Python Lists

List Methods

list.append(x)

Add an item to the end of the list; equivalent to a[len(a):] = [x].

list.extend(L)

Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

list.insert(i, x)

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

list.remove(x)

Remove the first item from the list whose value is x. It is an error if there is no such item.

list.pop([i])

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

list.index(x)

Return the index in the list of the first item whose value is x. It is an error if there is no such item.

list.count(x)

Return the number of times x appears in the list.

list.sort(cmp=None, key=None, reverse=False)

Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).

list.reverse()

Reverse the elements of the list, in place.

List Method Examples

>>> a = [66.25, 333, 333, 1, 1234.5]

>>> print a.count(333), a.count(66.25), a.count('x')

2 1 0

>>> a.insert(2, -1)

>>> a.append(333)

>>> a

[66.25, 333, -1, 333, 1, 1234.5, 333]

>>> a.index(333)

1

>>> a.remove(333)

>>> a

[66.25, -1, 333, 1, 1234.5, 333]

>>> a.reverse()

>>> a

[333, 1234.5, 1, 333, -1, 66.25]

>>> a.sort()

>>> a

[-1, 1, 66.25, 333, 333, 1234.5]

>>> a.pop()

1234.5

>>> a

[-1, 1, 66.25, 333, 333]

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index. For example:

>>> stack = [3, 4, 5]

>>> stack.append(6)

>>> stack.append(7)

>>> stack

[3, 4, 5, 6, 7]

>>> stack.pop()

7

>>> stack

[3, 4, 5, 6]

>>> stack.pop()

6

>>> stack.pop()

5

>>> stack

[3, 4]

Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends. For example:

>>> from collections import deque

>>> queue = deque(["Eric", "John", "Michael"])

>>> queue.append("Terry") # Terry arrives

>>> queue.append("Graham") # Graham arrives

>>> queue.popleft() # The first to arrive now leaves

'Eric'

>>> queue.popleft() # The second to arrive now leaves

'John'

>>> queue # Remaining queue in order of arrival

deque(['Michael', 'Terry', 'Graham'])

Functional Programming Tools

filter(function, sequence) returns a sequence consisting of those items from the sequence for which function(item) is true. If sequence is a str, unicode or tuple, the result will be of the same type; otherwise, it is always a list. For example, to compute a sequence of numbers divisible by 3 or 5:

>>> def f(x): return x % 3 == 0 or x % 5 == 0

...

>>> filter(f, range(2, 25))

[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]

map(function, sequence) calls function(item) for each of the sequence’s items and returns a list of the return values. For example, to compute some cubes:

>>> def cube(x): return x\*x\*x

...

>>> map(cube, range(1, 11))

[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or None if some sequence is shorter than another). For example:

>>> seq = range(8)

>>> def add(x, y): return x+y

...

>>> map(add, seq, seq)

[0, 2, 4, 6, 8, 10, 12, 14]

reduce(function, sequence) returns a single value constructed by calling the binary function function on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

>>> def add(x,y): return x+y

...

>>> reduce(add, range(1, 11))

55

If there’s only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

>>> def sum(seq):

... def add(x,y): return x+y

... return reduce(add, seq, 0)

...

>>> sum(range(1, 11))

55

>>> sum([])

0

List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

>>> squares = []

>>> for x in range(10):

... squares.append(x\*\*2)

...

>>> squares

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

We can obtain the same result with:

squares = [x\*\*2 for x in range(10)]

This is also equivalent to squares = map(lambda x: x\*\*2, range(10)), but it’s more concise and readable.

A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]

[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

and it’s equivalent to:

>>> combs = []

>>> for x in [1,2,3]:

... for y in [3,1,4]:

... if x != y:

... combs.append((x, y))

...

>>> combs

[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

Note how the order of the for and if statements is the same in both these snippets.

If the expression is a tuple (e.g. the (x, y) in the previous example), it must be parenthesized.

>>> vec = [-4, -2, 0, 2, 4]

>>> # create a new list with the values doubled

>>> [x\*2 for x in vec]

[-8, -4, 0, 4, 8]

>>> # filter the list to exclude negative numbers

>>> [x for x in vec if x >= 0]

[0, 2, 4]

>>> # apply a function to all the elements

>>> [abs(x) for x in vec]

[4, 2, 0, 2, 4]

>>> # call a method on each element

>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']

>>> [weapon.strip() for weapon in freshfruit]

['banana', 'loganberry', 'passion fruit']

>>> # create a list of 2-tuples like (number, square)

>>> [(x, x\*\*2) for x in range(6)]

[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

>>> # the tuple must be parenthesized, otherwise an error is raised

>>> [x, x\*\*2 for x in range(6)]

File "<stdin>", line 1, in <module>

[x, x\*\*2 for x in range(6)]

^

SyntaxError: invalid syntax

>>> # flatten a list using a listcomp with two 'for'

>>> vec = [[1,2,3], [4,5,6], [7,8,9]]

>>> [num for elem in vec for num in elem]

[1, 2, 3, 4, 5, 6, 7, 8, 9]

List comprehensions can contain complex expressions and nested functions:

>>> from math import pi

>>> [str(round(pi, i)) for i in range(1, 6)]

['3.1', '3.14', '3.142', '3.1416', '3.14159']

5.1.4.1. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

>>> matrix = [

... [1, 2, 3, 4],

... [5, 6, 7, 8],

... [9, 10, 11, 12],

... ]

The following list comprehension will transpose rows and columns:

>>> [[row[i] for row in matrix] for i in range(4)]

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

As we saw in the previous section, the nested listcomp is evaluated in the context of the for that follows it, so this example is equivalent to:

>>> transposed = []

>>> for i in range(4):

... transposed.append([row[i] for row in matrix])

...

>>> transposed

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

which, in turn, is the same as:

>>> transposed = []

>>> for i in range(4):

... # the following 3 lines implement the nested listcomp

... transposed\_row = []

... for row in matrix:

... transposed\_row.append(row[i])

... transposed.append(transposed\_row)

...

>>> transposed

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

In the real world, you should prefer built-in functions to complex flow statements. The zip() function would do a great job for this use case:

>>> zip(\*matrix)

[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]