More lists, dictionaries, tuples

Frédo Durand and Ana Bell
MIT EECS, 6.00
QUIZ

• March 6
  – 7:30–9:30 pm in 32–123 and 10–250
• Review session Monday 7:30-9:30 in 32-123
• 2 sheets (4 pages) with anything you want written on them
• Academic conflicts?
  – Email Emily to reschedule emilyrsu@mit.edu
  – Deadline to arrange conflict exam is this Friday
## Quiz topics

<table>
<thead>
<tr>
<th>Topics</th>
<th>Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is computation</td>
<td>Dictionaries</td>
</tr>
<tr>
<td>Variables</td>
<td>Mutations &amp; aliasing</td>
</tr>
<tr>
<td>Type</td>
<td>Tuples</td>
</tr>
<tr>
<td>Control, loops</td>
<td>Debugging, exceptions</td>
</tr>
<tr>
<td>Guess/check, bisection</td>
<td>Recursion</td>
</tr>
<tr>
<td>Functions &amp; abstraction</td>
<td>Pset 0-3</td>
</tr>
<tr>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td></td>
</tr>
</tbody>
</table>
Dictionaries
Keep grades for students

students = ['Ana', 'Fredo']
grades = ['A', 'B']

def grade_of(name, sl, gl):
    i = sl.index(name)
    return grades[i]
students

', 'Fredo', 'B', ideally
name, sl, gl

(name)

 dez []
Keep grades for students

- Idea 1: keep two lists

```python
# Associating students and grades the hard way
students = ['Ana', 'Fredo', 'John', 'Srini']
grades = ['A', 'B', 'A+', 'A']
```
Keep grades for students

- Idea 1: keep two lists

```python
# Associating students and grades the hard way
students = ['Ana', 'Fredo', 'John', 'Srini']
grades = ['A', 'B', 'A+', 'A']

def grade_of_student(name, grade_list, student_list):
    i = student_list.index(name)
    return grade_list[i]

print grade_of_student('Ana', grades, students)
```

- But makes it messy to query & maintain
What we want
What we want

• Kind of like a list, but not indexed by integers, indexed by strings

• We’d like to write score[‘Ana’]
Another way: dictionary

- Like lists but store PAIRS of values: a key and a value
- Can be retrieved through the key
  - In a way, allows for non-integer indices

- Create using curly ‘{ }’ and ‘:’

```python
grades={'Ana':'A', 'Fredo':'B', 'John':'A+', 'Srini':'A'}
print(grades['Ana'])
```
What if no entry exists for key?
What if no entry exists for key?

- `grades['Name']` raises a `KeyError` if ‘Name’ not in keys
Dictionary goodies

- `grades[‘Eric’]=‘A’` adds an entry
- ‘Srini’ in grades tests if a key is in a dict.
- `del grades[‘Fredo’]` deletes an entry
- `grades.keys()` list of keys
- `grades.values()` list of values
Dictionaries

• Can be used with any type for **Values**
• Limited set of types for key
  – Immutable: no list or dictionary
  – bad idea to use floats
    (equality is unreliable)
• Keys must be unique (duh)
dictionary + list

grades={'Ana': ['A', 'A', 'A+'], 'Fredo': ['C', 'B', 'A']}
print grades['Ana']
Counting characters
Counting characters

```python
s = 'the quick brown fox jumped over the lazy dog'

count = {}

for c in s:
    if c in count:
        count[c] += 1
    else:
        count[c] = 1
```
Counting characters

```python
s = 'the quick brown fox jumped over the lazy dog'
count ={}

for char in s:
    if char in count:
        count[char] = count[char] +1
    else:
        count[char] = 1

print count```

EtoF = {'bread': 'du pain', 'wine': 'du vin',
    'eats': 'mange', 'drinks': 'boit',
    'likes': 'aime', 1: 'un',  'and': 'et',
    'everybody': 'tout le monde', 6.00: '6.00', 1: 'une'}

print EtoF['bread']
print EtoF
print EtoF.keys()
print EtoF.keys

def keySearch(L, k):
    for elem in L:
        if elem[0] == k: return elem[1]
    return None

def translateWord(word, dictionary):
    if word in dictionary:
        return dictionary[word]
    else:
        return word

def translate(sentence, dictionary):
    translation = ''
    word = ''
    for c in sentence:
        if c != ' ':
            word = word + c
        else:
            translation = translation + ' ' + translateWord(word, dictionary)
            word = ''
    return translation[1:] + ' ' + translateWord(word, EtoF)

print translate('John eats bread', EtoF)
print translate('Fredo drinks wine', EtoF)
print translate('John and everybody likes 6.00', EtoF)
Tuples
Returning multiple variables
Returning multiple variables

• I want functions that return multiple values

\[ i, \text{mini} = \text{mini\_with\_index}(L) \]

• with a syntax like

```python
return current\_index, current\_min
```
Tuples

- Kind of like lists
- But
  - immutable
  - convenient syntax for returning multiple function outputs
- Two syntaxes: with and without ()
  1, 2
  (1, 2)
def mini_with_index(L):
    assert len(L) > 0
    current_min = L[0]
    current_index = 0
    for i in xrange(1, len(L)):
        if L[i] < current_min:
            current_min = L[i]
            current_index = i
    return current_index, current_min

L = [1, 2, 4, 5, 0]

i, mini = mini_with_index(L)
print 'element', i, 'is the minimum: ', mini
Recall swapping variables

\[ x = y \]
\[ y = x \]

\[ \text{temp} = x \]
\[ x = y \]
\[ y = \text{temp} \]
Recall swapping variables

• start with \( x=1; y=2 \)

• DO NOT WRITE:
  
  \[
  x=y  \\
  y=x
  \]

• Instead
  
  \[
  \text{tmp}=x  \\
  x=y  \\
  y=\text{tmp}
  \]

– (Python also has a simpler way. More later.)
swapping variables with tuple

\[ x, y = y, x \]

unpacks packs
swapping variables with tuple

\[
\begin{align*}
x &= 1 \\
y &= 2 \\
x, y &= y, x \\
\text{print 'x=', x, ', y=', y}
\end{align*}
\]
Why tuples?

- Return multiple values
- Swap variables
- Store fixed-length data
  - e.g. 2D point coordinates
- Can be used as keys in dictionaries, which lists cannot

\[ \text{coord}_2D = x, y, z \]
Mutation and aliasing
Recall variables

\[
x_{1} = 1 \\
x_{2} = x_{1} \\
x_{1} = 2 \\
\text{print } x_{1} \quad \rightarrow 2 \\
\text{print } x_{2} \quad \rightarrow 1
\]
Assignment and lists

L1 = [1, 2, 3]
L2 = L1
L1.append(4)
print L1  => (1, 2, 3, 4)
print L2  => [1, 2, 3, 4]

L1  => (1, 2, 3)
L2
\texttt{L1 = [1, 2, 3]}
\texttt{L2 = L1}
\texttt{L1.append(4)}  \text{\hspace{1cm}} \text{\textarrow{mutation}}
\texttt{print L1} \quad \rightarrow \quad (1, 2, 3, 4)
\texttt{print L2} \quad \rightarrow \quad [1, 2, 3, 4]

\texttt{L1} \rightarrow (1, 2, 3)
\texttt{L2} \rightarrow \text{change the binding}
\texttt{L1 @ 4} \rightarrow 4
\texttt{L2 = [1, 2, 3, 4]}
Assignment and lists

L1 = [1, 2, 3]
L2 = L1
L1.append(4)
print L1
print L2

• Lists (and dictionaries) behave differently from simple types such as integers, floats, strings, and Booleans

• A list is an “object” inside the memory

• Variables only point to that object
  – When an object is modified (or “mutated”) all variables bound to it are affected
Python tutor

```
1 x1 = 1
2 x2 = x1
3 x1 = 2
4 print x1
5 print x2
6
7
8 L1 = [1, 2, 3]
9 L2 = L1
10 L1.append(4)
11 print L1
12 print L2
```
Merging lists

• \( L3 = L1 + L2 \) does NOT modify \( L1 \) but creates a copy

1. \( L1 = [1, 2] \)
2. \( L2 = [3] \)
3. \( L3 = L1 + L2 \)

```
no mutation
```
Extending a lists: extend

- extend does modify the list, unlike +

```python
>>> L=[1, 2, 3]
>>> L2=[4, 5]
>>> L.extend(L2)
>>> L
[1, 2, 3, 4, 5]
```
Important words

- **Mutation**
  - modification of an object

- **Mutable**
  - Type that can be mutated

- **Aliasing**
  - Two variables are bound to the same object
  - Mutation through one variable affects the other variable(s)

- **Side effect**
  - When a function mutates variables
Copy

- L2 = L1[:] actually creates a new list and copies all the elements of L1

```python
1 L1=[1, 2, 3]
2 L2=L1
3 L3=L1[:]
4 L1.append(4)
5 print L2
6 print L3
```

- Global variables:
  - L1
  - L2
  - L3

- Frames:
  - list
    - 0
    - 1
    - 2
    - 3

- Objects:
  - list
    - 0
    - 1
    - 2
    - 3
    - 4

Program terminated  Forward  Last
Mutation

Techs = ['MIT', 'Cal Tech']
Ivys = ['Harvard', 'Yale', 'Brown']
Univs = [Techs, Ivys]

Univs1 = [['MIT', 'Cal Tech'],
          ['Harvard', 'Yale', 'Brown']]

print 'Univs = ', Univs
print 'Univs1 =', Univs1

print 'throw Harvard out of Ivy league'
Ivys.remove('Harvard')
print 'Univs = ', Univs
print 'Univs1 =', Univs1
Convenient to update data

count characters in strings

def update_count (word, count)
Convenient to update data

```python
def update_char_count(count, word):
    for char in word:
        if char in count:
            count[char] = count[char] +1
        else:
            count[char] = 1

#note: no need to return side effect

count={}
update_char_count(count, 'fox')
print count
update_char_count(count, 'quick')
print count
```
Recap

• Unlike basic types, variables bound to list only point to the object in memory

• If two variables point to the same list (*aliasing*) and the list is modified (*mutated*) though one, the other one is affected too
  – This can cause surprises!

• deep copies can be obtained with \texttt{L[:]}
To the wise

• Play with the online Python tutor
• Do the MITx finger exercises
Sort

• L1=[3, 2, 1]
• L1.sort()
  – sorts L “in place” (mutation)
  – print L1 shows [1, 2, 3]
Sort

- \( L1 = [3, 2, 1] \)
- \( L1.sort() \)
  - sorts \( L \) “in place” (mutation)
  - print \( L1 \) shows [1, 2, 3]
- \( L1 = [3, 2, 1] \)
- \( L2 = \text{sorted}(L1) \) returns a new sorted version of \( L1 \) (no mutation)
  - print \( L2 \) shows [1, 2, 3]
  - print \( L1 \) shows [3, 2, 1]
Reverse

- L=[3, 1, 2]
- L.reverse() reverses the order in place (mutation)
- print L shows [2, 1, 3]
- .reversed is a little more complicated
  - for the more curious:
    - doesn’t return a list but can be used in a for:
    - for i in reversed(L):
Bad scenario

- function that prints top 3 temperatures

```python
temperatures= [-10, -11, -3, -4, -5, -8, 1]

def print_top_3(L):
    L.sort()
    for i in xrange(3):
        print L[-1-i]

print_top_3(temperatures)
print temperatures
```

has a nasty side effect