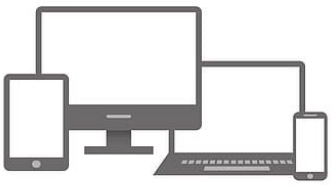

TechCure

Cure your Web

Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**`(`*`x`*`)`

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

`list.`**`extend`**(*`iterable`*)

Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

`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 equal to *x*. It raises a `ValueError` 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.`**`clear`**()

Remove all items from the list. Equivalent to `del a[:]`.

`list.`**`index`**(*`x`*[, *`start`*[, *`end`*]])

Return zero-based index in the list of the first item whose value is equal to *x*. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

`list.`**`count`**(*x*)

Return the number of times *x* appears in the list.

`list.`**`sort`**(*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.`**`copy`**()

Return a shallow copy of the list. Equivalent to `a[:]`.

An example that uses most of the list methods:

```
>>>
```

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple',
'banana']

>>> fruits.count('apple')

2

>>> fruits.count('tangerine')

0

>>> fruits.index('banana')

3

>>> fruits.index('banana', 4)  # Find next banana starting a
position 4

6

>>> fruits.reverse()

>>> fruits

['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']

>>> fruits.append('grape')

>>> fruits

['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange',
'grape']

>>> fruits.sort()

>>> fruits

['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange',
'pear']

>>> fruits.pop()

'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. 1 This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and *None* can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

5.1.1. 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]
```

5.1.2. 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'])
```

Lorem ipsum dolor sit amet

THIS IS A HEADER

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan.

> Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore.
>
> -Wendy Writer

LOREM IPSUM. DOLOR SIT.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

nisl ut aliquip ex ea commodo consequat.

Lorem ipsum dolor sit amet