List Comprehensions
Python’s higher-order functions

• Python supports higher-order functions that operate on lists similar to Scheme’s

```python
>>> def square(x):
    return x*x

>>> def even(x):
    return 0 == x % 2

>>> map(square, range(10,20))
[100, 121, 144, 169, 196, 225, 256, 289, 324, 361]

>>> filter(even, range(10,20))
[10, 12, 14, 16, 18]

>>> map(square, filter(even, range(10,20)))
[100, 144, 196, 256, 324]
```

• But many Python programmers prefer to use list comprehensions, instead
List Comprehensions

• A list comprehension is a programming language construct for creating a list based on existing lists
  • Haskell, Erlang, Scala and Python have them
• Why “comprehension”? The term is borrowed from math’s set comprehension notation for defining sets in terms of other sets
• A powerful and popular feature in Python
  • Generate a new list by applying a function to every member of an original list
• Python’s notation:
  \[
  [ \text{expression} \text{ for } \text{name} \text{ in } \text{list} ]
  \]
List Comprehensions

• The syntax of a list comprehension is somewhat tricky

\[ [x-10 \text{ for } x \text{ in } \text{grades if } x>0] \]

• Syntax suggests that of a for-loop, an in operation, or an if statement

• All three of these keywords (‘for’, ‘in’, and ‘if’) are also used in the syntax of forms of list comprehensions

\[ [\text{expression for name in list}] \]
List Comprehensions

>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]

[ **expression** for **name** in **list** ]

- Where **expression** is some calculation or operation acting upon the variable **name**.
- For each member of the **list**, the list comprehension
  1. sets **name** equal to that member,
  2. calculates a new value using **expression**,
- It then collects these new values into a list which is the return value of the list comprehension.

Note: Non-standard colors on next few slides clarify the list comprehension syntax.
List Comprehensions

• If list contains elements of different types, then expression must operate correctly on the types of all of list members.

• If the elements of list are other containers, then name can consist of a container of names matching the type and “shape” of the list members.

```python
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]
```

• Containers are objects that contain references to other objects (e.g., lists, types, dictionaries)
List Comprehensions

• *expression* can also contain user-defined functions.

```python
>>> def subtract(a, b):
    return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

[ *expression* for *name* in *list* ]
List comprehensions can be viewed as syntactic sugar for a typical higher-order functions

\[
[ \text{expression for name in list } ]
\]

\[
\text{map( lambda name: expression, list )}
\]

\[
[ 2x+1 \text{ for x in } [10, 20, 30] ]
\]

\[
\text{map( lambda x: 2x+1, [10, 20, 30] )}
\]
Filtered List Comprehension

• **Filter** determines whether **expression** is performed on each member of the **list**.

• For each element of **list**, checks if it satisfies the **filter condition**.

• If the **filter condition** returns *False*, that element is omitted from the **list** before the list comprehension is evaluated.

```
[ expression for name in list if filter]
```
Filtered List Comprehension

```python
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem*2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition
- So, only 12, 14, and 18 are produced.
More syntactic sugar

Including an if clause begins to show the benefits of the sweetened form

\[
\left[ \text{expression} \text{ for name in list if filt } \right]
\]

\[
\text{map}( \lambda \text{name} . \text{expression}, \text{filter}(\text{filt, list}))
\]

\[
\left[ \text{2x+1 for x in [10, 20, 30] if x > 0 } \right]
\]

\[
\text{map}( \lambda \text{x: 2x+1},
\text{filter}( \lambda \text{x: x > 0 }, [10, 20, 30] )
\)
Nested List Comprehensions

• Since list comprehensions take a list as input and produce a list as output, they are easily nested

```python
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in [item+1 for item in li] ]
[8, 6, 10, 4]
```

• The inner comprehension produces: [4, 3, 5, 2]
• So, the outer one produces: [8, 6, 10, 4]
Syntactic sugar

\[ \left[ e_1 \text{ for } n_1 \text{ in } \left[ e_1 \text{ for } n_1 \text{ list } \right] \right] \text{ map( lambda } n_1: e_1, \right. \\
\left. \text{ map( lambda } n_2: e_2, \text{ list } ) \text{ ) } \right) \]

\[ 2x+1 \text{ for } x \text{ in } \left[ y^2 \text{ for } y \text{ in } [10, 20, 30] \right] \text{ map( lambda } x: 2x+1, \right. \\
\left. \text{ map( lambda } y: y^2, [10, 20, 30] ) \right) \]