Comprehending Comprehensions

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

Transformation

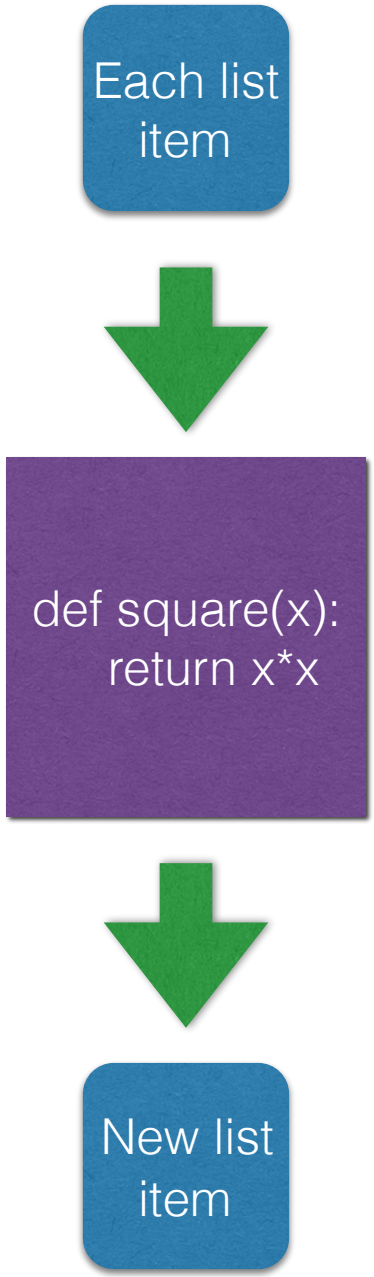
- You often want to turn one iterable into another
- For example, you might want to turn the list

```
[0,1,2,3,4]
```

- into the list

```
[0,1,4,9,16]
```

- We can *transform* the first list into the other by applying a Python function.


Each list item
def square(x):
return x*x
New list item

The usual solution

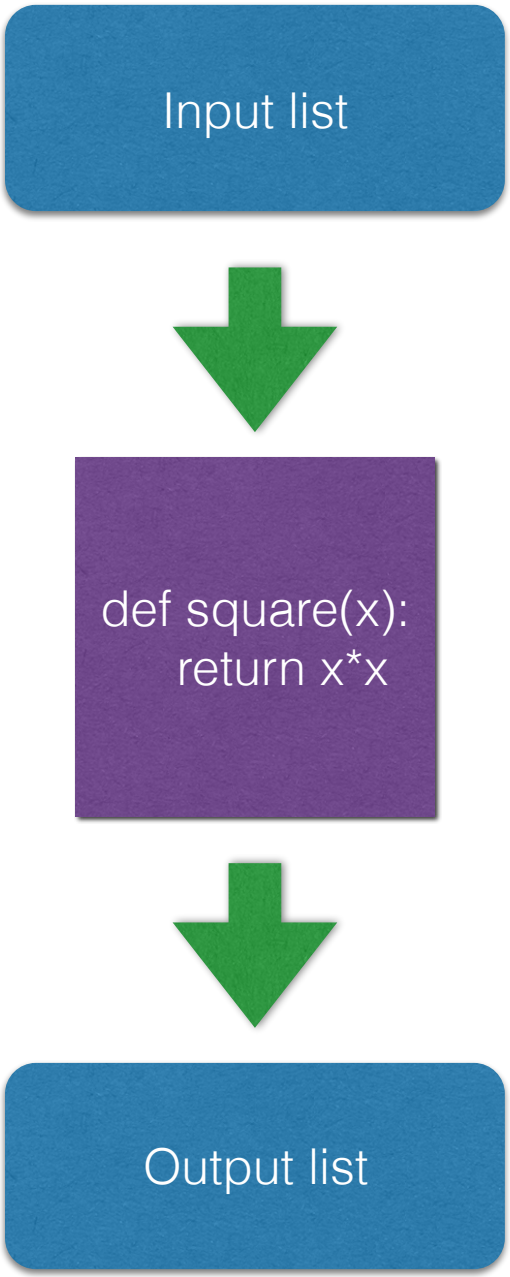
```
>>>> input = range(5)

>>>> def square(x):

        return x*x


>>> output = [ ]

>>> for x in input:

        output.append(square(x))


>>> output

[0, 1, 4, 9, 16]
```

What's wrong with this?

- Nothing is *wrong*.
- But functional programming looks at this, and says:
  - Why create a new variable ("output")?
  - Why are you building it, one step at a time?
  - Why are you modifying the state of "output"?
  - Why do it in such a clumsy way?

The elegant way

- "I want to apply my function, *square*, to each and every element of the input list, and get a list back."


Input list
def square(x):
return x*x
Output list

List comprehensions

- In Python, we do this with a "list comprehension":

```
[square(x) for x in range(5)]
```

- Or if you prefer:

```
[x*x for x in range(5)]
```

List comprehensions

- This expresses what we previously said:
  - I want a new list.
  - This new list should have the same number of elements as the input list.
  - Each element of the new list should be the result of applying square(x) to each element.

Many, many uses

- List comprehensions are powerful because they expression this idea in a compact, elegant form
- (And yes, it's a bit hard to read. I admit it!)
- Any time you have an iterable, and want to do something with each element, you likely want to use a list comprehension.

Ints to strings

- I can't say

```
', '.join(range(5))
```

- because str.join's input must contain strings.
- Solution:

```
', '.join([str(x) for x in range(5)])
```

Lowercase all words

- I can transform a sentence into all lowercase:

```
words = 'This is a sentence for my Python class'.split()

[word.lower() for word in words]
```

- Or even:

```
' '.join([word.lower() for word in words])
```

Filenames to files

- I can get a list of filenames from os.listdir:

```
os.listdir('/etc')
```

- I can get a file object for each of these:

```
[open('/etc/' + filename)
 for filename in os.listdir('/etc')]
```

File contents

- If I want to get the names of users on my Unix system, I can say

```
[ line.split(":")[0]

  for line in open('/etc/passwd') ]
```

Pig Latin!

```
def plword(word):
    vowels = 'aeiou'
    if word[0] in vowels:
        return word + 'way'
    else:
        return word[1:] + word[0] + 'ay'
```

Translation

```
' '.join([plword(word)

          for word in open('column-215') ])
```

List of dicts

- If I have a list of dicts ("people), each of which looks like:

```
p1 = {'first_name':'Reuven',
'last_name':'Lerner', 'phone':'054-496-8405'}
```

- I can get each person's full name as follows:

```
[ person['first_name']+' '+person['last_name']

 for person in people ]
```

Comprehensions

- Once you start to use list comprehensions, you'll see opportunities for this kind of transformation, or "mapping," just about everywhere.
- Python uses iterables in a lot of places, which means that you have many, many opportunities to do this
- It's often worth turning your data into an iterable, so that you can put it inside of a list comprehension!

Creating sets

- We can create sets by passing set() an iterable
- So we can create a set with:

```
set([x*x for x in range(5)])
```

- In modern versions of Python, we can also say:

```
{x*x for x in range(5)}
```

- Curly braces give us a set — a *set comprehension*

Set comprehensions

- Create a set, based on any iterable
- Lots of uses:
  - Usernames
  - Filenames
- Anything you get that's non-unique, which you want to make unique, is a perfect candidate!

Dict comprehensions

- Why let sets have all of the fun?
- Use curly braces, just like a set comprehension — but then separate the two values with a colon (:), just like in a dictionary definition.

```
{ line.split(':')[0] : line.split(':')[2]

  for line in open('/etc/passwd')

  if line[0] != '#' }
```

Dict comprehensions

- If a key appears more than once, the dict removes all but the first
- You need to have the

You can...

```
>>> query_string = 'a=1&b=2&c=xyz'

>>> [item.split('=')

    for item in query_string.split('&')]

[['a', '1'], ['b', '2'], ['c', 'abc']]


>>> dict([item.split('=')

    for item in query_string.split('&')])

{'a': '1', 'b': '2', 'c': 'xyz'}
```

... but even better

```
>>> query_string = 'a=1&b=2&c=xyz'

>>> { item.split('=')[0] : item.split('=')[1]
    for item in query_string.split('&') }

{'a': '1', 'b': '2', 'c': 'xyz'}
```

Filtering

- By default, a comprehension returns a collection with the same number of elements as its input.
- However, we can add an "if" statement to the end, which filters the output.
- Only those items for which the expression returns True" will be output

Filtering

- I can say:

```
[x*x for x in range(10) if x%2]
```

- That allows

```
[1, 9, 25, 49, 81]
```

Loops vs. comprehensions

- Many people ask me why they should use list comprehensions, when we already have "for" loops.
- The answer: These are completely different things!

Who cares?

- Comprehensions let you create lists, dictionaries, and sets quickly and easily.
- Moreover, they let you *map* the values from one collection to another
- Indeed, comprehensions are the modern incarnations of two very old functions, "map" and "filter"

Immutable data

- We know that Python has both mutable and immutable data structures
- In functional programming, we pretend that our data structures are immutable, even if they aren't
- But if we want to enforce immutable data, we can do it — typically using tuples

Dictionary comprehensions

- Just like a list comprehension, but with curly braces and name:value as the output

```
{ word:word.lower() for word in 'ABC DEF GHI'.split() }
```

```
{ 'ABC': 'abc', 'DEF': 'def', 'GHI': 'ghi'}
```

Set comprehensions

- Set comprehensions!

```
{ word.lower() for word in 'ABC DEF GHI'.split() }

    set(['abc', 'ghi', 'def'])
```

Nested list comprehensions

- A typical example:

```
[(x,y) for x in range(5) for y in range(5)]
```

- Huh?!?

More readable

```
[(x,y)
 for x in range(5)
 for y in range(5)]
```

More sophistication

```
[(x,y)
 for x in range(5)
 for y in range(x+1)]
```

Game scores

```
{'Reuven':[300, 250, 350, 400],
 'Atara':[200, 300, 450, 150],
 'Shikma':[250, 380, 420, 120],
 'Amotz':[100, 120, 150, 180]
}
def average(scores):
 return sum(scores) / len(scores)
```

Get all game scores

```
>>> [score
    for score_list in s.values()
    for score in score_list]
```

```
[300, 250, 350, 400, 100, 120, 150, 180,
200, 300, 450, 150, 250, 380, 420, 120]
```

Average score across all people

```
>>> average([ one_score
 for one_player_scores in scores.values()
 for one_score in one_player_scores ])
```

Average score across all people (but ignoring <200)

```
>>> [ one_score
    for one_player_scores in scores.values()
    for one_score in one_player_scores
   if one_score > 200]
```

```
[300, 250, 350, 400, 300, 450, 250, 380, 420]
```

Rooms

```
rooms = [[
 {'age': 14, 'hobby': 'horses', 'name': 'A'},
 {'age': 12, 'hobby': 'piano', 'name': 'B'},
 {'age': 9, 'hobby': 'chess', 'name': 'C'}],
 [{'age': 15, 'hobby': 'programming', 'name': 'D'},
 {'age': 17, 'hobby': 'driving', 'name': 'E'}],
 [{'age': 45, 'hobby': 'writing', 'name': 'F'},
 {'age': 43, 'hobby': 'chess', 'name': 'G'}]]
```

Names of guests

```
>>> [ person['name']
    for room in rooms
    for person in room ]

['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

Chess players' names

```
>>> [ person['name']
 for room in rooms
 for person in room
 if person['hobby'] == 'chess' ]


['C', 'G']
```