4.1. Lists

Contents

- Lists
  - Quick example
  - Difference between `append()` and `extend()`
  - Other list methods
  - Operators
  - Slicing
  - List comprehension
  - Filtering Lists
  - Lists as Stacks
  - Lists as Queues
  - How to copy a list

In python, **lists** are part of the standard language. You will find them everywhere. Like almost everything in Python, lists are objects. There are many methods associated to them. Some of which are presented here below.

4.1.1. Quick example

```
>>> l = [1, 2, 3]
>>> l[0]
1
>>> l.append(1)
>>> l
[1, 2, 3, 1]
```

4.1.2. Difference between `append()` and `extend()`

Lists have several methods amongst which the `append()` and `extend()` methods. The former appends an object to the end of the list (e.g., another list) while the later appends each element of the iterable object (e.g.,

SEARCH

Go

Enter search terms or a module, class or function name.

This page: Show source.

TABLE OF CONTENTS

1. Quick Start / Tutorial
2. Variables, expressions, statements, types
1. Iterators
2. Generators
3. Exceptions
4. Decorators
5. Classes
6. Namespace and scoping rules
7. Packaging
8. Notes about sorting lists and dictionaries
9. Notes about booleans and

For example, we can append an object (here the character ‘c’) to the end of a simple list as follows:

```
>>> stack = ['a','b']
>>> stack.append('c')
>>> stack
['a', 'b', 'c']
```

However, if we want to append several objects contained in a list, the result as expected (or not...) is:

```
>>> stack.append(['d', 'e', 'f'])
>>> stack
['a', 'b', 'c', ['d', 'e', 'f']]
```

The object `['d', 'e', 'f']` has been appended to the exiistng list. However, it happens that sometimes what we want is to append the elements one by one of a given list rather the list itself. You can do that manually of course, but a better solution is to use the **`extend()`** method as follows:

```
>>> stack = ['a', 'b', 'c']
>>> stack.extend(['d', 'e','f'])
>>> stack
['a', 'b', 'c', 'd', 'e', 'f']
```

4.1.3. Other list methods

4.1.3.1. index

The **`index()`** methods searches for an element in a list. For instance:

```
>>> my_list = ['a','b','c','b', 'a']
>>> my_list.index('b')
1
```

It returns the index of the first and only occurence of ‘b’. If you want to specify a range of valid index, you can indicate the start and stop indices:

```
>>> my_list = ['a','b','c','b', 'a']
>>> my_list.index('b', 2)
3
```

Warning

if the element is not found, an error is raised

SEARCH

Go

Enter search terms or a module, class or function name.

This page: Show source.

TABLE OF CONTENTS

1. Quick Start / Tutorial

2. Variables, expressions, statements, types

1. Iterators

2. Generators

3. Exceptions

4. Decorators

5. Classes

6. Namespace and scoping rules

7. Packaging

8. Notes about sorting lists and dictionaries

9. Notes about booleans and

You can remove element but also insert element wherever you want in a list:

```
>>> my_list.insert(2, 'a')
>>> my_list
['b', 'c', 'a', 'b']
```

The `insert()` methods insert an object before the index provided.

4.1.3.3. remove

Similarly, you can remove the first occurence of an element as follows:

```
>>> my_list.remove('a')
>>> my_list
['b', 'c', 'b', 'a']
```

4.1.3.4. pop

Or remove the last element of a list by using:

```
>>> my_list.pop()
'a'
>>> my_list
['b', 'c', 'b']
```

which also returns the value that has been removed.

4.1.3.5. count

You can count the number of element of a kind:

```
>>> my_list.count('b')
2
```

4.1.3.6. sort

There is a `sort()` method that performs an in-place sorting:

```
>>> my_list.sort()
>>> my_list
['a', 'b', 'b', 'c']
```

Here, it is quite simple since the elements are all characters. For standard types, the sorting works well. Imagine now that you have some non-standard types. You can overwrite the function used to perform the comparison as the first argument of the `sort()` method.

SEARCH

Go

Enter search terms or a module, class or function name.

This page: Show source.

TABLE OF CONTENTS

1. Quick Start / Tutorial

2. Variables, expressions, statements, types

1. Iterators

2. Generators

3. Exceptions

4. Decorators

5. Classes

6. Namespace and scoping rules

7. Packaging

8. Notes about sorting lists and dictionaries

9. Notes about booleans and

```
>>> my_list.sort(reverse=True)
>>> my_list
['c', 'b', 'b', 'a']
```

4.1.3.7. reverse

Finally, you can reverse the element in-place:

```
>>> my_list = ['a', 'c' ,'b']
>>> my_list.reverse()
>>> my_list
['b', 'c', 'a']
```

4.1.4. Operators

The + operator can be used to **extend** a list:

```
>>> my_list = [1]
>>> my_list += [2]
>>> my_list
[1, 2]
>>> my_list += [3, 4]
>>> my_list
[1, 2, 3, 4]
```

The * operator ease the creation of list with similar values

```
>>> my_list = [1, 2]
>>> my_list = my_list * 2
>>> my_list
[1, 2, 1, 2]
```

4.1.5. Slicing

Slicing uses the symbol *:* to access to part of a list:

```
>>> list[first index:last index:step]
>>> list[:]
```

```
>>> a = [0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5]
>>> a[2:]
[2, 3, 4, 5]
>>> a[:2]
[0, 1]
>>> a[2:-1]
[2, 3, 4]
```

By default the first index is 0, the last index is the last one..., and the step is 1. The step is optional. So the

SEARCH

Go

Enter search terms or a module, class or function name.

This page: Show source.

TABLE OF CONTENTS

1. Quick Start / Tutorial

2. Variables, expressions, statements, types

1. Iterators

2. Generators

3. Exceptions

4. Decorators

5. Classes

6. Namespace and scoping rules

7. Packaging

8. Notes about sorting lists and dictionaries

9. Notes about booleans and

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8]
>>> a[:]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> a[::1]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> a[0::1]
[1, 2, 3, 4, 5, 6, 7, 8]
```

4.1.6. List comprehension

Traditionally, a piece of code that loops over a sequence could be written as follows:

```
>>> evens = []
>>> for i in range(10):
...     if i % 2 == 0:
...         evens.append(i)
>>> evens
[0, 2, 4, 6, 8]
```

This may work, but it actually makes things slower for Python because the interpreter works on each loop to determine what part of the sequence has to be changed.

A **list comprehension** is the correct answer:

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

Besides the fact that it is more efficient, it is also shorter and involves fewer elements.

4.1.7. Filtering Lists

```
>>> li = [1, 2]
>>> [elem*2 for elem in li if elem>1]
[4]
```

4.1.8. Lists as Stacks

The Python documentation gives an example of how to use lists as stacks, that is a last-in, first-out data structures (**LIFO**).

An item can be added to a list by using the `append()` method. The last item can be removed from the list by using the `pop()` method without passing any index to it.

SEARCH

Go

Enter search terms or a module, class or function name.

This page: Show source.

TABLE OF CONTENTS

1. Quick Start / Tutorial

2. Variables, expressions, statements, types

1. Iterators

2. Generators

3. Exceptions

4. Decorators

5. Classes

6. Namespace and scoping rules

7. Packaging

8. Notes about sorting lists and dictionaries

9. Notes about booleans and

```
['a', 'b', 'c', 'd', 'e', 'f']
>>> stack.pop()
'f'
>>> stack
['a, 'b', 'c', 'd', 'e']
```

4.1.9. Lists as Queues

Another usage of list, again presented in Python documentation is to use lists as queues, that is a first in - first out (**FIFO**).

```
>>> queue = ['a', 'b', 'c', 'd']
>>> queue.append('e')
>>> queue.append('f')
>>> queue
['a', 'b', 'c', 'd', 'e', 'f']
>>> queue.pop(0)
'a'
```

4.1.10. How to copy a list

There are three ways to copy a list:

```
>>> l2 = list(l)
>>> l2 = l[:]
>>> import copy
>>> l2 = copy.copy(l)
```

Warning

Don’t do l2 = l, which is a reference, not a copy.

The preceding techniques for copying a list create *shallow copies*. IT means that nested objects will not be copied. Consider this example:

```
>>> a = [1, 2, [3, 4]]
>>> b = a[:]
>>> a[2][0] = 10
>>> a
[1, 2, [10, 4]]
>>> b
[1, 2, [10, 4]]
```

To get around this problem, you must perform a deep copy:

```
>>> import copy
>>> a = [1, 2, [3, 4]]
>>> b = copy.deepcopy(a)
```

SEARCH

Go

Enter search terms or a module, class or function name.

This page: Show source.

TABLE OF CONTENTS

1. Quick Start / Tutorial

2. Variables, expressions, statements, types

1. Iterators

2. Generators

3. Exceptions

4. Decorators

5. Classes

6. Namespace and scoping rules

7. Packaging

8. Notes about sorting lists and dictionaries

9. Notes about booleans and

```
>>> b
[1, 2, [3, 4]]
```

4.1.10.1. Inserting items into a sorted list

The bisect module provides tools to manipulate sorted lists.

```
>>> x = [4, 1]
>>> x.sort()
>>> import bisect
>>> bisect.insort(x, 2)
>>> x
[1, 2, 4]
```

To know where the index where the value would have been inserted, you could have use:

```
>>> x = [4, 1]
>>> x.sort()
>>> import bisect
>>> bisect.bisect(x, 2)
2
```

TABLE OF CONTENTS

1. Quick Start / Tutorial

2. Variables, expressions, statements, types

1. Iterators

2. Generators

3. Exceptions

4. Decorators

5. Classes

6. Namespace and scoping rules

7. Packaging

8. Notes about sorting lists and dictionaries

9. Notes about booleans and