# CS61A Lecture 16
## *Mutable Data Structures*

Jom Magrotker
UC Berkeley EECS
July 16, 2012

---

## Computer Science in the News
(Two Years Ago)

**Sony to end sales of 3.5 floppy disk; marks death of 30-year-old format**

BY MICHAEL SHERIDAN
DAILY NEWS STAFF WRITER
Tuesday, April 27, 2010

Say bye, bye to the 3.5.

After nearly 30 years, the once dominate 3.5-inch floppy disk will soon go the way of the cassette tape.

With the advent of CDs and later, DVDs, the use of the plastic floppys and its limited storage capacity were quickly deserted.

After the Apple G3, along with PCs, began shipping without the drives pre-installed the disks became virtually obsolete. However, the death of the format has only now become official with Sony's decision.

The company that virtually launched the format will completely abandon it in 2011, although began to end its production of the discs last year.

3.5 Floppy Disks are going the way of the audio cassette, as Sony officials... (Russell Illig/Getty )

http://articles.nydailynews.com/2010-04-27/news/27062899_1_marks-death-format-floppy-disk

2

---

## Computer Science in the News

**Computer Reads Manual, Plays Civ**

By Jim Rossignol on July 13th, 2011 at 7:38 pm.

MIT's Computer Science and Artificial Intelligence Lab **report** that they have boosted the effectiveness of a game-playing AI by enabling it to read the manual. "When the researchers augmented a machine-learning system so that it could use a player's manual to guide the development of a game-playing strategy, its rate of victory jumped from 46 percent to 79 percent."

http://www.rockpapershotgun.com/2011/07/13/computer-reads-manual-plays-civ/

**Would you like to play a game? New AI teaches itself the rules**

A Paris-based researcher has come up with a way for computers to learn how to play simple board games

An AI that can watch two-minute videos of some simple board games being played, learn the rules, and then play against human opponents has been developed by Lukasz Kaiser, a researcher at Paris Diderot University.

**MORE AI:** DARPA system to blend AI, machine learning to understand mountain of text

In the broad strokes, Kaiser's AI is a set of subroutines that work in concert -- a visual analysis system provides data to a game learning algorithm, and both are linked to an open-source game engine called Toss.

While the program is still unable to accuse you of cheating and leave the game in a huff, the level of sophistication displayed by Kaiser's invention is nonetheless highly impressive. Even more impressive is the fact that Kaiser's first tests of the AI -- on tic-tac-toe, Connect Four, Go-Moku, Pawns and Breakthrough -- were conducted on a laptop with a single-core processor and 4GB of RAM.

http://www.networkworld.com/news/2012/071112-ai-games-260821.html

3

---

## Today

- Review: Object-Oriented Programming
- Mutable Lists
- Identity and Equality
- Mutable Dictionaries

4

---

## Review: OOP Class Design

At Hogwarts, we want to write a class to store information about potions. How would you represent the following attributes?

- Name of the potion.
- Ingredients of the potion.
- How it interacts with another potion.
- Collection of all potions ever.

**CHOICES**
Instance Variable
Class Variable
Method

5

---

## Review: OOP Class Design

At Hogwarts, we want to write a class to store information about potions. How would you represent the following attributes?

- Name of the potion: ***Instance variable***
- Ingredients of the potion: ***Instance variable***
- How it interacts with another potion: ***Method***
- Collection of all potions ever: ***Class variable***

6

## Review: Mutable Data Structures

We have seen that objects possess state: they can change over time. Objects are thus ***mutable***.

The IRLists and IDicts we saw earlier were *im*mutable, which means that once created, they could not be modified.

Python has built-in list and dictionary data structures that *are* mutable.



## Mutable Lists: A Primer

```
>>> a = [3, 4, 5]
>>> a[0]
3
>>> a[1:]
[4, 5]
>>> a[0] = 7
>>> a
[7, 4, 5]
```

Slicing makes a new list.

Could not have done this with tuples!



## Mutable Lists: A Primer

```
>>> a
[7, 4, 5]
>>> a.append(10)
>>> a
[7, 4, 5, 10]
>>> a.extend([2, 3])
>>> a
[7, 4, 5, 10, 2, 3]
>>> a[2:4] = [6, 8]
[7, 4, 6, 8, 2, 3]
>>> a.pop()
3
>>> a
[7, 4, 6, 8, 2]
>>> a.remove(8)
[7, 4, 6, 2]
```

None of these operations create new lists. They all update the same list.

The slicing operator here does *not* make a new list. It refers to elements and positions in the original list.



## Mutable Lists: A Primer

```
>>> a
[7, 4, 6, 2]
>>> a[3] = a[0]
>>> a
[7, 4, 6, 7]
>>> a.count(7)
2
>>> a.index(4)
1
>>> a.reverse()
>>> a
[7, 6, 4, 7]
>>> a.sort()
>>> a
[4, 6, 7, 7]
>>> a.insert(1, 3)
>>> a
[4, 3, 6, 7, 7]
>>> del a[2]
>>> a
[4, 3, 7, 7]
```

None of these operations create new lists. They all update the same list.



## List Comprehensions

List comprehensions allow us to create new lists in the style of generator expressions.

```
>>> a = [2, 4, 6]
>>> b = [2*item for item in a]
>>> b
[4, 8, 12]
```

3. Add twice the element to the new list.

2. Call each element `item`.

1. Take the list a.



## List Comprehensions

List comprehensions allow us to create new lists in the style of generator expressions.

```
>>> a = [2, 4, 6]
>>> b = [2*item for item in a if item > 3]
>>> b
[8, 12]
```

3. Add twice the `item` to the new list…

2. Evaluate one element and call the result `item`.

1. Take the list a.

4. … but only if `item` is greater than 3.

# Working with Mutable Lists

Write a function called `square_list` that squares the items in a list. The function does *not* create a new list: it ***mutates*** the original list. (Assume the list is not deep.)

```
>>> my_list = [2, 7, 1, 8, 2]
>>> square_list(my_list)
>>> my_list[1]
49
```

We are *not* creating a new list.
We are *not* using the statement
`my_list = square_list(my_list)`



# Working with Mutable Lists

Write a function called `square_list` that squares the items in a list. The function does *not* create a new list: it ***mutates*** the original list. (Assume the list is not deep.)

```
def square_list(l):
    pos = 0
    while pos < len(l):
        l[pos] = l[pos] * l[pos]
        pos += 1
```

1. Initialize a variable that keeps track of the current position in the list.
2. As long as there are elements...
3. Update the current position in the list with the new value.
4. Move to the next position in the list.

No `return` statement!



# Comparing Lists and Tuples

```
def square_tuple(tup):
    results = ()
    for val in tup:
        results = results + (val*val,)
    return results

def square_list(l):
    pos = 0
    while pos < len(l):
        l[pos] = l[pos] * l[pos]
        pos += 1
```

A new tuple is created.

Items are added to the new tuple.

(Actually, new tuples are created in each iteration.)

The new tuple is returned.

The original list is modified.



# Working with Mutable Lists

Write a function called map_list that maps the function provided to the items of a list. The function does *not* create a new list: it *mutates* the original list.

```
>>> my_list = [2, 7, 1, 8, 2]
>>> map_list(lambda x: x**3, my_list)
>>> my_list[3]
512
```



# Working with Mutable Lists

Write a function called `map_list` that maps the function provided to the items of a list. The function does *not* create a new list: it *mutates* the original list.

```
def map_list(fn, l):
    pos = 0
    while pos < len(l):
        l[pos] = __________
        pos += 1
```



# Working with Mutable Lists

Write a function called `map_list` that maps the function provided to the items of a list. The function does *not* create a new list: it *mutates* the original list.

```
def map_list(fn, l):
    pos = 0
    while pos < len(l):
        l[pos] = fn(l[pos])
        pos += 1
```

## Announcements: Midterm 2

- Midterm 2 is on **Wednesday, July 25**.
  - *Where*? 2050 VLSB.
  - *When*? 7PM to 9PM.
  - *How much*? Material covered from July 4 until, and including, July 19 (from immutable trees until environment diagrams). You will also need to know material from Midterm 1.
- Closed book and closed electronic devices.
- One 8.5” x 11” ‘cheat sheet’ allowed.
- Group portion is 15 minutes long.
- Midterm review session on **Friday, July 20**.

19

## Announcements: Midterm 1

- Midterm 1 solutions will be released tonight.
- Midterm 1 regrade request protocol:
  - Attach, to the front, a sheet of paper describing the questions that you would like to be regraded, and the reasons why.
  - We reserve the right to regrade the entire midterm.
  - Regrade request deadline is end of day, **Thursday, July 26**.

20

## Announcements

- Homework 8 is due **Tuesday, July 17**.
- Homework 9 is due **Friday, July 20**.
- Project 3 is due **Tuesday, July 24**.

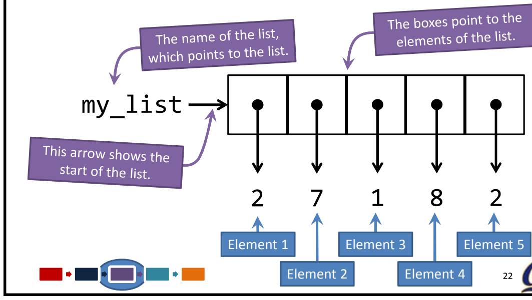
Please ***ask for help*** if you need to. There is a lot of work in the weeks ahead, so if you are ever confused, consult (in order of preference) your study group and Piazza, your TAs, and Jom.

***Don’t be confused!***

21

## Box-and-Pointer Diagrams

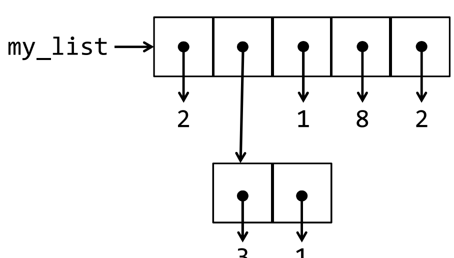
```
my_list = [2, 7, 1, 8, 2]
```




22

## Box-and-Pointer Diagrams

```
my_list = [2, [3, 1], 1, 8, 2]
```



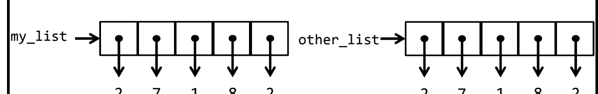

23

## Identity and Equality

```
>>> my_list = [2, 7, 1, 8, 2]
>>> other_list = [2, 7, 1, 8, 2]
>>> my_list == other_list
True
>>> my_list is other_list
False
```

my_list and other_list have the same values: they are *equal*.

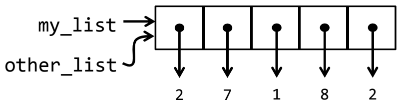
my_list and other_list are different lists: they are *not identical*.




24

## IDENTITY AND EQUALITY

```
>>> my_list = [2, 7, 1, 8, 2]
>>> other_list = my_list
>>> my_list == other_list
True
```
my_list and other_list have the same values: they are ***equal***.
```
>>> my_list is other_list
True
```
my_list and other_list are the same lists: they *are* ***identical***.






## IDENTITY AND EQUALITY

```
>>> kaushik = Person(...)
>>> kaushy = kaushik
>>> kaushik is kaushy
True
```
kaushik and kaushy are different names for the same object: they are ***identical***.
```
>>> kaushik == kaushy
True
```
Since kaushik and kaushy are different names for the same object, they have the same value, and are ***equal***.

Identity implies equality.



## IDENTITY AND EQUALITY

```
>>> kaushik = Person(...)
>>> shrivats = Person(...)
>>> shrivats is kaushik
False
```
kaushik and shrivats are names for different objects: they are ***not identical***.
```
>>> shrivats == kaushik
True
```
kaushik and shrivats are names for different objects that have the "same value", and are thus ***equal***.

Equality does not necessarily imply identity.



## IDENTITY AND EQUALITY: CAVEAT

By default, in Python, == acts the same as is.

a == b
is converted to
a.__eq__(b)

Classes can define what the "equality" of their instances means.

The Person class in the previous slide should implement the __eq__ method, or else
shrivats == kaushik
would have *also* evaluated to False.



## IDENTITY AND EQUALITY

The classes for the native data types in Python (numbers, Booleans, strings, tuples, lists, dictionaries) implement the __eq__ method.

```
>>> 3 == 4
False
>>> [1, 2, 3] == [1, 2, 3]
True
>>> True.__eq__(False)
False
```



## IDENTITY AND EQUALITY: PRACTICE

With the following expressions
```
>>> a = [2, 7, 1, 8]
>>> b = [3, 1, 4]
>>> a[2] = b
>>> c = [2, 7, [3, 1, 4], 8]
```
what do the following expressions evaluate to?
(A box-and-pointer diagram may be useful here.)
```
>>> a[2][0]
>>> a[2] is b
>>> a is c
>>> a == c
```

## Identity and Equality: Practice

With the following expressions

```
>>> a = [2, 7, 1, 8]
>>> b = [3, 1, 4]
>>> a[2] = b
>>> c = [2, 7, [3, 1, 4], 8]
```

what do the following expressions evaluate to?
(A box-and-pointer diagram may be useful here.)

```
>>> a[2][0]        2
>>> a[2] is b      True
>>> a is c         False
>>> a == c         True
```

31

## Identity and Equality: Practice

With the following expressions

```
>>> a = [2, 7, 1, 8]
>>> b = [3, 1, 4]
>>> a[2] = b
>>> c = [2, 7, [3, 1, 4], 8]
>>> a[2][1] = 3
```

what do the following expressions evaluate to?
(A box-and-pointer diagram may be useful here.)

```
>>> a[2][1]
>>> b[1]
>>> c[2][1]
```

32

## Identity and Equality: Practice

With the following expressions

```
>>> a = [2, 7, 1, 8]
>>> b = [3, 1, 4]
>>> a[2] = b
>>> c = [2, 7, [3, 1, 4], 8]
>>> a[2][1] = 3
```

what do the following expressions evaluate to?
(A box-and-pointer diagram may be useful here.)

```
>>> a[2][1]        3
>>> b[1]           3
>>> c[2][1]        1
```

33

## Identity and Equality: Practice

Implement the method __eq__ for the Pokemon class that will allow us to check if one Pokémon is "equal" to another, which means that it has the same HP as the other.

```
>>> ashs_pikachu = Pokemon('Pikachu', 'Ash', 300)
>>> brocks_pikachu = Pokemon('Pikachu', 'Brock', 300)
>>> ashs_pikachu is brocks_pikachu
False
>>> ashs_pikachu == brocks_pikachu
True
```

34

## Identity and Equality: Practice

Implement the method __eq__ for the Pokemon class that will allow us to check if one Pokémon is "equal" to another, which means that it has the same HP as the other.

```
class Pokemon:
    ...
    def __eq__(self, other):
        return ________________________________________
```

35

## Identity and Equality: Practice

Implement the method __eq__ for the Pokemon class that will allow us to check if one Pokémon is "equal" to another, which means that it has the same HP as the other.

```
class Pokemon:
    ...
    def __eq__(self, other):
        return self.get_hit_pts() == other.get_hit_pts()
```

36

## Survey Responses

Survey responses are generally very positive.

- "They have great pictures to help me learn."
- "Love the breaks: come back attentive."
- "Tom and Jon create a comfortable environment that invites questions."
- "I actually look forward to coming to lecture."
- "Needs more ponies."

http://www.gametrailers.com/side-mission/files/2012/06/my-little-pony-friendship-is-magic.jpg



## Survey Responses

- "Homework is more difficult than questions presented in lecture."

*This is intentional!* Lectures describe and explain the ideas; discussions and labs reinforce the ideas; homework and projects allow you to wrestle with, and learn, the material.

- *"*Please post the lecture slides earlier."

We will try and get them online by 9am, so that you have time to look over them if you need to. The answers to the questions will also be available soon after lecture.



## Survey Responses

- "Class goes too fast."

Again, we apologize, but summer courses tend to go faster than normal. Please don't hesitate to talk to your study group and the staff whenever you are stuck. *Don't be confused!*

- "More practical examples in lecture."

We will try to include more practical examples.

- "Lectures are behind the homework."

Yes, but we have now caught up!



## Survey Responses

- "Too many questions being asked in class."

We will try to address as many questions as we can without compromising on pace. Your questions are our constant feedback regarding whether the concept makes sense. So, keep asking questions!

*However*, if the question is tangential, or relates to other languages like Java or C, we may defer the question to after class, or office hours, or Piazza.

*Don't be afraid to ask questions!* We will definitely address your question, even if not in lecture. However, you **must** re-ask it, on Piazza, so we remember to answer it.



## Haiku Break

"I love comp science,
Bring on the problem, baby
This is what we do."

"CS not that hard.
But sometimes it no make sense."

"Computer science,
Solutions are ecstasy,
Debugging is hell."

"This is a haiku.
I do not want to write this.
But now I am done."

"Class is good.
I am learning lots.
Keep it up."

"To describe CS
With a short, succinct haiku.
Out of syllables."



## Mutable Dictionaries

Lists allow us to index elements by *integers*; dictionaries allow us to index elements by keys that are not necessarily integers.

This is particularly useful when we want to establish a correspondence between a descriptive *key* and a *value*.

## Mutable Dictionaries

Keys / Values

```
>>> roman_numerals = { 'I' : 1,
                       'V' : 5,
                       'X' : 10 }
>>> roman_numerals['V']
5
>>> roman_numerals.__getitem__('V')
5
>>> list(roman_numerals.keys())
['I', 'V', 'X']
>>> 'C' in roman_numerals
False
```



## Mutable Dictionaries

Write the function `count_words`, which returns a dictionary that maps a word to the number of times it occurs in a list of words.

```
>>> word_list = ['the', 'rain', 'in',
                 'spain', 'falls',
                 'mainly', 'in', 'the',
                 'plain']
>>> word_counts = count_words(word_list)
>>> word_counts['the']
2
```



## Mutable Dictionaries

```
def count_words(l):
    count_dict = {}
    for word in l:
        if word not in count_dict:
            count_dict[word] = 1
        else:
            count_dict[word] += 1
    return count_dict
```

1. Create a new dictionary for the counts.

2. For each word in the list of words...

3. If the word is not already in the dictionary...

4. Make a new entry for it.

5. Otherwise, update the existing entry.



## Comparing Dictionaries and IDicts

```
def count_words_idict(l):
    counts = make_idict()
    for word in l:
        count = idict_select(counts, word)
        if count is not None:
            counts = idict_update(counts, word, count+1)
        else:
            counts = idict_update(counts, word, 1)
    return counts
```

New immutable dictionaries are created in each iteration.



## Mutable Dictionaries: Practice

Write a function `tally_votes` that takes in a *list of tuples*, each of which has two elements: a candidate for an election, and number of votes received by that candidate in a particular state. `tally_votes` will return a dictionary that maps a candidate to the number of votes received by the candidate.

```
>>> votes = [('louis', 30), ('eva', 45), ('ben', 4),
             ('eva', 30), ('ben', 6), ('louis', 15)]
>>> total_votes = tally_votes(votes)
>>> total_votes['ben']
10
```



## Mutable Dictionaries: Practice

```
def tally_votes(votes):
    vote_dict = {}
    for vote in votes:
        candidate = _______
        vote_count = _______
        if ____________________________:
            __________________________________
        else:
            __________________________________
    return vote_dict
```

## Mutable Dictionaries: Practice

```
def tally_votes(votes):
    vote_dict = {}
    for vote in votes:
        candidate = vote[0]
        vote_count = vote[1]
        if candidate not in vote_dict:
            vote_dict[candidate] = vote_count
        else:
            vote_dict[candidate] += vote_count
    return vote_dict
```



## OOP: Class Methods

*Class variables* are variables shared by *all* instances of a class: they are not specific to a particular instance.

*Methods* are specific to a particular instance.

Occasionally, it would be useful to have a method that is *independent of any instance*, but *relevant to the class* as a whole.



## OOP: Class Methods

Suppose that we want to maintain a trainer directory in our Pokemon class, which would allow us to find the list of trainers that own a certain Pokémon.

This is not specific to a particular Pokemon object. As with `total_pokemon`, this is relevant to the Pokemon class as a whole.



## OOP: Class Methods

```
class Pokemon:
    total_pokemon = 0
    trainer_dir = {}

    def __init__(self, name, owner, hp):
        ...
        if name not in Pokemon.trainer_dir:
            Pokemon.trainer_dir[name] = [owner]
        else:
            Pokemon.trainer_dir[name].append(owner)
```



## OOP: Class Methods

We define a ***class method*** (or a ***static method***) that takes in the name of a Pokémon and returns a list of the trainers that own that Pokémon. We use the `staticmethod` ***decorator***.

```
class Pokemon:
    ...
    @staticmethod
    def trainers(name):
        if name in Pokemon.trainer_dir:
            return Pokemon.trainer_dir[name]
        return None
```



## OOP: Class Methods

```
>>> mistys_togepi = Pokemon('Togepi',
                            'Misty', 245)
>>> ashs_pikachu = ElectricPokemon('Pikachu',
                                   'Ash', 300)
>>> brocks_pikachu = ElectricPokemon('Pikachu',
                                     'Brock', 300)
>>> Pokemon.trainers('Pikachu')
['Ash', 'Brock']
>>> Pokemon.trainers('Charmander')
```

## CONCLUSION

- Python has built-in mutable lists and dictionaries. These can be *mutated*, which means that the original lists and dictionaries are modified.
- There are two different kinds of equality: one that checks if two objects are the same *(identical),* and one that checks if two objects have the same characteristics or values (*equal*).

  The second kind of equality can be defined by the class.
- OOP allows us to have methods that are relevant to the whole class, not just to specific instances.
- ***Preview***: Mutable recursive lists and environment diagrams.

55