List Comprehensions
Examples to Get Us Started

- \([x**2 \text{ for } x \text{ in } \text{range}(10)]\)
  
  \([0, 1, 4, 9, 16, 25, 36, 49, 64, 81]\)

- \([\text{str}(x)+\text{str}(x) \text{ for } x \text{ in } \text{range}(10)]\)
  
  \(['00', '11', '22', '33', '44', '55', '66', '77', '88', '99']\)

- \([\text{str}(x)+\text{str}(x) \text{ for } x \text{ in } \text{range}(10) \text{ if } x\%2 == 0]\)
  
  \(['00', '22', '44', '66', '88']\)
These are all list comprehensions

- They provide a flexible, fast, and compact way of creating new lists from old lists.

- List comprehensions provide a more compact and more efficient alternative to explicitly using for-loops.

- See Sections 5.1.3 and 5.1.4 (on List Comprehensions) from Python 3 tutorial at www.python.org
List Comprehension: Basic Syntax

\[ expr \text{ for } x \text{ in list} \]

Notes:

- `for` and `in` are Python keywords, used just as in for-loops.
- `x` is a variable that takes on values of elements in list, in order.
- `expr` is Python expression, typically involving the variable `x`.
- The expression \[ expr \text{ for } x \text{ in list} \] evaluates to a list made up of the different values that `expr` takes on for different `x`.
- This is similar to the “set builder” notation used in math: \{x*y | x and y are even\}.
List Comprehensions: Syntax with if-clause

\[[\text{expr} \text{ for } x \text{ in } \text{list} \text{ if } \text{bool-expr}]\]

Notes:
- \text{bool-expr} is a boolean expression involving \text{x}.
- The overall expression evaluates to a list of values of \text{expr} evaluated for all values of \text{x} in \text{list} satisfying the \text{bool-expr}.
- \textbf{Example:} \[[\text{str}(x)+\text{str}(x) \text{ for } x \text{ in } \text{range}(10) \text{ if } x\%2 == 0] \text{ evaluates to } ['00', '22', '44', '66', '88']\]
Examples

- Generating lists of lists.

   \[
   \text{[range(x) for x in range(1, 5)]} \\
   \text{Evaluates to: [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3]]}
   \]

- All numbers in the range 0..49 containing the digit “7”.

   \[
   \text{[x for x in range(50) if "7" in str(x)]} \\
   \text{Evaluates to: [7, 17, 27, 37, 47]}
   \]
Nested List Comprehensions

Example:
\[ x \times y \text{ for } x \text{ in } \text{range}(3) \text{ for } y \text{ in } \text{range}(3) \]
\[ [0, 0, 0, 0, 1, 2, 0, 2, 4] \]

Notes:
- As in nested loops, for every iteration of the first loop (the for-x loop), all iterations of the second loop (the for-y loop) are executed.
Example: Generating Perfect Squares

\[ x \text{ for } x \text{ in range}(100) \text{ for } y \text{ in range}(x) \text{ if } y^2 == x \]
\[ 4, 9, 16, 25, 36, 49, 64, 81 \]

Notes:
- Those \( x \) and \( y \) values (from their respective lists) that satisfy the condition \( y^2 = x \), are generated.
- Thus all \( x \) values generated in this manner are perfect squares.
Example: Generating Composites

```python
composites = [x for y in range(2, 10) for x in range(2*y, 100, y)]
```

**Notes:**

- For each $y = 2, 3, \ldots, 9$, the variable $x$ takes on values that are multiples of $y$.
- For $y = 2$, the variable $x$ takes on values 4, 6, 8,\ldots, 98.
- For $y = 3$, the variable $x$ takes on values 6, 9, 12,\ldots, 99.
- Thus the values of $x$ generated in this manner are (strict) multiples of 2, 3, 4,\ldots, 9.
- This covers all composites in the range 2..99.
Example: Generating Prime Numbers

`primes = [x for x in range(2, 100) if x not in composites]`

Notes:

- Primes in the range 2..99 can be obtained by taking the complement of the generated composites.
Example: Flattening Lists

```python
>>> nestedList = [range(x) for x in range(1, 4)]
>>> nestedList
[[0], [0, 1], [0, 1, 2]]
>>> [y for x in nestedList for y in x]
[0, 0, 1, 0, 1, 2]
```
Example: Transposing a Matrix

```python
>>> mat = [[3, 0, 1],
        [2, 1, 7],
        [1, 3, 9]]

>>> [[mat[i][j] for i in range(len(mat))] for j in range(len(mat))]
>>=[[3, 2, 1], [0, 1, 3], [1, 7, 9]]
```

**Notes:**
- The expression, which is the first element of the list comprehension, itself happens to be a list comprehension.
- Therefore, each element of the constructed list, is a list itself.
Warning!

• The danger with list comprehensions is that your code may become hard to understand, especially with nested list comprehensions.

• If by using a list comprehension, you are making your code hard to understand, then it is time to desist.