List Comprehensions, Lambdas and Sorting Multidimensional Structures

Advanced Container Processing

This week we will complete our tour of the core Python data processing features.

So far we have explored the reading and parsing of data; the loading of data into built-in structures; and the aggregation and sorting of these structures.

This section explores advanced tools for container processing.

list comprehensions

a = ['hello', 'there', 'harry']
print [var.upper() for var in a if var.startswith('h')]
['HELLO', 'HARRY']

lambda functions

```
names = ['Joe Wilson', 'Pete Johnson', 'Mary Rowe']
sorted_names = sorted(names, key=lambda x: x.split()[1])
print sorted_names             # ['Pete Johnson', 'Mary Rowe', 'Joe Wilso
```

ternary assignment

```
rev_sort = True if user_input == 'highest' else False

pos_val = x if x >= 0 else x * -1
```

conditional assignment

```
val = this or that       # 'this' if this is True else 'that'
val = this and that      # 'this' if this is False else 'that'
```

List comprehensions: filtering a container's elements

List comprehensions abbreviate simple loops into one line.

Consider this loop, which filters a list so that it contains only positive integer values:

```
myints = [0, -1, -5, 7, -33, 18, 19, 55, -100]
myposints = []
for el in myints:
  if el > 0:
    myposints.append(el)

print myposints                    # [7, 18, 19, 55]
```

This loop can be replaced with the following one-liner:

```
myposints = [ el for el in myints if el > 0 ]
```

See how the looping and test in the first loop are distilled into the one line? The first **el** is the element that will be added to **myposints** - list comprehensions automatically build new lists and return them when the looping is done.

The operation is the same, but the order of operations in the syntax is different:

```
# this is pseudo code
# target list = item for item in source list if test
```

Hmm, this makes a list comprehension less intuitive than a loop. However, once you learn how to read them, list comprehensions can actually be easier and quicker to read - primarily because they are on one line.

This is an example of a *filtering* list comprehension - it allows some, but not all, elements through to the new list.

List comprehensions: transforming a container's elements

Consider this loop, which doubles the value of each value in it:

```
nums = [1, 2, 3, 4, 5]
dblnums = []
for val in nums:
  dblnums.append(val*2)

print dblnums                          # [2, 4, 6, 8, 10]
```

This loop can be distilled into a list comprehension thusly:

```
dblnums = [ val * 2 for val in nums ]
```

This *transforming* list comprehension transforms each value in the source list before sending it to the target list:

```
# this is pseudo code
# target list = item transform for item in source list
```

We can of course combine filtering and transforming:

```
vals = [0, -1, -5, 7, -33, 18, 19, 55, -100]
doubled_pos_vals = [ i*2 for i in vals if i > 0 ]
print doubled_pos_vals                # [14, 36, 38, 110]
```

List comprehensions: examples

If they only replace simple loops that we already know how to do, why do we need list comprehensions? As mentioned, once you are comfortable with them, list comprehensions are much easier to read and comprehend than traditional loops. They say in one statement what loops need several statements to say - and reading multiple lines certainly takes more time and focus to understand.

Some common operations can also be accomplished in a single line. In this example, we produce a list of lines from a file, stripped of whitespace:

```
stripped_lines = [ i.rstrip() for i in open('FF_daily.txt').readlines() ]
```

Here, we're only interested in lines of a file that begin with the desired year (1972):

```
totals = [ i for i in open('FF_daily.txt').readlines() if i.startswith('1
```

If we want the MktRF values for our desired year, we could gather the bare amounts this way:

```
mktrf_vals = [ float(i.split()[1]) for i in open('FF_daily.txt').readline
```

And in fact we can do part of an earlier assignment in one line -- the sum of MktRF values for a year:

```
mktrf_sum = sum([ float(i.split()[1]) for i in open('FF_daily.txt').readl
```

From experience I can tell you that familiarity with these forms make it very easy to construct and also to decode them very quickly - much more quickly than a 4-6 line loop.

List Comprehensions with Dictionaries

Remember that dictionaries can be expressed as a list of 2-element tuples, converted using **items()**. Such a list of 2-element tuples can be converted back to a dictionary with **dict()**:

```
mydict =  {'a': 5, 'b': 0, 'c': -3, 'd': 2, 'e': 1, 'f': 4}

my_items = mydict.items()      # my_items is now [('a',5), ('b',0), ('c',
mydict2 = dict(my_items)       # mydict2 is now   {'a':5,   'b':0,   'c':
```

It becomes very easy to filter or transform a dictionary using this structure. Here, we're filtering a dictionary by value - accepting only those pairs whose value is larger than 0:

```
mydict = {'a': 5, 'b': 0, 'c': -3, 'd': 2, 'e': -22, 'f': 4}
filtered_dict = dict([ (i, j) for (i, j) in mydict.items() if j > 0 ])
```

Here we're switching the keys and values in a dictionary, and assigning the resulting dict back to **mydict**, thus seeming to change it in-place:

```
mydict = dict([ (j, i) for (i, j) in mydict.items() ])
```

The Python database module returns database results as tuples. Here we're pulling two of three values returned from each row and folding them into a dictionary.

```
# 'tuple_db_results' simulates what a database returns
tuple_db_results = [
  ('joe', 22, 'clerk'),
  ('pete', 34, 'salesman'),
  ('mary', 25, 'manager'),
]

names_jobs = dict([ (name, role) for name, age, role in tuple_db_results
```

range() and enumerate()

The **range()** function produces a *new list of consecutive integers* which can be used for counting:

```
intlist = range(10)

for val in intlist:
    print val,            # comma:  keep print output on one line
                          # 0 1 2 3 4 5 6 7 8 9

for val in range(5,10):
    print val,            # 5 6 7 8 9

print sum(range(100))     # sum up the values from 0 to 99
```

enumerate() takes any *iterable* (thing that can be looped over) and "marries" it to a range of integers, so you can keep a simultaneous count of something

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

for count, element in enumerate(mylist):
    print "element {}:  {}".format(count, element)
```

This can be handy for example with a filehandle. Since we can loop over a file, we can also pass it to **enumerate()**, which would render a line number with each line:

```
fh = open('file.txt')

for count, line in enumerate(fh):
    print "line {}:  {}".format(count, line.rstrip())

    ## (stripping the line above for clean-looking output)
```

Sorting Multidimensional Structures

Having built multidimensional structures in various configurations, we should now learn how to sort them -- for example, to sort the keys in a dictionary of dictionaries by one of the values in the inner dictionary (in this instance, the last name):

```
def by_last_name(key):
    return dod[key]['lname']

dod = {
         'db13':  {
                     'fname': 'Joe',
                     'lname': 'Wilson',
                     'tel':   '9172399895'
                  },
         'mm23':  {
                     'fname': 'Mary',
                     'lname': 'Doodle',
                     'tel':   '2122382923'
                  }
       }

sorted_keys = sorted(dod, key=by_last_name)
print sorted_keys                              # ['mm23', 'db13']
```

The trick here will be to put together what we know about obtaining the value from an inner structure with what we have learned about custom sorting.

Sorting review

A quick review of sorting: recall how Python will perform a default sort (numeric or *ASCII*-betical) depending on the objects sorted. If we wish to modify this behavior, we can pass each element to a function named by the **key=** parameter:

```
mylist = ['Alpha', 'Gamma', 'episilon', 'beta', 'Delta']

print sorted(mylist)                        # ASCIIbetical sort
                                            # ['Alpha', 'Gamma', 'Delta', '

mylist.sort()                               # sort mylist in-place

print sorted(mylist, key=str.lower)         # alphabetical sort
                                            # (lowercasing each item by tel
                                            # to str.lower)
                                            # ['Alpha', 'beta', 'Delta', 'e

print sorted(mylist, key=len)               # sort by length
                                            # ['beta', 'Alpha', 'Gamma', 'D
```

Sorting review: sorting dictionary keys by value: dict.get

When we loop through a dict, we can loop through a list of *keys* (and use the keys to get values) or loop through *items*, a list of (key, value) tuple pairs. When sorting a dictionary by the values in it, we can also choose to sort **keys** or **items**.

To sort keys, **mydict.get** is called with each key - and **get** returns the associated value. So the keys of the dictionary are sorted by their values.

```
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
mydict_sorted_keys = sorted(mydict, key=mydict.get)
for i in mydict_sorted_keys:
    print "{0} = {1}".format(i, mydict[i])

                ## z = 0
                ## c = 1
                ## b = 2
                ## a = 5
```

Sorting dictionary items by value: operator.itemgetter

Recall that we can render a dictionary as a list of tuples with the **dict.items()** method:

```
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
mydict_items = mydict.items()                          # [(a, 5), (c, 1), (
```

To sort dictionary **items** by value, we need to sort each two-element tuple by its second element. The built-in module **operator.itemgetter** will return whatever element of a sequence we wish - in this way it is like a subscript, but in function format (so it can be called by the Python sorting algorithm).

```
import operator
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
mydict_items = mydict.items()                          # [(a, 5), (c, 1), (
mydict_items.sort(key=operator.itemgetter(1))
print mydict_items                                     # [(z, 0), (c, 1), (
for key, val in mydict_items:
    print "{0} = {1}".format(key, val)

                    ## z = 0
                    ## c = 1
                    ## b = 2
                    ## a = 5
```

The above can be conveniently combined with looping, effectively allowing us to loop through a "sorted" dict:

```
for key, val in sorted(mydict.items(), key=operator.itemgetter(1)):
    print "{0} = {1}".format(key, val)
```

Database results come as a list of tuples. Perhaps we want our results sorted in different ways, so we can store as a list of tuples and sort using **operator.itemgetter**. This example sorts by the third field, then by the second field (last name, then first name):

```
import operator
items =[ (123, 'Joe', 'Wilson', 35, 'mechanic'),
         (124, 'Sam', 'Jones', 22, 'mechanic'),
         (125, 'Pete', 'Jones', 40, 'mechanic'),
         (126, 'Irina', 'Bibi', 31, 'mechanic'),
       ]
items.sort(key=operator.itemgetter(2,1)) # sorts by last, first name
for this_pair in items:
  print "{0} {1}".format(this_pair[1], this_pair[2])

       ## Irina Bibi
       ## Pete Jones
       ## Sam Jones
       ## Joe Wilson
```

Multi-dimensional structures: sorting with custom function

Similar to **itemgetter**, we may want to sort a complex structure by some inner value - in the case of **itemgetter** we sorted a whole tuple by its third value. If we have a list of dicts to sort, we can use the custom sub to specify the sort value from inside each dict:

```
def by_dict_lname(this_dict):
  return this_dict['lname'].lower()

list_of_dicts = [
  { 'id': 123,
    'fname': 'Joe',
    'lname': 'Wilson',
  },
  { 'id': 124,
    'fname': 'Sam',
    'lname': 'Jones',
  },
  { 'id': 125,
    'fname': 'Pete',
    'lname': 'abbott',
  },
]
list_of_dicts.sort(key=by_dict_lname)      # custom sort function (above)
for this_dict in list_of_dicts:
  print "{0} {1}".format(this_dict['fname'], this_dict['lname'])

# Pete abbot
# Sam Jones
# Joe Wilson
```

So, although we are sorting dicts, our sub says "take this dictionary and sort by this inner element of the dictionary".

Multi-dimensional structures: sorting with lambda custom function

Functions are useful but they require that we declare them separately, elsewhere in our code. A *lambda* is a function in a single statement, and can be placed in data structures or passed as arguments in function calls. The advantage here is that our function is used exactly where it is defined, and we don't have to maintain separate statements.

A common use of lambda is in sorting. The format for lambdas is lambda **arg**:

return_val. Compare each pair of regular function and lambda, and note the argument and return val in each.

```
def by_lastname(name):
  fname, lname = name.split()
  return lname

names = [ 'Josh Peschko', 'Gabriel Feghali', 'Billy Woods', 'Arthur Fisch
sortednames = sorted(names, key=lambda name:  name.split()[1])


list_of_dicts = [
  { 'id': 123,
    'fname': 'Joe',
    'lname': 'Wilson',
  },
  { 'id': 124,
    'fname': 'Sam',
    'lname': 'Jones',
  },
  { 'id': 125,
    'fname': 'Pete',
    'lname': 'abbott',
  },
]

def by_dict_lname(this_dict):
  return this_dict['lname'].lower()

sortedlenstrs = sorted(list_of_dicts, key=lambda this_dict:  this_dict['l
```

In each, the label after **lambda** is the argument, and the expression that follows the colon is the return value. So in the first example, the lambda argument is **name**, and the lambda returns **name.split()[1]**. See how it behaves exactly like the regular function itself?

Again, what is the advantage of lambdas? They allow us to design our own functions which can be placed inline, where a named function would go. This is a convenience, not a necessity. **But** they are in common use, so they must be understood by any serious programmer.

Lambda expressions: breaking them down

Many people have complained that lambdas are hard to grok (absorb), but they're really very simple - they're just so short they're hard to read. Compare these two functions, both of which add/concatenate their arguments:

```
def addthese(x, y):
  return x + y

addthese2 = lambda x, y:  x + y

print addthese(5, 9)        # 14
print addthese2(5, 9)       # 14
```

The function definition and the lambda statement are equivalent - they both produce a function with the same functionality.

Lambda expression example: dict.get and operator.itemgetter

Here are our standard methods to sort a dictionary:

```
import operator
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
for key, val in sorted(mydict.items(), key=operator.itemgetter(1)):
    print "{0} = {1}".format(key, val)

for key in sorted(mydict, key=mydict.get):
    print "{0} = {1}".format(key, mydict[key])
```

Imagine we didn't have access to **dict.get** and **operator.itemgetter**. What could we do?

```
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
for key, val in sorted(mydict.items(), key=lambda keyval:  keyval[1]):
    print "{0} = {1}".format(key, val)

for key in sorted(mydict, key=lambda key:  mydict[key]):
    print "{0} = {1}".format(key, mydict[key])
```

These lambdas do exactly what their built-in counterparts do:

in the case of **operator.itemgetter**, take a 2-element tuple as an argument and return the 2nd element

in the case of **dict.get**, take a key and return the associated value from the dict