List Comprehensions (And Other Comprehensions)

What is a list comprehension?

A **list comprehension** is a way to create a list in Python.

It **high level** and **declarative**.

Squares

```
>>>
>>> squares = []
>>> for n in range(5):
...     squares.append(n**2)
...
>>> squares
[0, 1, 4, 9, 16]
```

Higher Level

```
>>> [ n**2 for n in range(5) ]
[0, 1, 4, 9, 16]
```

Structure

```
[ EXPR for VAR in SEQ ]
```

```
>>> [ n**2 for n in range(5) ]
[0, 1, 4, 9, 16]
```

- Expression (in terms of variable)
- The name of that variable
- The source sequence

Source can be any sequence

```
[ EXPR for VAR in SEQ ]
```

SEQ can be:

- A list or tuple
- An iterator
- Even a set

Any experssion

```
[ EXPR for VAR in SEQ ]
```

EXPR can be any Python expression:

- Arithmetic expressions like `n+3`
- A function call like `f(m)`, using `m` as the variable
- A slice operation (like `s[::-1]`, to reverse a string)
- Method calls (`foo.bar()`, iterating over a sequence of objects)

Examples

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> [ 2*m+3 for m in range(10, 20, 2) ]
[23, 27, 31, 35, 39]

>>> [ abs(num) for num in numbers ]
[9, 1, 4, 20, 11, 3]

>>> [ 10 - x for x in numbers ]
[1, 11, 14, -10, -1, 13]
```

Examples

```
>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> def repeat(s):
...     return s + s

>>> [ pet.lower() for pet in pets ]
['dog', 'parakeet', 'cat', 'llama']

>>> [ "The " + pet for pet in sorted(pets) ]
['The cat', 'The dog', 'The llama', 'The parakeet']

>>> [ repeat(pet) for pet in pets ]
['dogdog', 'parakeetparakeet', 'catcat', 'llamallama']
```

Practice the syntax

Type in the following on a Python prompt:

```
>>> colors = ["red", "green", "blue"]
>>> [ n*3 for n in range(5) ]
[0, 3, 6, 9, 12]

>>> [ abs(x) for x in range(-3, 3) ]
[3, 2, 1, 0, 1, 2]

>>> [ color.upper() for color in colors ]
['RED', 'GREEN', 'BLUE']
```

Filtering

List comprehensions can exclude elements.

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]

>>> # Positive numbers:
... [ n for n in numbers if n > 0 ]
[9, 20, 11]

>>> # Squares of even numbers:
... [ n**2 for n in numbers if n % 2 == 0 ]
[16, 400]
```

"If" Structure

```
[ EXPR for VAR in SEQ if CONDITION ]
```

```
>>> [ n+1 for n in numbers if n > 0 ]
[10, 21, 12]
>>> [ n**2 for n in numbers if n % 2 == 0 ]
[16, 400]
```

More complex If

```
>>> def is_palindrome(s):
...     return s == s[::-1]
>>> words = ["bias", "dad", "eye", "deed", "tooth"]
>>> palindromes = [ word for word in words
...                  if is_palindrome(word)]
>>> print(palindromes)
['dad', 'eye', 'deed']
```

Function of VAR

```
[ EXPR for VAR in SEQ if CONDITION ]
```

In general, both EXR and CONDITION will be a function of VAR.

```
[ some_expr(n) for n in some_seq
  if some_condition(n)]
```

One of the only exceptions:

```
# List of ten 0's
[ 0 for n in range(10) ]
```

Required Syntax

You must have the **for** and **in** keywords, always. (And **if** if you're filtering). Even if the expression and variable are the same.

```
>>> # Like this:
... [ n for n in numbers if n > 3 ]
[9, 20, 11]
>>> # Nope:
... [ n in numbers if n > 3 ]
  File "<stdin>", line 2
    [ n in numbers if n > 3 ]
                           ^
SyntaxError: invalid syntax
```

Practice the syntax

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> def is_even(n):
...     return n % 2 == 0
...
>>> [ n for n in numbers if n > 0 ]
[9, 20, 11]
>>>
>>> [ n*2 for n in numbers if n < 10 ]
[18, -2, -8, -6]
>>>
>>> [ 20-n for n in numbers if is_even(n)]
[24, 0]
```

Benefits

- Very readable
- Low cognitive overhead
- Very maintainable

Indendation

You can (and should!) split the comprehension across multiple lines.

```
def double_short_words(words):
    return [ word + word
             for word in words
             if len(word) < 5 ]
```

Indendation 2

Python's normal whitespace rules are suspended within a list comprehension's brackets. Here's another way to do it:

```
def double_short_words(words):
    return [
        word + word
        for word in words
        if len(word) < 5
    ]
```

Lab: List Comprehensions

Lab file: `comprehensions/listcomp.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... then do `comprehensions/listcomp_extra.py`

Other Comprehensions

Dictionary comprehensions:

```
>>> blocks = { n: "x" * n for n in range(5) }
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

Set comprehensions

Imagine a Student class, which includes a "major" attribute.

```
>>> # A list of student majors...
... [ student.major for student in students ]
['Computer Science', 'Economics', 'Computer Science', 'Economics', 'Basket
Weaving']
>>> # And a set of majors:
... { student.major for student in students }
{'Economics', 'Computer Science', 'Basket Weaving'}
>>> # You can also use the set() built-in.
... set(student.major for student in students)
{'Economics', 'Computer Science', 'Basket Weaving'}
```

Generator Expressions

If you use parenthesis instead of square brackets, something really interesting happens.

```
>>> items = ( n*3 for n in range(5))
>>> type(items)
<class 'generator'>
>>> next(items)
0
>>> next(items)
3
>>> next(items)
6
```

Generator Expressions

```
>>> list_comp = [ n*3 for n in range(5) ]
>>> type(list_comp)
<class 'list'>
>>> gen_expr = ( n*3 for n in range(5) )
>>> type(gen_expr)
<class 'generator'>
```

Another way to generate

It turns out that this...

```
items = ( n*3 for n in range(5) )
```

... is EXACTLY equivalent to this:

```
def gen_items():
    for n in range(5):
        yield n*3

items = gen_items()
```

((Pro Tip))

When passing a generator expression inline to a function, you can omit the parenthesis:

```
>>> sorted( (student.name for student in students) )
['Jones, Tina', 'Shan, Geetha', 'Simmons, Russell', 'Smith, Joe']
>>> sorted( student.name for student in students )
['Jones, Tina', 'Shan, Geetha', 'Simmons, Russell', 'Smith, Joe']
```

But not always:

```
>>> sorted( student.name for student in students, reversed=True)
  File "<stdin>", line 1
SyntaxError: Generator expression must be parenthesized if not sole argument
```

Rule of thumb: `( ( ... ) )` can be replaced with `( ... )`

Expression vs. Comprehension

Why are they called "generator expressions" instead of "generator comprehensions"?

Historical reasons. But some of us are trying to change that.