Introduction to Python

Christopher Barker

UW Continuing Education / Isilon

July 18, 2012

Table of Contents

Review of Previous Class

- String formatting
- File reading and writing
- Unicode
- Exception Handling
- Path and Directories

Questions?

Homework review

Homework notes

CodingBat

List-1 – sum2

```
def sum2(nums):
    if ( len( nums ) > 1 ):
        return nums[0] + nums[1]
    elif ( nums ):
        return nums[0]
    return 0
```

sum() is handy:

```
def sum2(nums):
    return sum(nums[:2])
```

CodingBat

String-1 -- make_tags

```
def make_tags(tag, word):
    return "<" + tag + ">" + word + "</" + tag + ">"
```

string formatting...

```
def make_tags(tag, word):
    return "<%s>%s</%s>"%(tag,word,tag)
```

```
def make_tags(tag, word):
    return "<%(tag)s>%(word)s</%(tag)s>"% \
                        {'tag':tag, 'word':word}
```

Lightning Talk

Lighting Talk:

Joshua

Default Parameters

Sometimes you don't need the user to specify everything every time

```
In [142]: def fun(x,y,z=5):
    .....:         print x,y,z

In [143]: fun(1,2)
1 2 5

In [144]: fun(1,2,3)
1 2 3
```

Keyword arguments

You can specify only what you need – any order

```
In [151]: def fun(x,y=0,z=0):
              print x,y,z
          .....:

In [152]: fun(1,2,3)
1 2 3

In [153]: fun(1, z=3)
1 0 3

In [154]: fun(1, z=3, y=2)
1 2 3
```

Keyword arguments

You can specify only what you need – any order

```
In [151]: def fun(x,y=0,z=0):
              print x,y,z
          .....:

In [152]: fun(1,2,3)
1 2 3

In [153]: fun(1, z=3)
1 0 3

In [154]: fun(1, z=3, y=2)
1 2 3
```

Keyword arguments

A Common Idiom

```
def fun(x,y=None):
    if y is None:
        do_something_different

    go_on_here
```

Keyword arguments

Can set defaults to variables

```
In [156]: y = 4

In [157]: def fun(x=y):
    ....:     print "x is:", x
    ....:

In [158]: fun()
x is: 4
```

Keyword arguments

Defaults are evaluated when the function is defined

```
In [156]: y = 4

In [157]: def fun(x=y):
    ....:     print "x is:", x
    ....:

In [158]: fun()
x is: 4

In [159]: y = 6

In [160]: fun()
x is: 4
```

Keyword arguments

```
>>> l = []
>>> for i in range(3):
>>>     def fun(x=i):
>>>         print x
>>>     l.append(fun)
>>> l
[<function __main__.fun>, <function __main__.fun>, <functi
>>> l[0]
<function __main__.fun>
>>> l[0]()
0
>>> l[1]()
1
```

lambda

"Anonymous" functions

```
In [171]: f = lambda x, y: x+y

In [172]: f(2,3)
Out[172]: 5
```

Can only be an expression – not a statement

lambda

Can also use keyword arguments

```
In [186]: l = []
In [187]: for i in range(3):
    l.append(lambda x, e=i: x**e)
   .....:
In [189]: for f in l:
    print f(3)
1
3
9
```

LAB

keyword arguments

- Write a function that has four optional parameters (with defaults):
  - foreground_color
  - background_color
  - link_color
  - visited_link_color
- Have it print the colors.
- Call it with a couple different parameters set

Lightning Talk

Lighting Talk:

David

Lists

List Literals

```
>>> []
[]
>>> list()
[]
>>> [1, 2, 3]
[1, 2, 3]
>>> [1, 3.14, "abc"]
[1, 3.14, 'abc']
```

Lists

Indexing just like all sequences

```
>>> food = ['spam', 'eggs', 'ham']
>>> food[2]
'ham'
>>> food[0]
'spam'
>>> food[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Lists

List are mutable

```
>>> food = ['spam', 'eggs', 'ham']
>>> food[1] = 'raspberries'
>>> food
['spam', 'raspberries', 'ham']
```

Lists

Each element is a value, and can be in multiple lists and have multiple names (or no name)

```
>>> name = 'Brian'
 >>> a = [1, 2, name]
 >>> b = [3, 4, name]
 >>> name
 'Brian'
 >>> a
 [1, 2, 'Brian']
 >>> b
 [3, 4, 'Brian']
 >>> a[2]
 'Brian'
 >>> b[2]
 'Brian'
```

Lists

.append(), .insert()

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.append('sushi')
>>> food
['spam', 'eggs', 'ham', 'sushi']
>>> food.insert(0, 'carrots')
>>> food
['carrots', 'spam', 'eggs', 'ham', 'sushi']
```

Lists

.extend()

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.extend(['fish', 'chips'])
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']
```

could be any sequence:

```
>>>  food
>>>  ['spam', 'eggs', 'ham']
>>>  silverware = ('fork', 'knife', 'spoon') # a tuple
>>>  food.extend(silverware)
>>>  food
>>>  ['spam', 'eggs', 'ham', 'fork', 'knife', 'spoon']
```

Lists

pop(), remove()

```
In [203]: food = ['spam', 'eggs', 'ham', 'toast']
In [204]: food.pop()
Out[204]: 'toast'
In [205]: food.pop(0)
Out[205]: 'spam'
In [206]: food
Out[206]: ['eggs', 'ham']
In [207]: food.remove('ham')
In [208]: food
Out[208]: ['eggs']
```

Lists

`list()` accepts any sequence and returns a list of that sequence

```
>>> word = 'Python '
>>> chars = []
>>> for char in word:
...     chars.append(char)
>>> chars
['P', 'y', 't', 'h', 'o', 'n', ' ']
>>> list(word)
['P', 'y', 't', 'h', 'o', 'n', ' ']
```

Lists

If you need to change individual letters... you can do this, but usually `somestring.replace()` will be enough

```
In [216]: name = 'Chris'
In [217]: lname = list(name)
In [218]: lname[0:2] = 'K'
In [219]: name = ''.join(lname)
In [220]: name
Out[220]: 'Kris'
```

Lists

Building up strings:

```
In [221]: msg = []

In [222]: msg.append('The first line of a message')

In [223]: msg.append('The second line of a message')

In [224]: msg.append('And one more line')

In [225]: print '\n'.join(msg)
The first line of a message
The second line of a message
And one more line
```

Slicing

Slicing makes a copy

```
In [227]: food = ['spam', 'eggs', 'ham', 'sushi']

In [228]: some_food = food[1:3]

In [229]: some_food[1] = 'bacon'

In [230]: food
Out[230]: ['spam', 'eggs', 'ham', 'sushi']

In [231]: some_food
Out[231]: ['eggs', 'bacon']
```

Slicing

Easy way to copy a whole list

```
In [232]: food
Out[232]: ['spam', 'eggs', 'ham', 'sushi']

In [233]: food2 = food[:]

In [234]: food is food2
Out[234]: False
```

but the copy is "shallow"

Slicing

"Shallow" copy

```
In [249]: food = ['spam', ['eggs', 'ham']]
In [251]: food_copy = food[:]
In [252]: food[1].pop()
Out[252]: 'ham'
In [253]: food
Out[253]: ['spam', ['eggs']]
In [256]: food.pop(0)
Out[256]: 'spam'
In [257]: food
Out[257]: [['eggs']]
In [258]: food_copy
Out[258]: ['spam', ['eggs']]
```

Name Binding

Assigning to a name does not copy:

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> food_again = food
>>> food_copy = food[:]
>>> food.remove('sushi')
>>> food
['spam', 'eggs', 'ham']
>>> food_again
['spam', 'eggs', 'ham']
>>> food_copy
['spam', 'eggs', 'ham', 'sushi']
```

Iterating

Iterating over a list

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> for x in food:
...     print x
...
spam
eggs
ham
sushi
```

Processing lists

A common pattern

```
filtered = []
for x in somelist:
    if should_be_included(x):
        filtered.append(x)
del(somelist)  # maybe
```

you don't want to be deleting items from the list while iterating...

Mutating Lists

if youre going to change the list, iterate over a copy for safety

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> for x in food[:]:
    ...    # change the list somehow
    ...
```

insidious bugs otherwise

operators vs methods

What's the difference?

```
>>> food = ['spam', 'eggs', 'ham']
>>> more = ['fish', 'chips']
>>> food = food + more
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']

>>> food = ['spam', 'eggs', 'ham']
>>> more = ['fish', 'chips']
>>> food.extend(more)
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']
```

in

```
>>> food = ['spam', 'eggs', 'ham']
>>> 'eggs' in food
True
>>> 'chicken feet' in food
False
```

reverse()

Iterating over a list

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.reverse()
>>> food
['ham', 'eggs', 'spam']
```

sort()

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> food.sort()
>>> food
['eggs', 'ham', 'spam', 'sushi']
```

note:

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> result = food.sort()
>>> print result
None
```

Sorting

How should this sort?

```
>>> s
[[2, 'a'], [1, 'b'], [1, 'c'], [1, 'a'], [2, 'c']]
```

Sorting

How should this sort?

```
>>> s
[[2, 'a'], [1, 'b'], [1, 'c'], [1, 'a'], [2, 'c']]

>>> s.sort()
>>> s
[[1, 'a'], [1, 'b'], [1, 'c'], [2, 'a'], [2, 'c']]
```

Sorting

You can specify your own compare function:

```
In [279]: s = [[2, 'a'], [1, 'b'], [1, 'c'], [1, 'a'], [2,
In [281]: def comp(s1,s2):
    .....:     if s1[1] > s2[1]: return 1
    .....:     elif s1[1]<s2[1]: return -1
    .....:     else:
    .....:         if s1[0] > s2[0]: return 1
    .....:         elif s1[0] < s2[0]: return -1
    .....:     return 0
In [282]: s.sort(comp)
In [283]: s
Out[283]: [[1, 'a'], [2, 'a'], [1, 'b'], [1, 'c'], [2, 'c']
```

Sorting

Mixed types can be sorted.

"objects of different types always compare unequal, and are ordered consistently but arbitrarily."

`http://docs.python.org/reference/expressions.html#not-in`

Searching

Finding or Counting items

```
In [288]: l = [3,1,7,5,4,3]

In [289]: l.index(5)
Out[289]: 3

In [290]: l.count(3)
Out[290]: 2
```

List Performance

- indexing is fast and constant time: O(1)
- x in s proportional to n: O(n)
- visiting all is proportional to n: O(n)
- operating on the end of list is fast and constant time: O(1)
  append(), pop()
- operating on the front (or middle) of the list depends on n: O(n)
  pop(0), insert(0, v)
  But, reversing is fast. Also, collections.deque

`http://wiki.python.org/moin/TimeComplexity`

Lists vs. Tuples

List or Tuples

If it needs to mutable: list

If it needs to be immutable: tuple
(dict key, safety when passing to a function)

Otherwise ... taste and convention

List vs Tuple

Convention:

Lists are Collections (homogeneous):

– contain values of the same type
– simplifies iterating, sorting, etc

tuples are mixed types:

– Group multiple values into one logical thing – Kind of like simple C structs.

List vs Tuple

- Do the same operation to each element?
- Small collection of values which make a single logical item?
- To document that these values won't change?
- Build it iteratively?
- Transform, filter, etc?

List vs Tuple

- Do the same operation to each element? **list**
- Small collection of values which make a single logical item? **tuple**
- To document that these values won't change? **tuple**
- Build it iteratively? **list**
- Transform, filter, etc? **list**

Named Tuple (Collections Module)

```
>>> Point = collections.namedtuple('Point',('x','y'))
>>> p = Point(3.4, 5.2)
>>> p
 Point(x=3.4, y=5.2)
>>> p.x
 3.4
>>> p[1]
 5.2
>>> p = Point(y=2.3, x=3.1)
>>>  p
 Point(x=3.1, y=2.3)
```

Named Tuple (Collections Module)

Named Tuple

handy for database records: sqlite, csv, etc

`http://docs.python.org/library/collections.html#module-collections`

List comprehensions

A bit of functional programming:

```
new_list = [expression for variable in a_list]
```

same as for loop:

```
new_list = []
for variable in a_list:
    new_list.append(expression)
```

List comprehensions

Examples:

```
In [341]: [x**2 for x in range(3)]
Out[341]: [0, 1, 4]

In [342]: [x+y for x in range(3) for y in range(2)]
Out[342]: [0, 1, 1, 2, 2, 3]

In [343]: [x*2 for x in range(6) if not x%2]
Out[343]: [0, 4, 8]
```

List comprehensions

Remember this from last week?

```
[name for name in dir(__builtin__) if "Error" in name]
```

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BufferError',
 'EOFError',
 'EnvironmentError',
```

Generator Expressions

Like a list comprehension, but generates the items on the fly:

```
In [393]: g = ( x**2 for x in [3, 4, 5])

In [394]: g
Out[394]: <generator object <genexpr> at 0x17b0df0>

In [395]: for i in g:
    ....:     print i
    ....:
9
16
25
```

List Docs

The list docs:

`http://docs.python.org/library/stdtypes.html#mutable-sequence-types`

(actually any mutable sequence....)

LAB

Dan's list Lab

Lightning Talk

Lighting Talk:

Rob

Dictionary

Python calls it a `dict`

Other terms:

- dictionary
- associative array
- map
- hash table
- hash
- key-value pair

Dictionary Constructors

```
>>> {'key1': 3, 'key2': 5}
{'key1': 3, 'key2': 5}

>>> dict([('key1', 3),('key2', 5)])
{'key1': 3, 'key2': 5}

>>> dict(key1=3, key2= 5)
{'key1': 3, 'key2': 5}

>>> d = {}
>>> d['key1'] = 3
>>> d['key2'] = 5
>>> d
{'key1': 3, 'key2': 5}
```

Dictionary Indexing

```
>>> d = {'name': 'Brian', 'score': 42}
>>> d['score']
42
>>> d = {1: 'one', 0: 'zero'}
>>> d[0]
'zero'
>>> d['non-existing key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'fish'
```

Dictionary Indexing

Keys can be any immutable:

- numbers
- string
- tuples

```
In [325]: d[3] = 'string'
In [326]: d[3.14] = 'pi'
In [327]: d['pi'] = 3.14
In [328]: d[ (1,2,3) ] = 'a tuple key'
In [329]: d[ [1,2,3] ] = 'a list key'
   TypeError: unhashable type: 'list'
```

Actually – any "hashable" type.

Dictionary Indexing

hash functions convert arbitrarily large data to a small proxy (usually int)

always return the same proxy for the same input

MD5, SHA, etc

Dictionary Indexing

Dictionaries hash the key to an integer proxy and use it to find the key and value

Key lookup is efficient because the hash function leads directly to a bucket with a very few keys (often just one)

Dictionary Indexing

What would happen if the proxy changed after storing a key?

Hashability requires immutability

Dictionary Indexing

Key lookup is very efficient

Same average time regardless of size

also... Python name look-ups are implemented with dict:
— its highly optimized

Dictionary Indexing

key to value
lookup is one way

value to key
requires visiting the whole dict

if you need to check dict values often, create another dict or set (up to you to keep them in sync)

Dictionary Indexing

dictionaries have no defined order

```
In [352]: d = {'one':1, 'two':2, 'three':3}

In [353]: d
Out[353]: {'one': 1, 'three': 3, 'two': 2}

In [354]: d.keys()
Out[354]: ['three', 'two', 'one']
```

dict iterating

`for` iterates the keys

```
>>> d = {'name': 'Brian', 'score': 42}
>>> for x in d:
...     print x
...
score name
```

note the different order...

dict keys and values

```
>>> d.keys()
['score', 'name']

>>> d.values()
[42, 'Brian']

>>> d.items()
[('score', 42), ('name', 'Brian')]
```

dict keys and values

iterating on everything

```
>>> d = {'name': 'Brian', 'score': 42}
>>> for k, v in d.items():
...     print "%s: %s" % (k, v)
...
score: 42
name: Brian
```

Dictionary Performance

- indexing is fast and constant time: O(1)
- x in s cpnstant time: O(1)
- visiting all is proportional to n: O(n)
- inserting is constant time: O(1)
- deleting is constant time: O(1)

`http://wiki.python.org/moin/TimeComplexity`

Dict Comprehensions

You can do it with dicts, too:

```
new_dict = { key:value for variable in a_sequence}
```

same as for loop:

```
new_dict = {}
for key in a_list:
    new_dict[key] = value
```

Dict Comprehensions

Example

```
In [340]: { i: "this_%i"%i for i in range(5) }
Out[340]: {0: 'this_0', 1: 'this_1', 2: 'this_2',
           3: 'this_3', 4: 'this_4'}
```

(not as useful with the dict() constructor...)

Switch ?

How do you spell switch/case in Python?

Put the values to switch on in the keys:

Functions to call in values:

demo: sample code (`switch_case.py`)

Sets

`set` is an unordered collection of distinct values

Essentially a dict with only keys

Set Constructors

```
>>> set()
set([])
>>> set([1, 2, 3])
set([1, 2, 3])
# as of 2.7
>>> {1, 2, 3}
set([1, 2, 3])
>>> s = set()
>>> s.update([1, 2, 3])
>>> s
set([1, 2, 3])
```

Set Properties

Set members must be hashable

Like dictionary keys – and for same reason (efficient lookup)

No indexing (unordered)

```
>>> s[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Set Methods

```
>> s = set([1])
>>> s.pop() # an arbitrary member
1
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'

>>> s = set([1, 2, 3])
>>> s.remove(2)
>>> s.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

Set Methods

```
s.isdisjoint(other)

s.issubset(other)

s.union(other, ...)

s.intersection(other, ...)

s.difference(other, ...)

s.symmetric_difference( other, ...)
```

Frozen Set

Also `frozenset`

immutable – for use as a key in a dict
(or another set...)

```
>>> fs = frozenset((3,8,5))
>>> fs.add(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Function arguments in variables

function arguments are really just
– a tuple (positional arguments)
– a dict (keyword arguments)

```
def f(x, y, w=0, h=0):
    print "position: %s, %s -- shape: %s, %s"%(x, y, w, h)

position = (3,4)
size = {'h': 10, 'w': 20}

>>> f( *position, **size)
position: 3, 4 -- shape: 20, 10
```

Function parameters in variables

You can also pull in the parameters out in the function as a tuple and a dict

```
def f(*args, **kwargs):
    print "the positional arguments are:", args
    print "the keyword arguments are:", kwargs

In [389]: f(2, 3, this=5, that=7)
the positional arguments are: (2, 3)
the keyword arguments are: {'this': 5, 'that': 7}
```

LAB

Dan's dict LAB

or
Optional LAB

- Coding Kata 14 - Dave Thomas
  `http://codekata.pragprog.com/2007/01/kata_fourteen_t.html`
- See how far you can get on this task using The Adventures of Sherlock Holmes as input: sherlock.txt in the week04 directory (ascii)
- This is intentionally open-ended and underspecified. There are many interesting decisions to make.

Homework

- Spend more time (or some time) with the Coding Kata from lab. Get it basically working.
- Experiment with different lengths for the lookup key. (3 words, 4 words, 3 letters, etc)
- This assignment is about playing around with the algorithm and data.