Python collections

Reuven M. Lerner, PhD
reuven@lerner.co.il

1

Lists

- Use square brackets — [ ]   (and not { })
- Can contain any data types, including lists
- A list may contain different types

List examples

```
mylist = [ ]    # empty list

mylist = [1,2,3]

mylist = ['a', 'b', 'c']

mylist = [1, 'a', 2, 'b']

biglist = [mylist, mylist, mylist]
```

3

Checking membership

```
mylist = [1,2,3]

if 4 in mylist:

  print 'I found 4!'

else:

  print '4 is not in the list!'
```

Slicing lists

First element	`mylist[0]`
Second element	`mylist[1]`
Final element	`mylist[-1]`
First 5 elements	`mylist[0:5] or mylist[:5]`
Final 5 elements	`mylist[-5:]`

List methods

Locate “a”	`mylist.index('a')`
Add one item	`mylist.append('zzz')`
Add one item	`mylist.append([1,2,3])`
Add all items	`mylist.extend([1,2,3])`
Insert item (pushing existing ones aside)	`mylist.insert(5, 'zzz')`

6

More list methods

How many as?	`mylist.count('a')`
How many items?	`len(mylist)`
Remove from the end	`mylist.pop()`
Remove from the front	`mylist.pop(0)`

7

Stacks and queues

- List as a stack:

```
mylist.append('z')

mylist.pop()
```

- List as a queue:

```
mylist.append('z')

mylist.pop(0)
```

Replacing elements

```
mylist[0] = 'a'

mylist[0] = [1,2,3]
```

9

Replacing slices

```
mylist = ['a', 'b', 'c', 'd', 'e']
mylist[2:4]
  ['c', 'd']

mylist[2:4] = [1,2,3,4,5]
mylist
  ['a', 'b', 1, 2, 3, 4, 5, 'e']
```

10

Adding, multiplying

```
[1,2,3] + [4,5,6]

    [1, 2, 3, 4, 5, 6]

[1,2,3] * 2

    [1, 2, 3, 1, 2, 3]

[1] * 2 + [2] * 3

    [1, 1, 2, 2, 2]
```

11

Sorting and reversing

```
mylist.sort()

mylist.sort(reverse=True)


mylist.reverse()
```

Remember range?

- Create a list with the range operator:

```
range(5)          # Same as [0,1,2,3,4]

range(10, 20)     # Same as [10, 11, 12 ... 19]

range(10,20,2)    # Same as [10,12,14,16,18]
```

- Create a “lazy” range with xrange()

Mutable vs. immutable

- Mutability is important in Python
- Most data types are mutable
  - These are easiest to understand
- Mutable types cannot be used everywhere
  - (e.g., dictionary keys)

14

Mutability

- Remember: Data can be immutable, but variables can always be reassigned
- Example: Numbers are immutable!

```
a = 5

a = 6
```

- a changed, but its value did not

What is assignment?

- In Python, the = sign means, “assign the value on the right to the name on the left”
- It doesn’t affect the object to which the name previously pointed!

Assignment

- Assign values with =
- Variables refer to values

```
a = 5
b = a
a = 7
print b
```

```
a = [1,2,3]
b = a
a.append(4)
print b
```

split

- The "split" method returns a list from a string:

```
>>> s = 'a,b,c'
>>> s.split(',')
   ['a', 'b', 'c']
>>> s.split('b')
   ['a,', ',c']
```

Parameters to split

- Parameter is a string, not a character or regexp!
- Be careful of multiple, adjacent occurrences

```
>>> s = 'abc def  ghi jkl' # Notice 'f  g'
>>> s.split(' ')
['abc', 'def', '', 'ghi', 'jkl']
```

19

Split on all whitespace

- Don't pass any parameter to str.split(), and it'll use any combination of whitespace:

```
>>> s = 'abc def  ghi jkl'

>>> s.split()

['abc', 'def', 'ghi', 'jkl']

>>> s = 'abc\tdef  ghi\tjkl'

>>> s.split()

['abc', 'def', 'ghi', 'jkl']
```

20

join

- join is a string method (not a list method)
- Pass it any iterable (i.e., sequence) of strings

```
>>> ','.join(['abc', 'def', 'ghi'])
'abc,def,ghi'
>>> '**'.join(['abc', 'def', 'ghi'])
'abc**def**ghi'
>>> '**'.join('abc')
'a**b**c'
```

Loop on list

- You can loop on a list, just like on a string
- Elements of the list are assigned to the variable

```
for item in ['abc', 'def', 'ghi']:
  print item
```

Adding strings

```
rows = [['abc', 'def', 'ghi'],
        ['jkl', 'mno', 'qrs']]
output = ''
for row in rows:
    output += '\t'.join(row) + '\n'
print output
```

Better, use join

```
rows = [['abc', 'def', 'ghi'],
        ['jkl', 'mno', 'qrs']]
output = [ ]
for row in rows:
    output.append('\t'.join(row))
print '\n'.join(output)
```

24

Tuples

- Like lists, but immutable
- Why do they exist?
    - Faster, immutable (useful as keys)
    - Honestly, I don’t use them that much
- Don’t forget the comma!
    - `t = (1) vs. t = (1,)`

25

Working with tuples

- Create with parentheses ( )
- Access with [ ]

```
t = (1,2,3)

t[2]            # Item at index 2

t.count(2)      # Number of 2 values

t.index(2)      # First index of value 2
```

26

Immutable!

```
t = (1,2,3)

t[2] = 5      # Error: No assignment

t = t[1:]     # OK, not changing data

t.sort()      # Does not exist

t.reverse() # Does not exist
```

27

Tuple are immutable; their contents might not be

```
>>> t = (['a', 'b', 'c'], ['d', 'e' 'f'])

>>> t[0] = 'abc'

    TypeError: 'tuple' object does not support item assignment

>>> t[0][0] = '!!!'

>>> t

    (['!!!', 'b', 'c'], ['d', 'ef'])
```

28

Weird errors!

```
>>> t
    (['!!!', 'b', 'c'], ['d', 'ef'])
>>> t[0] += ['Z']
TypeError: 'tuple' object does not support item assignment
>>> t
 (['!!!', 'b', 'c', 'Z'], ['d', 'ef'])
```

29

Lists to tuples (and back)

```
mylist = [1,2,3]

t = (1,2,3)

tuple(mylist)   # (1,2,3)

list(t)         # [1,2,3]
```

Sequences

- Strings, lists, and tuples are all “sequences”
- Many things work on sequences
- For example, in:

```
>>> mylist = [1,2,3,4,5]
>>> 1 in mylist
True
>>> 10 in mylist
False
```

Slicing sequences

First element	`seq[0]`
Second element	`seq[1]`
Final element	`seq[-1]`
First 5 elements	`seq[0:5] or seq[:5]`
Final 5 elements	`seq[-5:]`

32

Slices

- Remember that the result of a slice is a new object of the same type

```
>>> mylist = ['a', 'b', 'c']

>>> mylist[0]

'a'

>>> mylist[:1]

['a']
```

33

Slice objects

- You can even create a slice object, and reuse it:

```
>>> s= slice(3,20,3)

>>> alphabet = 'abcdefghijklmnopqrstuvwxyz'

>>> alphabet[s]  # same as alphabet[3:20:3]

'dgjmps'
```

34

Sequence functions

True if any element is True	`any(seq)`
True if all elements are True	`all(seq)`
Smallest element	`min(seq)`
Largest element	`max(seq)`