Map, Filter, Lambda, and List Comprehensions in Py

Author: R.G. Erdmann

`map()`, `filter()`, `lambda`, and list comprehensions provide compact, elegant, and efficient ways to enc programming. We often encounter the following scanarios involving for-loops:

Some for-loop examples to rewrite more compactly

- Building up a list from scratch by looping over a sequence and performing some calculation on e

1. For example, suppose we want to build a list of the squares of the integers from 0 to 9:

```
>>> squares = [] # start with an empty list
>>> for x in range(10): # step over every element in the list of integers from 0 to 9
...     squares.append(x**2) # grow the list one element at a time
...
>>> print squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2. Suppose we want to build a list of the lengths of the names in a list:

```
>>> names = ['Anne', 'Amy', 'Bob', 'David', 'Carrie', 'Barbara', 'Zach']
>>> lengths = []
>>> for name in names:
...     lengths.append(len(name))
...
>>> print lengths
[4, 3, 3, 5, 6, 7, 4]
```

- Building up a list from scratch by looping over nested sequences.

3. For example, suppose we want a list of all possible pairs of drink and food from the lists `['ham', 'eggs', 'spam']`, respectively:

```
>>> possible_choices = []
>>> for drink in ['water', 'tea', 'juice']:
...     for food in ['ham', 'eggs', 'spam']:
...         possible_choices.append([drink,food])
...
>>> print possible_choices
[['water', 'ham'], ['water', 'eggs'], ['water', 'spam'], ['tea', 'ham'], ['tea', 'eggs'],
```

4. Suppose we want a list of coordinates on a rectangular grid:

```
>>> coords = []
>>> for x in range(5):
...     for y in range(3):
...         coordinate = (x, y)
...         coords.append(coordinate)
...
>>> print coords
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1),
```

- Building a list from scratch by filtering a sequence according to some criterion or criteria.

5. For example, suppose we want a list of the squares of the integers from 0 to 9 where the less than 50:

```
>>> special_squares = []
>>> for x in range(10):
...     square = x**2
...     if square > 5 and square < 50:
...         special_squares.append(square)
...
>>> print special_squares
[9, 16, 25, 36, 49]
```

6. Suppose we want to take a list of names and find only those starting with the letter *B*:

```
>>> names = ['Anne', 'Amy', 'Bob', 'David', 'Carrie', 'Barbara', 'Zach']
>>> b_names = []
>>> for name in names:
...     if name.startswith('B'):
...         b_names.append(name)
...
>>> print b_names
['Bob', 'Barbara']
```

Map, Lambda, and Filter

One way to achieve the same goals as in the above examples is to use some of Python's tools from fu
`filter()`, and **`lambda()`**.

map()

The `map()` function applies a function to every member of an iterable and returns the result. Typically, c
inline function as defined by *lambda*, but it is possible to use any function. The *first example above* ca
`map()` as follows:

```
>>> def square(x):
...     return x**2
...
>>> squares = map(square, range(10))
>>> print squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In the case of the *second example*, the `len()` function already exists, so we can use `map()` to apply it to

```
>>> names = ['Anne', 'Amy', 'Bob', 'David', 'Carrie', 'Barbara', 'Zach']
>>> lengths = map(len, names)
>>> print lengths
[4, 3, 3, 5, 6, 7, 4]
```

lambda

In the first *map* example above, we created a function, called `square`, so that *map* would have a funct
This is a common occurrence, so Python provides the ability to create a simple (no statements allowe
function using a so-called lambda form. Thus, an anonymous function that returns the square of its
`lambda x: x**2`. This means, "Here is an anonymous (nameless) function that takes one arguement, ca
of *x*. Since the lambda form actually evaluates to a function, we can also call it inline. (This is gener
show it here to demonstrate that lambda forms are actually inline function objects.):

```
>>> print (lambda x: x**2)(5) # first parens contain a lambda form that is a squaring function, seco
25
>>> # Make a function of two arguments (x and y) that returns their product, then call the function
>>> print (lambda x, y: x*y)(3, 4)
12
>>> print (lambda x: x.startswith('B'))('Bob')
True
>>> print (lambda x: x.startswith('B'))('Robert')
False
>>> incrementer = lambda input: input+1
>>> # above is same as def incrementer(input): return input+1
>>> # now we call it
>>> print incrementer(3)
4
```

Using lambda and map together

Lambda forms can be used as the required function argument to the `map()` function. For example, th
written as

```
>>> squares = map(lambda x: x**2, range(10))
>>> print squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81
```

In English, this means, “Apply a function that squares its argument (the lambda form) to every membe to 9 (the `range()`), and store the result in a list called *squares*.

`filter()`

The *fifth* and *sixth* examples above can also be achieved with the `filter()` built-in function. Filter take `False` and applies it to a sequence, returning a list of only those members of the sequence for which th

Lambda forms can also be used with the *filter* function; in fact, they can be used anywhere a function *fifth example*, the list of squares is filtered according to whether the given entries are greater than 5 form that returns `True` when this condition is met is `lambda x: x > 5 and x < 50`. Thus, we can reproduce

```
>>> squares = map(lambda x: x**2, range(10))
>>> special_squares = filter(lambda x: x > 5 and x < 50, squares)
>>> print special_squares
[9, 16, 25, 36, 49]
```

In English, this means, “Find every member of the *squares* list for which the member is greater tha member for which `lambda x: x > 5 and x < 50` returns `True`), and store that in a new variable called *speci*

Similarly, the *sixth example <sixth-example-list-comprehension>* can be reproduced using filter as follo

```
>>> names = ['Anne', 'Amy', 'Bob', 'David', 'Carrie', 'Barbara', 'Zach']
>>> b_names = filter(lambda s: s.startswith('B'), names)
>>> print b_names
['Bob', 'Barbara']
```

List Comprehensions

All of the six original examples can be achieved using a consistent, clean, and elegant syntax called lis

Simple list comprehensions

The simplest form of a list comprehension is

> **[** `expression-involving-loop-variable` **for** `loop-variable` **in** `sequence` **]**

This will step over every element of `sequence`, successively setting `loop-variable` equal to every elemer build up a list by evaluating `expression-involving-loop-variable` for each one. This eliminates the need t generally produces a much more readable code than using `map()` and a more compact code than using

The *first example <first-example-list-comprehension>* can thus be written compactly as:

```
>>> squares = [ x**2 for x in range(10) ]
>>> print squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Some other simple examples:

- Print the length of each word in the list of `names`:

  ```
  >>> print [ len(name) for name in names ]
  [4, 3, 3, 5, 6, 7, 4]
  ```

- Print the last letter of each name in the list of names:

  ```
  >>> print [ name[-1] for name in names ]
  ['e', 'y', 'b', 'd', 'e', 'a', 'h']
  ```

- Print the reverse of each name in the list of names:

  ```
  >>> print [ name[::-1] for name in names ]
  ['ennA', 'ymA', 'boB', 'divaD', 'eirraC', 'arabraB', 'hcaZ']
  ```

Note that complex expressions can be put in the slot for `expression-involving-loop-variable`. For exam names, first letters, and lengths for each name in the list:

```
>>> print [ [name, name[0], len(name)] for name in names ]
[['Anne', 'A', 4], ['Amy', 'A', 3], ['Bob', 'B', 3], ['David', 'D', 5], ['Carrie', 'C', 6], ['Barbar
```

Where `[name, name[0], len(name)]` occupies the `expression-involving-loop-variable` slot, so that the list cc `[name, name[0], len(name)]` for every `name` in the `names` sequence.

Nested list comprehensions

List comprehensions can be nested, in which case they take on the following form:

[`expression-involving-loop-variables` **for** `outer-loop-variable` **in** `outer-sequence` **for** `inner-loop-variable` **in** `i`

This is equivalent to writing:

```
results = []
for outer_loop_variable in outer_sequence:
    for inner_loop_variable in inner_sequence:
        results.append( expression_involving_loop_variables )
```

Thus, the *third example* can be written compactly as:

```
>>> possible_choices = [ [drink,food] for drink in ['water', 'tea', 'juice'] for food in ['ham', 'eg
>>> print possible_choices
[['water', 'ham'], ['water', 'eggs'], ['water', 'spam'], ['tea', 'ham'], ['tea', 'eggs'], ['tea', 's
```

And example *4.* can be written as

```
>>> coords = [ (x,y) for x in range(5) for y in range(3) ] # points on a rectangular grid
>>> print coords
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2), (4,
```

Filtered list comprehensions

The final form of list comprehension involves creating a list and filtering it similarly to using `filte` comprehension takes the following form:

> **[** `expression-involving-loop-variable` **for** `loop-variable` **in** `sequence` **if** `boolean-expression-involving-loo`

This form is similar to the simple form of list comprehension, but it evaluates `boolean-expression-inv` item and keeps only those members for which the boolean expression is `True`. Thus we can use li *example 5* as

```
>>> special_squares = [ x**2 for x in range(10) if x**2 > 5 and x**2 < 50 ]
>>> print special_squares
[9, 16, 25, 36, 49]
```

Note that the above is inefficient, however, since it has to calculate the square of `x` three separate t loop. Thus, the following is an equivalent and more efficient approach:

```
>>> squares = [ x**2 for x in range(10) ]
>>> special_squares = [ s for s in squares if s > 5 and s < 50 ]
```

Finally, note that the foregoing can be written on a single line using a pair of list comprehensions as fo

```
>>> special_squares = [ s for s in [ x**2 for x in range(10) ] if s > 5 and s < 50 ]
```

Example 6 can be rewritten using a list comprehension as:

```
>>> names = ['Anne', 'Amy', 'Bob', 'David', 'Carrie', 'Barbara', 'Zach']
>>> b_names = [ name for name in names if name.startswith('B') ]
>>> print b_names
['Bob', 'Barbara']
```

Or, again combining the first two lines into one,

```
>>> b_names = [ name for name in ['Anne', 'Amy', 'Bob', 'David', 'Carrie', 'Barbara', 'Zach'] if nam
```