Introduction to Python Topics

Christopher Barker

UW Continuing Education

October 15, 2013

Table of Contents

Review of Previous Class

- .
- .
- .

Lightning Talks

Lightning talks today:

Homework review

Homework Questions?

My Solution

topic

Some Stuff

```
sample code
```

Sequences

Sequences are ordered collections of objects

They can be indexed, sliced, iterated over,...

They have a length: `len(sequence)`

Common sequences (Remember Duck Typing?):

- strings
- tuples
- lists

Indexing

square brackets for indexing: []

Indexing starts at zero

```
In [98]: s = "this is a string"

In [99]: s[0]
Out[99]: 't'

In [100]: s[5]
Out[100]: 'i'
```

Indexing

Negative indexes count from the end

```
In [105]: s = "this is a string"

In [106]: s[-1]
Out[106]: 'g'

In [107]: s[-6]
Out[107]: 's'
```

Slices

Slicing: Pulling a range out of a sequence

```
sequence[start:finish]
```

```
indexes for which:
```

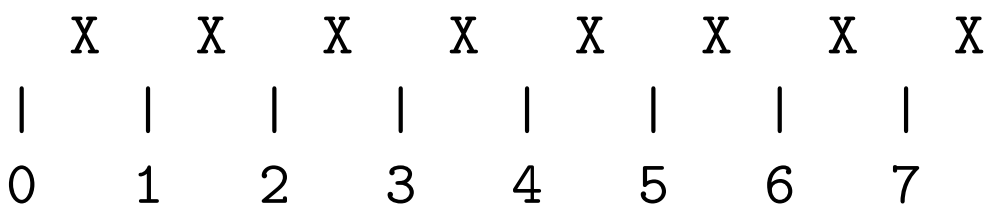
```
start <= i < finish
```

Slices

```
In [121]: s = "a bunch of words"
In [122]: s[2]
Out[122]: 'b'

In [123]: s[6]
Out[123]: 'h'

In [124]: s[2:6]
Out[124]: 'bunc'

In [125]: s[2:7]
Out[125]: 'bunch'
```

Slices

the indexes point to the spaces between the items



Slices

Slicing satisfies nifty properties:

```
len( seq[a:b] ) == b - a
```

```
seq[a:b] + seq[b:c] == seq
```

Slicing vs. Indexing

Indexing returns a single element

```
In [86]: l
Out[86]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [87]: type(l)
Out[87]: list

In [88]: l[3]
Out[88]: 3

In [89]: type( l[3] )
Out[89]: int
```

Slicing vs. Indexing

Unless it's a string:

```
In [75]: s = "a string"

In [76]: s[3]
Out[76]: 't'

In [77]: type(s[3])
Out[77]: str
```

There is no single character type

Slicing vs. Indexing

Slicing returns a sequence:

```
In [68]: l
Out[68]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [69]: l[2:4]
Out[69]: [2, 3]
```

Even if it's one element long

```
In [70]: l[2:3]
Out[70]: [2]

In [71]: type(l[2:3])
Out[71]: list
```

Slicing vs. Indexing

Indexing out of range produces an error

```
In [129]: s = "a bunch of words"
In [130]: s[17]
----> 1 s[17]
IndexError: string index out of range
```

Slicing just gives you what's there

```
In [131]: s[10:20]
Out[131]: ' words'

In [132]: s[20:30]
Out[132]: ''
```

(demo)

Multiplying and slicing

from CodingBat: Warmup-1 – front3

```
def front3(str):
  if len(str) < 3:
    return str+str+str
  else:
    return str[:3]+str[:3]+str[:3]
```

or

```
def front3(str):
    return str[:3] * 3
```

Slicing

from CodingBat: Warmup-1 – `missing_char`

```
def missing_char(str, n):
  front = str[0:n]
  l = len(str)-1
  back = str[n+1:l+1]
  return front + back

def missing_char(str, n):
    return str[:n] + str[n+1:]
```

Slicing

you can skip items, too

```
In [289]: string = "a fairly long string"

In [290]: string[0:15]
Out[290]: 'a fairly long s'

In [291]: string[0:15:2]
Out[291]: 'afil ogs'

In [292]: string[0:15:3]
Out[292]: 'aallg'
```

Command Line Input

`input` evaluates the input:

```
In [265]: val = input("a message> ")
a message> 4.5
In [266]: type(val)
Out[266]: float
```

`raw_input` gives you the plain string:

```
In [265]: val = input("a message> ")
a message> 4.5
In [266]: type(val)
Out[266]: float
```

(demo)

LAB

`def count_them(letter):`

- prompts the user to input a letter
- counts the number of times the given letter is input
- prompts the user for another letter
- continues until the user inputs "x"
- returns the count of the letter input

`def count_letter_in_string(string, letter):`

- counts the number of instances of the letter in the string
- ends when a period is encountered
- if no period is encountered – prints "hey, there was no period!"

LAB

Write some functions that:

- return a string with the first and last characters exchanged.
- return a string with every other character removed
- return a string with the first and last 4 characters removed, and every other char in between
- return a string reversed (just with slicing)
- return a string with the middle, then last, then first third in a new order

Lists

Lists Literals

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

List Indexing

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
```

List Mutability

Lists are mutable

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

List Elements

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]
```

List Methods

`.append()`, `.insert()`

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

List Methods

.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']
```

List Methods

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']
```

List Constructor

`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', ' ']
```

String to List to String

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'
```

Building up strings in a list

```
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
```

List 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']
```

List 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":

`http://docs.python.org/library/copy.html`

List 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']
```

List 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 you're 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']
```

(the operator makes a new list...)

in

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

reverse()

```
>>> 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**

List Docs

The list docs:

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

(actually any mutable sequence....)

tuples and commas..

Tuples don't NEED parentheses...

```
In [161]: t = (1,2,3)
In [162]: t
Out[162]: (1, 2, 3)

In [163]: t = 1,2,3
In [164]: t
Out[164]: (1, 2, 3)

In [165]: type(t)
Out[165]: tuple
```

tuples and commas..

Tuples do need commas...

```
In [156]: t = ( 3 )

In [157]: type(t)
Out[157]: int

In [158]: t = (3,)
In [159]: t
Out[159]: (3,)

In [160]: type(t)
Out[160]: tuple
```

LAB

List Lab

`week-03/code/list_lab.rst`

for loops

looping through sequences

```
for x in sequence:
    do_something_with_x
```

for loops

```
In [170]: for x in "a string":
    .....:         print x
    .....:
a

s
t
r
i
n
g
```

range

looping a known number of times..

```
In [171]: for i in range(5):
    .....:     print i
    .....:
0
1
2
3
4
```

(you don't need to do anything with i...

range

`range` defined similarly to indexing

```
In [183]: range(4)
Out[183]: [0, 1, 2, 3]

In [184]: range(2,4)
Out[184]: [2, 3]

In [185]: range(2,10,2)
Out[185]: [2, 4, 6, 8]
```

indexing?

Python only loops through a sequence – not like C, Javascript, etc...

```
for(var i=0; i<arr.length; i++) {
    var value = arr[i];
    alert(i =") "+value);
}
```

indexing?

Use range?

```
In [193]: letters = "Python"

In [194]: for i in range(len(letters)):
   .....:     print letters[i]
   .....:
P
y
t
h
o
n
```

indexing?

More Pythonic – for loops through sequences

```
In [196]: for l in letters:
   .....:     print l
   .....:
P

y

t

h

o

n
```

Never index in normal cases

enumerate

If you need an index – enumerate

```
In [197]: for i, l in enumerate(letters):
   .....:     print i, l
   .....:
0 P
1 y
2 t
3 h
4 o
5 n
```

multiple sequences – zip

If you need to loop though parallel sequences – `zip`

```
In [200]: first_names = ['Fred', 'Mary', 'Jane']

In [201]: last_names = ['Baker', 'Jones', 'Miller']

In [203]: for first, last in zip(first_names, last_names):
   .....:     print first, last
   .....:
Fred Baker
Mary Jones
Jane Miller
```

xrange

`range` creates the whole list

`xrange` is a generator – creates it as it's needed –

a good idea for large numbers

```
In [207]: for i in xrange(3):
   .....:     print i
0
1
2
```

(Python 3 – `range == xrange`)

for

`for` does NOT create a name space:

```
In [172]: x = 10

In [173]: for x in range(3):
    .....:     pass
    .....:

In [174]: x
Out[174]: 2
```

while

`while` is for when you don't know how many loops you need

Continues to execute the body until condition is not `True`

```
while a_condition:
    some_code
    in_the_body
```

while

`while` is more general than `for` – you can always express for as while, but not always vice-versa.

`while` is more error-prone – requires some care to terminate

loop body must make progress, so condition can become `False`

potential error: infinite loops

while vs. for

```
letters = 'Python'
i=0
while i < len(letters):
    print letters[i]
    i += 1
```

vs.

```
letters = 'Python'
for c in letters:
    print c
```

while

Shortcut: recall – 0 or empty sequence is `False`

break

break ends a loop early

```
x = 0
while True:
    print x
    if x > 3:
        break
    x = x + 1
In [216]: run for_while.py
0
1
2
3
4
```

This is a pretty common idiom

break

same way with a `for` loop

```
name = "Chris Barker"
for c in name:
    print c,
    if c == "B":
        break
print "I'm done"
```

```
C h r i s   B
I'm done
```

continue

`continue` skips to the start of the loop again

```
print "continue in a for loop"
name = "Chris Barker"
for c in name:
    if c == "B":
        continue
    print c,
print "\nI'm done"
```

```
continue in a for loop
C h r i s   a r k e r
I'm done
```

continue

`continue` works for a `while` loop too.

```
print "continue in a while loop"
x = 6
while x > 0:
    x = x-1
    if x%2:
        continue
    print x,
print "\nI'm done"
```

```
continue in a while loop
4 2 0
I'm done
```

else again

`else` block run if the loop finished naturally – no `break`

```
print "else in a for loop"
x = 5
for i in range(5):
    print i
    if i == x:
        break
else:
    print "else block run"
```

LAB

Some lab excercises

Lightning Talk

Lightning Talks:

person 1

person 2

Homework

Recommended Reading:

- some stuff

Do:

- Some things

Homework

Recommended Reading:

- Read Think Python: 9, 14
- extra: string methods: `http://docs.python.org/library/stdtypes.html#string-methods`
- extra: unicode: `http://www.joelonsoftware.com/articles/Unicode.html`

Do:

- Six more CodingBat exercises.
- LPTHW: for extra practice with the concepts – some of:

  strings: ex5, ex6, ex7, ex8, ex9, ex10

  raw_input(), sys.argv: ex12, ex13, ex14 (needed for files)

  files: ex15, ex16, ex17