List Comprehensions

Squares

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

Higher Level

As a **list comprehension**:

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

Exactly equivalent to this:

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

Structure

High level and **declarative**.

`[ EXPR for VAR in SEQ ]`

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

EXPR (in terms of VAR) can be:	SEQ can be:
• 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)	• A list or tuple • A generator object • Any iterator • Even another comprehension

Examples

```
>>> [ 2*m+3 for m in range(10, 20, 2) ]
[23, 27, 31, 35, 39]

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

>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> [ pet.upper() for pet in pets ]
['DOG', 'PARAKEET', 'CAT', 'LLAMA']

>>> def repeat(s):
...     return s + s
>>> [ 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"]
>>> [ z*3 for z 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']
```

Multiple for's

Comprehensions can have several `for` clauses.

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard', 'orange doll',
'purple bike', 'purple basketball', 'purple skateboard', 'purple doll', 'pink
bike', 'pink basketball', 'pink skateboard', 'pink doll']
```

Chaining "for" clauses

You can chain multiple `for` clauses together, effectively generating the source sequence.

```
>>> ranges = [range(1, 7), range(4, 12, 3), range(-5, 9, 4)]
>>> [ float(num)
...   for subrange in ranges
...   for num in subrange ]
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 4.0, 7.0, 10.0, -5.0, -1.0, 3.0, 7.0]
```

But order matters.

```
>>> [ float(num)
...   for num in subrange
...   for subrange in ranges ]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'subrange' is not defined
```

Chaining Vs. Combos

If the variables in the `for` clauses overlap, you're chaining. The expression depends only on the variable in the last clause.

```
[ float(num)
  for subrange in ranges
  for num in subrange ]
```

If no overlap, the expression will depend on *all* the for-clause variables. Mathematically, it's like an outer product.

```
[ color + " " + toy
  for color in colors
  for toy in toys ]
```

For this last one, does the order of the `for` expressions matter?

Ordering "for" clauses

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard', 'orange doll',
'purple bike', 'purple basketball', 'purple skateboard', 'purple doll', 'pink
bike', 'pink basketball', 'pink skateboard', 'pink doll']
>>>
>>> [ color + " " + toy
...   for toy in toys
...   for color in colors ]
['orange bike', 'purple bike', 'pink bike', 'orange basketball', 'purple
basketball', 'pink basketball', 'orange skateboard', 'purple skateboard',
'pink skateboard', 'orange doll', 'purple doll', 'pink doll']
```

`for` clause order affects the resulting element order. You can choose the order if the `for` clauses are independent (not chained).

Filtering

List comprehensions can exclude elements.

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

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

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

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

>>> # Incrementing some numbers.
... [ x+1 for x in numbers if x > 0 ]
[10, 21, 12]
```

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(x) for x in some_seq
  if some_condition(x)]
```

One of the only exceptions:

```
# List of ten 0's
[ 0 for x 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:
... [ x for x in numbers if x > 3 ]
[9, 20, 11]
>>> # Nope:
... [ x in numbers if x > 3 ]
  File "<stdin>", line 2
    [ x in numbers if x > 3 ]
                            ^
SyntaxError: invalid syntax
```

Indentation

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 ]
```

Another style:

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

Multiple for's and if's

```
>>> weights = [0.2, 0.5, 0.9]
>>> values = [27.5, 13.4]
>>> offsets = [4.3, 7.1, 9.5]
>>>
>>> [ (weight, value, offset)
...   for weight in weights
...   for value in values
...   for offset in offsets ]
[(0.2, 27.5, 4.3), (0.2, 27.5, 7.1), (0.2, 27.5, 9.5), (0.2, 13.4, 4.3),
(0.2, 13.4, 7.1), (0.2, 13.4, 9.5), (0.5, 27.5, 4.3), (0.5, 27.5, 7.1), (0.5,
27.5, 9.5), (0.5, 13.4, 4.3), (0.5, 13.4, 7.1), (0.5, 13.4, 9.5), (0.9, 27.5,
4.3), (0.9, 27.5, 7.1), (0.9, 27.5, 9.5), (0.9, 13.4, 4.3), (0.9, 13.4, 7.1),
(0.9, 13.4, 9.5)]
>>> [ (weight, value, offset)
...   for weight in weights
...   for value in values
...   for offset in offsets
...   if offset > 5.0
...   if weight * value < offset ]
[(0.2, 27.5, 7.1), (0.2, 27.5, 9.5), (0.2, 13.4, 7.1), (0.2, 13.4, 9.5),
(0.5, 13.4, 7.1), (0.5, 13.4, 9.5)]
```

Benefits

- Very readable
- Low cognitive overhead
- Very maintainable

AND: List comprehensions are the foundation of other useful techniques. More in the next video.

Lab: List Comprehensions

Lab file: `comprehensions/listcomp.py`

- In `labs` folder
- When you are done, study the solution - compare to what you wrote.
- … and then optionally do `comprehensions/listcomp_extra.py`