CS61A Lecture 16
Mutable Data Structures

Jom Magrotker
UC Berkeley EECS
July 16, 2012

COMPUTER SCIENCE IN THE NEWS
(TWO YEARS AGO)

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


COMPUTER SCIENCE IN THE NEWS


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
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 immutable, 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!

>>> a
[7, 4, 5]

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]
>> a
[7, 4, 6, 6, 8, 2, 3]
>> a.pop()
>> 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.

MUTABLE LISTS: A PRIMER

>> a
[7, 4, 6, 2]
>> a[3] = a[0]
>> a
[4, 6, 7, 7]
>> a.sort()
>> a
[4, 6, 7, 7]

>> a.append(10)
>> a
[7, 4, 6, 7]
>> a.index(7)
2
>> a.reverse()
>> a
[7, 6, 4, 7]
>> a.sort()
>> a
[4, 3, 7, 7]

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

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]

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

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

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

COMPARING LISTS AND TUPLES

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

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

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

```python
def map_list(l):
    return map(lambda x: x**3, l)
```

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

```python
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**.

**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**.

**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!*

**BOX-AND-POINTER DIAGRAMS**

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

**IDENTITY AND EQUALITY**

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

```
my_list
2 7 1 8 2
3 1

other_list
2 7 1 8 2
```

```python
>>> my_list = [2, 3, 1, 1, 8, 2]
>>> my_list is other_list
False
```

```
my_list
2 7 1 8 2
3 1

other_list
2 7 1 8 2
```
IDENTITY AND EQUALITY

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

`my_list` and `other_list` have the same values; they are equal.

`my_list` and `other_list` are the same lists; they are identical.

---

IDENTITY AND EQUALITY

```python
>>> kaushik = Person(...)  # A dummy implementation of Person
>>> kaushy = kaushik
>>> kaushik is kaushy
True
>>> kaushik == kaushy
True
```

Since `kaushik` and `kaushy` are different names for the same object, they have the same value, and are equal.

---

IDENTITY AND EQUALITY

```python
>>> kaushik = Person(...)  # A dummy implementation of Person
>>> shrivats = Person(...)  # Another dummy implementation of Person
>>> shrivats is kaushik
False
>>> shrivats == kaushik
True
```

`kaushik` and `shrivats` are names for different objects; they are not identical.

---

IDENTITY AND EQUALITY: CAVEAT

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

```
>>> a = b
>>> a == b
False
```

This is because `==` is converted to `a.__eq__(b)` by default.

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: PRACTICE

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

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

---

IDENTITY AND EQUALITY: PRACTICE

With the following expressions:

```python
>>> 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.)

```python
>>> a[2][0]
>>> a[2] is b
>>> a is c
>>> a == c
```
IDENTITY AND EQUALITY: PRACTICE

With the following expressions

```python
>>> 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.)

```python
>>> a[2][0] = 2
>>> a[2] is b
>>> a is c
>>> a == c
```

IDENTITY AND EQUALITY: PRACTICE

With the following expressions

```python
>>> 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.)

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

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.

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

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.
**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.”

**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.”

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

“CS not that hard.
But sometimes it no make sense.”

“Class is good.
I am learning lots.
Keep it up.”

“Computer science,
Solutions are ecstasy,
Debugging is hell.”

“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**

```python
>>> 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.

```python
>>> word_list = ['the', 'rain', 'in', 'spain', 'falls', 'mainly', 'in', 'the', 'plain']

>>> word_counts = count_words(word_list)

>>> word_counts['the']
2
```

**Mutable Dictionaries**

```python
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
```

**Comparing Dictionaries and IDicts**

```python
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
```

**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.

```python
>>> 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**

```python
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 Pokémon object. As with total_pokemon, this is relevant to the Pokemon class as a whole.

**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.

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

**OOP: Class Methods**

```python
>>> 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')
None
```
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.