in half $\rightarrow$ recursively sort both halves $\rightarrow$ merge the sorted halves

```
def mergeSort(L, compare = operator.lt):
    """Assumes L is a list, compare defines an ordering
    on elements of L
    Returns a new sorted list containing the same elements as L"""
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = mergeSort(L[:middle], compare)
        right = mergeSort(L[middle:], compare)
        return merge(left, right, compare)
```

Each merge takes $O(\text{len}(L))$ time because the two lists are already sorted

Height of tree is $O(\log(\text{len}(L)))$

Work per tree level is $O(\text{len}(L))$