List

All Versions

Examples

List methods and supported operators

Starting with a given list `a`:

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` – appends a new element to the end of the list.

   ```
   # Append values 6, 7, and 7 to the list
   a.append(6)
   a.append(7)
   a.append(7)
   # a: [1, 2, 3, 4, 5, 6, 7, 7]

   # Append another list
   b = [8, 9]
   a.append(b)
   # a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]

   # Append an element of a different type, as list elements do not need to have the same type
   my_string = "hello world"
   a.append(my_string)
   # a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
   ```

   Note that the `append()` method only appends one new element to the end of the list. If you append a list to another list, the list that you append becomes a single element at the end of the first list.

   ```
   # Appending a list to another list
   a = [1, 2, 3, 4, 5, 6, 7, 7]
   b = [8, 9]
   a.append(b)
   # a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
   a[8]
   # Returns: [8,9]
   ```

2. `extend(enumerable)` – extends the list by appending elements from another enumerable.

   ```
   a = [1, 2, 3, 4, 5, 6, 7, 7]
   b = [8, 9, 10]

   # Extend list by appending all elements from b
   a.extend(b)
   # a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

   # Extend list with elements from a non-list enumerable:
   a.extend(range(3))
   # a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
   ```

   Lists can also be concatenated with the `+` operator. Note that this does not modify any of the original lists:

   ```
   a = [1, 2, 3, 4, 5, 6] + [7, 7] + b
   # a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
   ```

3. `index(value, [startIndex])` – gets the index of the first occurrence of the input value. If the input value is not in the list a `ValueError` exception is raised. If a second argument is provided, the search is started at that specified index.

   ```
   a.index(7)
   # Returns: 6

   a.index(49) # ValueError, because 49 is not in a.

   a.index(7, 7)
   # Returns: 7

   a.index(7, 8) # ValueError, because there is no 7 starting at index 8
   ```

4. `insert(index, value)` – inserts `value` just before the specified `index`. Thus after the insertion the new element occupies position `index`.

   ```
   a.insert(0, 0)  # insert 0 at position 0
   a.insert(2, 5)  # insert 5 at position 2
   # a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
   ```

5. pop([index]) – removes and returns the item at index . With no argument it removes and returns the last element of the list.

```
a.pop(2)
# Returns: 5
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
a.pop(8)
# Returns: 7
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# With no argument:
a.pop()
# Returns: 10
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. remove(value) – removes the first occurrence of the specified value. If the provided value cannot be found, a ValueError is raised.

```
a.remove(0)
a.remove(9)
# a: [1, 2, 3, 4, 5, 6, 7, 8]
a.remove(10)
# ValueError, because 10 is not in a
```

7. reverse() – reverses the list in-place and returns None .

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

There are also other ways of reversing a list .

8. count(value) – counts the number of occurrences of some value in the list.

```
a.count(7)
# Returns: 2
```

9. sort() – sorts the list in numerical and lexicographical order and returns None .

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Lists can also be reversed when sorted using the reverse=True flag in the sort() method.

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

If you want to sort by attributes of items, you can use the key keyword argument:

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

In case of list of dicts the concept is the same:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'height': 175},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'height': 180},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'height': 185}]

l.sort(key=lambda item: item['name'])
# l: [Chuck Norris, John Cena, Jon Skeet]
```

```
l.sort(key=lambda item: item['birthday'])
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Sort by sub dict :

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'size': {'height': 175, 'w
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'size' : {'height': 180, '
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'size': {'height': 185, 'weig

l.sort(key=lambda item: item['size']['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Better way to sort using attrgetter and itemgetter

Lists can also be sorted using attrgetter and itemgetter functions from the operator module. These can help improve readability and reusability. Here are some examples,

```
from operator import itemgetter,attrgetter

people = [{'name':'chandan','age':20,'salary':2000},
          {'name':'chetan','age':18,'salary':5000},
          {'name':'guru','age':30,'salary':3000}]
by_age = itemgetter('age')
by_salary = itemgetter('salary')

people.sort(key=by_age) #in-place sorting by age
people.sort(key=by_salary) #in-place sorting by salary
```

itemgetter can also be given an index. This is helpful if you want to sort based on indices of a tuple.

```
list_of_tuples = [(1,2), (3,4), (5,0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[(5, 0), (1, 2), (3, 4)]
```

Use the attrgetter if you want to sort by attributes of an object,

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175),
           Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
           Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] #reusing Person class from above

person.sort(key=attrgetter('name')) #sort by name
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #sort by birthday
```

10. clear() – removes all items from the list

```
a.clear()
# a = []
```

11. **Replication** – multiplying an existing list by an integer will produce a larger list consisting of that many copies of the original. This can be useful for example for list initialization:

```
b = ["blah"] * 3
# b = ["blah", "blah", "blah"]
b = [1, 3, 5] * 5
# [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Take care doing this if your list contains references to objects (eg a list of lists), see Common Pitfalls - List multiplication and common references .

12. **Element deletion** – it is possible to delete multiple elements in the list using the del keyword and slice notation:

```
a = list(range(10))
del a[::2]
# a = [1, 3, 5, 7, 9]
del a[-1]
# a = [1, 3, 5, 7]
del a[:]
# a = []
```

13. **Copying**

The default assignment "=" assigns a reference of the original list to the new name. That is, the original name and new name are both pointing to the same list object. Changes made through any of them will be reflected in another. This is often not what you intended.

```
b = a
a.append(6)
# b: [1, 2, 3, 4, 5, 6]
```

If you want to create a copy of the list you have below options.

You can slice it:

```
new_list = old_list[:]
```

You can use the built in list() function:

```
new_list = list(old_list)
```

You can use generic copy.copy():

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

This is a little slower than list() because it has to find out the datatype of old_list first.

If the list contains objects and you want to copy them as well, use generic copy.deepcopy():

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Obviously the slowest and most memory-needing method, but sometimes unavoidable.

Python 3.x ≥3.0

copy() – Returns a shallow copy of the list

```
aa = a.copy()
# aa = [1, 2, 3, 4, 5]
```

Accessing list values

Python lists are zero-indexed, and act like arrays in other languages.

```
lst = [1, 2, 3, 4]
lst[0]  # 1
lst[1]  # 2
```

Attempting to access an index outside the bounds of the list will raise an IndexError .

```
lst[4]  # IndexError: list index out of range
```

Negative indices are interpreted as counting from the *end* of the list.

```
lst[-1]  # 4
lst[-2]  # 3
lst[-5]  # IndexError: list index out of range
```

This is functionally equivalent to

```
lst[len(lst)-1]  # 4
```

Lists allow to use *slice notation* as lst[start:end:step] . The output of the slice notation is a new list containing elements from index start to end-1 . If options are omitted start defaults to beginning of list, end to end of list and step to 1:

```
lst[1:]      # [2, 3, 4]
lst[:3]      # [1, 2, 3]
lst[::2]     # [1, 3]
lst[::-1]    # [4, 3, 2, 1]
lst[-1:0:-1] # [4, 3, 2]
lst[5:8]     # [] since starting index is greater than length of lst, returns empty list
lst[1:10]    # [2, 3, 4] same as omitting ending index
```

With this in mind, you can print a reversed version of the list by calling

```
lst[::-1]    # [4, 3, 2, 1]
```

When using step lengths of negative amounts, the starting index has to be greater than the ending index otherwise the result will be an empty list.

```
lst[3:1:-1] # [4, 3]
```

Using negative step indices are equivalent to the following code:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

The indices used are 1 less than those used in negative indexing and are reversed.

Advanced slicing

When lists are sliced the `__getitem__()` method of the list object is called, with a `slice` object. Python has a builtin slice method to generate slice objects. We can use this to *store* a slice and reuse it later like so,

```
data = 'chandan purohit    22 2000'  #assuming data fields of fixed length
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)

#now we can have more readable slices
print(data[name_slice]) #chandan purohit
print(data[age_slice]) #'22'
print(data[salary_slice]) #'2000'
```

This can be of great use by providing slicing functionality to our objects by overriding `__getitem__` in our class.

Any and All

You can use `all()` to determine if all the values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Likewise, `any()` determines if one or more values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

While this example uses a list, it is important to note these built-ins work with any iterable, including generators.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Checking if list is empty

The emptiness of a list is associated to the boolean `False` , so you don't have to check `len(lst) == 0` , but just `lst` or `not lst`

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

Checking whether an item is in a list

Python makes it very simple to check whether an item is in a list. Simply use the in operator.

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

Note: the in operator on sets is asymptotically faster than on lists. If you need to use it many times on potentially large lists, you may want to convert your list to a set , and test the presence of elements on the set .

```
slst = set(lst)
'test' in slst
# Out: True
```

Iterating over a list

Python supports using a for loop directly on a list:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# Output: foo
# Output: bar
# Output: baz
```

You can also get the position of each item at the same time:

```
for (index, item) in enumerate(my_list):
    print('The item in position {} is: {}'.format(index, item))

# Output: The item in position 0 is: foo
# Output: The item in position 1 is: bar
# Output: The item in position 2 is: baz
```

The other way of iterating a list based on the index value:

```
for i in range(0,len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
baz
```

Note that changing items in a list while iterating on it may have unexpected results:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

In this last example, we deleted the first item at the first iteration, but that caused bar to be skipped.

Concatenate and Merge lists

1. **The simplest way to concatenate** list1 and list2 :

   ```
   merged = list1 + list2
   ```

2. **zip returns a list of tuples** , where the i-th tuple contains the i-th element from each of the argument sequences or iterables:

   ```
   alist = ['a1', 'a2', 'a3']
   blist = ['b1', 'b2', 'b3']

   for a, b in zip(alist, blist):
   ```

```
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3
```

If the lists have different lengths then the result will include only as many elements as the shortest one:

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3', 'b4']
for a, b in zip(alist, blist):
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3

alist = []
len(list(zip(alist, blist)))

# Output:
# 0
```

For padding lists of unequal length to the longest one with None s use itertools.zip_longest (itertools.izip_longest in Python 2)

```
alist = ['a1', 'a2', 'a3']
blist = ['b1']
clist = ['c1', 'c2', 'c3', 'c4']

for a,b,c in itertools.zip_longest(alist, blist, clist):
    print(a, b, c)

# Output:
# a1 b1 c1
# a2 None c2
# a3 None c3
# None None c4
```

3. **Insert to a specific index values:**

```
alist = [123, 'xyz', 'zara', 'abc']
alist.insert(3, [2009])
print("Final List :", alist)
```

Output:

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Length of a list

Use len() to get the one-dimensional length of a list.

```
len(['one', 'two'])  # returns 2

len(['one', [2, 3], 'four'])  # returns 3, not 4
```

len() also works on strings, dictionaries, and other data structures similar to lists.

Note that len() is a built-in function, not a method of a list object.

Also note that the cost of len() is O(1) , meaning it will take the same amount of time to get the length of a list regardless of its length.

Remove duplicate values in list

Removing duplicate values in a list can be done by converting the list to a set (that is an unordered collection of distinct objects). If a list data structure is needed, then the set can be converted back to a list using the function list() :

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Note that by converting a list to a set the original ordering is lost.

To preserve the order of the list one can use an OrderedDict

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

Reversing list elements

You can use the reversed function which returns an iterator to the reversed list:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Note that the list "numbers" remains unchanged by this operation, and remains in the same order it was originally.

To reverse in place, you can also use the reverse method .

You can also reverse a list (actually obtaining a copy, the original list is unaffected) by using the slicing syntax, setting the third argument (the step) as -1:

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
Out[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Comparison of lists

It's possible to compare lists and other sequences lexicographically using comparison operators. Both operands must be of the same type.

```
[1, 10, 100] < [2, 10, 100]
# True, because 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, because the lists are equal
[1, 10, 100] <= [1, 10, 100]
# True, because the lists are equal
[1, 10, 100] < [1, 10, 101]
# True, because 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, because 0 < 1
```

If one of the lists is contained at the start of the other, the shortest list wins.

```
[1, 10] < [1, 10, 100]
# True
```

Accessing values in nested list

Starting with a three-dimensional list:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Accessing items in the list:

```
print(alist[0][0][1])
#2
#Accesses second element in the first list in the first list

print(alist[1][1][2])
#10
#Accesses the third element in the second list in the second list
```

Performing support operations:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
```

```
#Appends 11 to the end of the first list in the first list
```

Using nested for loops to print the list:

```
for row in alist: #One way to loop through nested lists
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

Note that this operation can be used in a list comprehension or even as a generator to produce efficiencies, e.g.:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Not all items in the outer lists have to be lists themselves:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Another way to use nested for loops. The other way is better but I've needed to use this on occasion:

```
for row in range(len(alist)): #A less Pythonic way to loop through lists
    for col in range(len(alist[row])):
       print(alist[row][col])

#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
#[12, 13, 14]
```

Using slices in nested list:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

The final list:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

Initializing a List to a Fixed Number of Elements

For **immutable** elements (e.g. `None`, string literals etc.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

For **mutable** elements, the same construct will result in all elements of the list referring to the same object, for example, for a set:

```
>>> my_list=[{1}] * 10
>>> print(my_list)
[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]
>>> my_list[0].add(2)
>>> print(my_list)
[{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}]
```

Instead, to initialize the list with a fixed number of **different mutable** objects, use:

```
my_list=[{1} for _ in range(10)]
```

Syntax

```
[value, value, ...]
```

```
list([iterable])
```

Parameters

Remarks

`list` is a particular type of iterable, but it is not the only one that exists in Python. Sometimes it will be better to use set, tuple, or dictionary

`list` is the name given in Python to dynamic arrays (similar to `vector<void*>` from C++ or Java's `ArrayList<Object>`). It is not a linked-list.

Accessing elements is done in constant time and is very fast. Appending elements to the end of the list is amortized constant time, but once in a while it might involve allocation and copying of the whole `list`.

List comprehensions are related to lists.