An Introduction to Python Lists

Fredrik Lundh | August 2006

Overview

The list type is a container that holds a number of other objects, in a given order. The list type implements the sequence protocol, and also allows you to add and remove objects from the sequence.

Creating Lists

To create a list, put a number of expressions in square brackets:

```
L = []
L = [expression, ...]
```

This construct is known as a "list display". Python also supports computed lists, called "list comprehensions". In its simplest form, a list comprehension has the following syntax:

```
L = [expression for variable in sequence]
```

where the expression is evaluated once, for every item in the sequence.

The expressions can be anything; you can put all kinds of objects in lists, including other lists, and multiple references to a single object.

You can also use the built-in **list** type object to create lists:

```
L = list() # empty list
L = list(sequence)
L = list(expression for variable in sequence)
```

The sequence can be any kind of sequence object or iterable, including tuples and generators. If you pass in another list, the **list** function makes a copy.

Note that Python creates a single new list every time you *execute* the [] expression. No more, no less. And Python never creates a new list if you assign a list to a variable.

```
A = B = [] # both names will point to the same list

A = []
B = A # both names will point to the same list

A = []; B = [] # independent lists
```

For information on how to add items to a list once you've created it, see *Modifying Lists* below.

Accessing Lists

Lists implement the standard sequence interface; **len(L)** returns the number of items in the list, **L[i]** returns the item at index *i* (the first item has index 0), and **L[i:j]** returns a new list, containing the objects between *i* and *j*.

```
n = len(L)

item = L[index]

seq = L[start:stop]
```

If you pass in a negative index, Python adds the length of the list to the index. L[-1] can be used to access the last item in a list.

For normal indexing, if the resulting index is outside the list, Python raises an **IndexError** exception. Slices are treated as boundaries instead, and the result will simply contain all items between the boundaries.

Lists also support slice steps:

```
seq = L[start:stop:step]

seq = L[::2] # get every other item, starting with the first
seq = L[1::2] # get every other item, starting with the second
```

Looping Over Lists

The **for-in** statement makes it easy to loop over the items in a list:

```
for item in L:
    print item
```

If you need both the index and the item, use the **enumerate** function:

```
for index, item in enumerate(L):
    print index, item
```

If you need only the index, use **range** and **len**:

```
for index in range(len(L)):
    print index
```

The list object supports the iterator protocol. To explicitly create an iterator, use the built-in **iter** function:

```
i = iter(L)
item = i.next() # fetch first value
item = i.next() # fetch second value
```

Python provides various shortcuts for common list operations. For example, if a list contains numbers, the built-in **sum** function gives you the sum:

```
v = sum(L)

total = sum(L, subtotal)

average = float(sum(L)) / len(L)
```

If a list contains strings, you can combine the string into a single long string using the **join** string method:

```
s = ''.join(L)
```

Python also provides built-in operations to search for items, and to sort the list. These operations are described below.

Modifying Lists

The list type also allows you to assign to individual items or slices, and to delete them.

```
L[i] = obj
L[i:j] = sequence
```

Note that operations that modify the list will modify it in place. This means that if you have multiple variables that point to the same list, all variables will be updated at the same time.

```
L = []
M = L

# modify both lists
L.append(obj)
```

To create a separate list, you can use slicing or the **list** function to quickly create a copy:

```
L = []
M = L[:] # create a copy
```

```
# modify L only
L.append(obj)
```

You can also add items to an existing sequence. The **append** method adds a single item to the end of the list, the **extend** method adds items from another list (or any sequence) to the end, and **insert** inserts an item at a given index, and move the remaining items to the right.

```
L.append(item)
L.extend(sequence)
L.insert(index, item)
```

To insert items from another list or sequence at some other location, use slicing syntax:

```
L[index:index] = sequence
```

You can also remove items. The **del** statement can be used to remove an individual item, or to remove all items identified by a slice. The **pop** method removes an individual item and returns it, while **remove** searches for an item, and removes the first matching item from the list.

```
del L[i]
del L[i:j]
item = L.pop() # last item
item = L.pop(0) # first item
item = L.pop(index)
L.remove(item)
```

The **del** statement and the **pop** method does pretty much the same thing, except that **pop** returns the removed item.

Finally, the list type allows you to quickly reverse the order of the list.

```
L.reverse()
```

Reversing is fast, so temporarily reversing the list can often speed things up if you need to remove and insert a bunch of items at the beginning of the list:

```
L.reverse()
# append/insert/pop/delete at far end
L.reverse()
```

Note that the **for-in** statement maintains an internal index, which is incremented for each loop iteration. This means that if you modify the list you're looping over, the indexes will get out of sync, and you may end up skipping over items, or process the same item multiple times. To work around this, you can loop over a copy of the list:

```
for object in L[:]:
    if not condition:
        del L[index]
```

Alternatively, you can use create a new list, and append to it:

```
out = []
for object in L:
    if condition:
        out.append(object)
```

A common pattern is to apply a function to every item in a list, and replace the item with the return value from the function:

```
for index, object in enumerate(L):
    L[index] = function(object)

out = []
for object in L:
    out.append(function(object))
```

The above can be better written using either the built-in **map** function, or as a list comprehension:

```
out = map(function, L)

out = [function(object) for object in L]
```

For straightforward function calls, the **map** solution is more efficient, since the function object only needs to be fetched once. For other constructs (e.g. expressions or calls to object methods), you have to use a callback or a **lambda** to wrap the operation; in such cases, the list comprehension is more efficient, and usually also easier to read.

Again, if you need both the item and the index, use **enumerate**:

```
out = [function(index, object) for index, object in enumerate(L)]
```

You can use the list type to implement simple data structures, such as stacks and queues.

```
stack = []
stack.append(object) # push
object = stack.pop() # pop from end

queue = []
queue.append(object) # push
object = queue.pop(0) # pop from beginning
```

The list type isn't optimized for this, so this works best when the structures are small (typically a few hundred items or smaller). For larger structures, you may need a specialized data structure, such as **collections.deque**.

Another data structure for which a list works well in practice, as long as the structure is reasonably small, is an LRU (least-recently-used) container. The following statements moves an object to the end of the list:

```
lru.remove(item)
lru.append(item)
```

If you do the above every time you access an item in the LRU list, the least recently used items will move towards the beginning of the list. (for a simple cache implementation using this approach, see *Caching*.)

Searching Lists

The **in** operator can be used to check if an item is present in the list:

```
if value in L:
    print "list contains", value
```

To get the index of the first matching item, use **index**:

```
i = L.index(value)
```

The **index** method does a linear search, and stops at the first matching item. If no matching item is found, it raises a **ValueError** exception.

```
try:
    i = L.index(value)
except ValueError:
    i = -1 # no match
```

To get the index for all matching items, you can use a loop, and pass in a start index:

```
i = -1
try:
    while 1:
        i = L.index(value, i+1)
        print "match at", i
except ValueError:
    pass
```

Moving the loop into a helper function makes it easier to use:

```
def findall(L, value, start=0):
    # generator version
    i = start - 1
    try:
        i = L.index(value, i+1)
```

```
            yield i
        except ValueError:
            pass

for index in findall(L, value):
    print "match at", i
```

To count matching items, use the **count** method:

```
n = L.count(value)
```

Note that **count** loops over the entire list, so if you just want to check if a value is present in the list, you should use **in** or, where applicable, **index**.

To get the smallest or largest item in a list, use the built-in **min** and **max** functions:

```
lo = min(L)
hi = max(L)
```

As with **sort** (see below), you can pass in a **key** function that is used to map the list items before they are compared:

```
lo = min(L, key=int)
hi = max(L, key=int)
```

Sorting Lists

The **sort** method sorts a list in place.

```
L.sort()
```

To get a sorted copy, use the built-in **sorted** function:

```
out = sorted(L)
```

An in-place sort is slightly more efficient, since Python does not have to allocate a new list to hold the result.

By default, Python's sort algorithm determines the order by comparing the objects in the list against each other. You can override this by passing in a callable object that takes two items, and returns -1 for "less than", 0 for "equal", and 1 for "greater than". The built-in **cmp** function is often useful for this:

```
def compare(a, b):
    return cmp(int(a), int(b)) # compare as integers

L.sort(compare)

def compare_columns(a, b):
    # sort on ascending index 0, descending index 2
    return cmp(a[0], b[0]) or cmp(b[2], a[2])

out = sorted(L, compare_columns)
```

Alternatively, you can specify a mapping between list items and search keys. If you do this, the sort algorithm will make one pass over the data to build a key array, and then sort both the key array and the list based on the keys.

```
L.sort(key=int)

out = sorted(L, key=int)
```

If the transform is complex, or the list is large, this can be a lot faster than using a compare function, since the items only have to be transformed once.

Python's sort is stable; the order of items that compare equal will be preserved.

Printing Lists

By default, the list type does a **repr** on all items, and adds brackets and commas as necessary. In other words, for built-in types, the printed list looks like the corresponding list display:

```
print [1, 2, 3] # prints [1, 2, 3]
```

To control formatting, use the string **join** method, combined with either **map** or a list comprehension or generator expression.

```
print "".join(L) # if all items are strings

print ", ".join(map(str, L))

print "|".join(str(v) for v in L if v > 0)
```

To print a list of string fragments to a file, you can use **writelines** instead of **write**:

```
sys.stdout.writelines(L) # if all items are strings
```

Performance Notes

The list object consists of two internal parts; one object header, and one separately allocated array of object references. The latter is reallocated as necessary.

The list has the following performance characteristics:

- The list object stores pointers to objects, not the actual objects themselves. The size of a list in memory depends on the number of objects in the list, not the size of the objects.
- The time needed to get or set an individual item is constant, no matter what the size of the list is (also known as “O(1)” behaviour).
- The time needed to append an item to the list is “amortized constant”; whenever the list needs to allocate more memory, it allocates room for a few items more than it actually needs, to avoid having to reallocate on each call (this assumes that the memory allocator is fast; for huge lists, the allocation overhead may push the behaviour towards O(n*n)).
- The time needed to insert an item depends on the size of the list, or more exactly, how many items that are to the right of the inserted item (O(n)). In other words, inserting items at the end is fast, but inserting items at the beginning can be relatively slow, if the list is large.
- The time needed to remove an item is about the same as the time needed to insert an item at the same location; removing items at the end is fast, removing items at the beginning is slow.
- The time needed to reverse a list is proportional to the list size (O(n)).
- The time needed to sort a list varies; the worst case is O(n log n), but typical cases are often a lot better than that.

Last Updated: November 2006