

EECS 12: Lecture 4

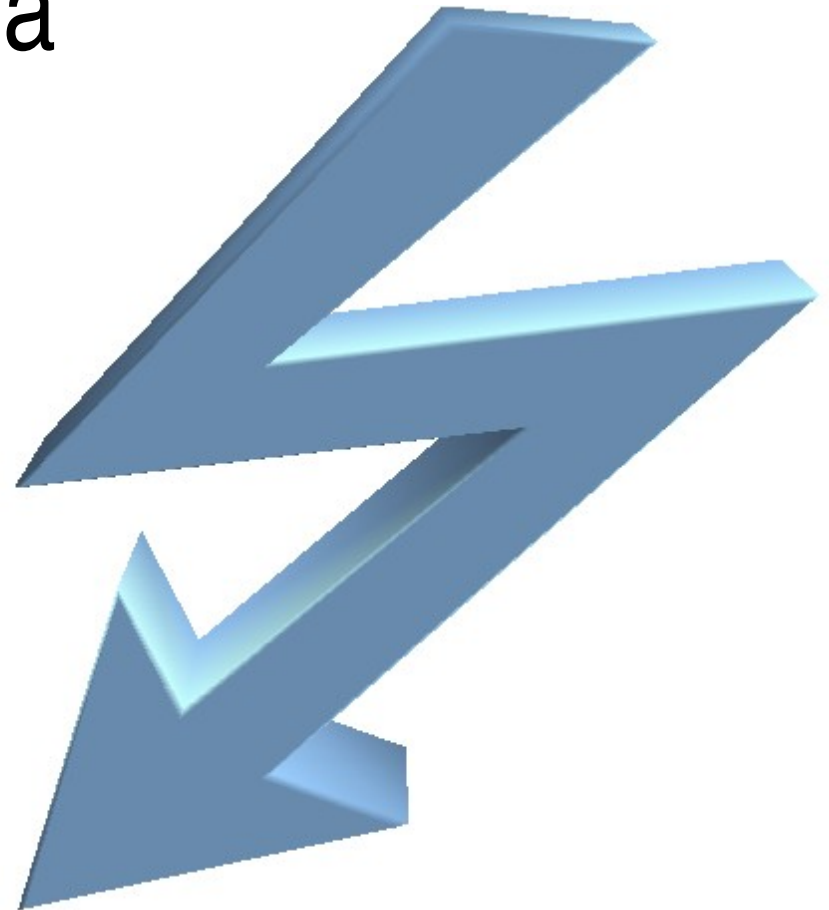
Tuples, Lists, Dictionaries

Mark E. Phair
mphair@gmail.com
UC Irvine EECS

July 10th, 2006

Agenda

- Tuples
- Lists
- Methods
- Dictionaries
- random Module
- Python factoid of the day



No, seriously. I mean, moose bites can be pretty nasty!

Tuples are like strings, but with stuff besides characters...

```
>>> tup = 1, 'cow', 3.5, (4, 'five')
```

```
>>> tup
```

```
(1, 'cow', 3.5, (4, 'five'))
```

```
>>> tup[0]
```

```
1
```

```
>>> tup[3]
```

```
(4, 'five')
```

tuple in a tuple



More tuple excitement

```
>>> tup = 1, 'cow', 3.5, (4, 'five')
```

```
>>> type(tup[0])
```

```
<type 'int'>
```

```
>>> type(tup)
```

```
<type 'tuple'>
```

```
>>> type(tup[3])
```

```
<type 'tuple'>
```

Even more tuple excitement!

```
>>> 1, # this will make a tuple because of the comma
```

```
(1,)
```

```
>>> (1) # this will not make a tuple: no comma
```

```
1
```

```
>>> type(1,2) # interpreted as multiple arguments
```

```
[error message]
```

```
>>> type((1,2))
```

```
<type 'tuple'>
```

Tuple arithmetic!

```
>>> a = 'w', 2, 1.0
```

```
>>> a + (4, 'cow', 6)
```

```
('w', 2, 1.0, 4, 'cow', 6)
```

```
>>> a*2
```


```
('w', 2, 1.0, 'w', 2, 1.0)
```

Tuple slicing!

```
>>> aTuple = 'a', 'b', 'c', 5, 1.0, 'cow'
>>> aTuple[1:]
('b', 'c', 5, 1.0, 'cow')
>>> aTuple[:2]
('a', 'b')
>>> aTuple[-4:]
('c', 5, 1.0, 'cow')
```

Steps in slicing (can do with strings, too)

```
>>> aTuple = 'a', 'b', 'c', 5, 1.0, 'cow'
>>> aTuple[1::2]
('b', 5, 'cow')
>>> aTuple[::1]
('a', 'b', 'c', 5, 1.0, 'cow')
>>> aTuple[-4::2]
('c', 1.0)
```



You can even use `in`!

```
>>> aTuple = 'a', 'b', 'c', 5, 1.0, 'cow'
```

```
>>> 'a' in aTuple
```

```
True
```

```
>>> for item in aTuple:
```

```
    print item, '#',
```

```
a#b#c#5#1.0#cow#
```

Lists are like tuples, but mutable!!!

```
>>> lst = [1, 'cow', 3.5, (4, 'five')]
```

```
>>> lst
```

```
[1, 'cow', 3.5, (4, 'five')]
```

```
>>> lst[1] = 'moo'
```

```
>>> lst
```

```
[1, 'moo', 3.5, (4, 'five')]
```

VERY mutable

```
>>> lst = [1, 'moo', 1.0, (4, 'five')]
```

```
>>> del lst[3]
```

(4 , ' five ') has been DELETED!

```
>>> lst
```

```
[1, 'moo', 1.0]
```

```
>>> lst[3:] = ['hi']
```

new stuff has been added

```
>>> lst
```

```
[1, 'moo', 1.0, 'hi']
```

Tuples are immutable, but not like strings

- You cannot change an individual value of a tuple

```
>>> tup = 1, 2, 3
```

```
>>> tup[1] = 5
```

```
[error]
```

list nested inside
of a tuple

- But.....

```
>>> tup = 1, 2, [3, 4]
```

```
>>> tup[2][0] = 5
```

```
>>> tup
```

```
(1, 2, [5, 4])
```

accessing the
list nested
inside

`id()` : unique for each object

```
>>> a,b = 'w', 'w'
```

```
>>> id('w'), id(a), id(b)
```

```
(-1209889024, -1209889024, -1209889024)
```

```
>>> id('ww')
```

```
-1210149056
```

id() : what about lists?

```
>>> c,d = ['w'], ['w']
>>> id(c), id(d)
(-1210164500, -1210150388)
>>> c = c + [] # doing "nothing"
>>> del d[0]
>>> id(c), id(d)
(-1210150164, -1210150388)
```

Aliasing

```
>>> a = [[1,2],[3,4]]
```

```
>>> b = a[0]
```

```
>>> b
```

```
[1, 2]
```

```
>>> b[0] = 5
```

ACK!

```
>>> a,b
```

```
[[5, 2], [3, 4]], [5, 2]
```



Cloning

```
>>> a = [[1, 2], [3, 4]]
```

```
>>> b = a[0][:]
```

slice!



```
>>> b
```

```
[1, 2]
```

```
>>> b[0] = 5
```

Yay!



```
>>> a, b
```

```
[[1, 2], [3, 4]], [5, 2]
```

Methods

- Functions attached to specific objects
- Can use `dir(ob)` to find methods of `ob`
- Dot notation

```
>>> lst = [2, 1, 9, 4]
```

```
>>> lst.sort() # no return value!
```

```
>>> lst
```

```
[1, 2, 4, 9]
```

← Changed “in place”

split and join

```
>>> 'a-b-c-d'.split('-')
```

```
['a', 'b', 'c', 'd']
```

```
>>> '#'.join(_)
```

```
'a#b#c#d'
```

Tuples are not just immutable lists

- As far as available functions and methods go, tuples act like immutable lists
- The “python way of doing things,” however, is more subtle:
 - Use lists when order matters but not position (e.g., an ordered list)
 - Use tuples when position matters (e.g., the 0th item is name, the 1th item is age, etc.)

Dictionaries

```
>>> grades = { }
```

```
>>> grades[ 'A' ] = 90
```

```
>>> grades[ 'B' ] = 80
```

```
>>> grades[ 'B' ]
```

```
80
```

```
>>> grades[ 'A' ]
```

```
90
```

```
>>> grades[ 'C' ]
```

```
[error]
```

Dictionaries continued

```
>>> grades = { 'A' : 90, 'B' : 90 }
```

```
>>> grades[ 'C' ] = 70
```

```
>>> grades
```

```
{ 'B' : 90, 'A' : 90, 'C' : 70 } #UNORDERED!
```

```
>>> del grades[ 'A' ]
```

```
>>> grades
```

```
{ 'B' : 90, 'C' : 70 }
```

Dictionary Methods

```
>>> sound = { 'cow' : 'Moo!', 'pig' : 'Oink!' }
```

```
>>> sound.keys()
```

```
['cow', 'pig']
```

```
>>> sound.values()
```

```
['Moo!', 'Oink!']
```

```
>>> sound.items()
```

```
[('cow', 'Moo!'), ('pig', 'Oink!')]
```

Dictionary Methods Continued

```
>>> sound = { 'cow' : 'Moo!', 'pig' : 'Oink!' }
```

```
>>> sound.has_key( 'dog' )
```

```
False
```

```
>>> sound.has_key( 'Cow' )
```

```
False
```

```
>>> sound.has_key( 'Cow'.lower( ) )
```

```
True
```

The random module

```
>>> import random
>>> lst = range(5)
>>> random.shuffle(lst)
>>> lst
[1, 3, 2, 0, 4]
>>> random.random()
0.25077102516520366
```

Python factoid of the day: Long integers

- In many programming languages, integers are currently stored as 32-bit numbers, but there is often a longer integer that is sometimes 64-bit and sometimes of an arbitrary size.
- Overflow happens when you have too large a number
- As of version 2.2 python unified integers, so the changeover from 32-bit to arbitrary is automatic... they act like the actual integers