Searching

All Versions

Examples

Searching for an element

All built-in collections in Python implement a way to check element membership using `in` .

List

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist   # True
10 in alist  # False
```

Tuple

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple    # False
'4' in atuple  # True
```

String

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

Set

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset  # True
10 in aset        # False
```

Dict

`dict` is a bit special: the normal `in` only checks the *keys* . If you want to search in *values* you need to specify it. The same if you want to search for *key-value* pairs.

```
adict = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
1 in adict                 # True   - implicitly searches in keys
'a' in adict               # False
2 in adict.keys()          # True   - explicitly searches in keys
'a' in adict.values()      # True   - explicitly searches in values
(0, 'a') in adict.items()  # True   - explicitly searches key/value pairs
```

Getting the index for sorted sequences: bisect.bisect_left()

Sorted sequences allow the use of faster searching algorithms: `bisect.bisect_left()` [1] :

```
import bisect

def index_sorted(sorted_seq, value):
    """Locate the leftmost value exactly equal to x or raise a ValueError"""
    i = bisect.bisect_left(sorted_seq, value)
    if i != len(sorted_seq) and sorted_seq[i] == value:
        return i
    raise ValueError

alist = [i for i in range(1, 100000, 3)] # Sorted list from 1 to 100000 with step 3
index_sorted(alist, 97285) # 32428
index_sorted(alist, 4)     # 1
index_sorted(alist, 97286)
```

ValueError

For very large **sorted sequences** the speed gain can be quite high. In case for the first search approximatly 500 times as fast:

```
%timeit index_sorted(alist, 97285)
# 100000 loops, best of 3: 3 µs per loop
%timeit alist.index(97285)
# 1000 loops, best of 3: 1.58 ms per loop
```

While it's a bit slower if the element is one of the very first:

```
%timeit index_sorted(alist, 4)
# 100000 loops, best of 3: 2.98 µs per loop
%timeit alist.index(4)
# 1000000 loops, best of 3: 580 ns per loop
```

Getting the index for strings: str.index(), str.rindex() and str.find(), str.rfind()

`String` also have an `index` method but also more advanced options and the additional `str.find`. For both of these there is a complementary *reversed* method.

```
astring = 'Hello on StackOverflow'
astring.index('o')  # 4
astring.rindex('o') # 20

astring.find('o')   # 4
astring.rfind('o')  # 20
```

The difference between `index` / `rindex` and `find` / `rfind` is what happens if the substring is not found in the string:

```
astring.index('q') # ValueError: substring not found
astring.find('q')  # -1
```

All of these methods allow a start and end index:

```
astring.index('o', 5)    # 6
astring.index('o', 6)    # 6 - start is inclusive
astring.index('o', 5, 7) # 6
astring.index('o', 5, 6) #  - end is not inclusive
```

ValueError: substring not found

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right

astring.rindex('o', 4, 7) # 6
```

Getting the index list and tuples: list.index(), tuple.index()

`list` and `tuple` have an `index`-method to get the position of the element:

```
alist = [10, 16, 26, 5, 2, 19, 105, 26]
# search for 16 in the list
alist.index(16) # 1
alist[1]        # 16

alist.index(15)
```

ValueError: 15 is not in list

But only returns the position of the first found element:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26)   # 2
atuple[2]          # 26
atuple[7]          # 26 - is also 26!
```

Searching in custom classes: __contains__ and __iter__

To allow the use of `in` for custom classes the class must either provide the magic method `__contains__` or, failing that, an `__iter__`-method.

Suppose you have a class containing a `list` of `list`s:

```
class ListList:
    def __init__(self, value):
        self.value = value
        # Create a set of all values for fast access
        # Create a set of all values for fast access
        self.setofvalues = set(item for sublist in self.value for item in sublist)

    def __iter__(self):
        print('Using __iter__.')
        # A generator over all sublist elements
        return (item for sublist in self.value for item in sublist)

    def __contains__(self, value):
        print('Using __contains__.')
        # Just lookup if the value is in the set
        return value in self.setofvalues

        # Even without the set you could use the iter method for the contains-check:
        # return any(item == value for item in iter(self))
```

Using membership testing is possible using `in` :

```
a = ListList([[1,1,1],[0,1,1],[1,5,1]])
10 in a    # False
# Prints: Using __contains__.
5 in a     # True
# Prints: Using __contains__.
```

even after deleting the `__contains__` method:

```
del ListList.__contains__
5 in a     # True
# Prints: Using __iter__.
```

Note: The looping `in` (as in `for i in a`) will always use `__iter__` even if the class implements a `__contains__` method.

Searching key(s) for a value in dict

`dict` have no builtin method for searching a value or key because *dictionaries* are unordered. You can create a function that gets the key (or keys) for a specified value:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[key] == value:
            foundkeys.append(key)
    return foundkeys
```

This could also be written as an equivalent list comprehension:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

If you only care about one found key:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

The first two functions will return a `list` of all `keys` that have the specified value:

```
adict = {'a': 10, 'b': 20, 'c': 10}
getKeysForValue(adict, 10)     # ['c', 'a'] - order is random could as well be ['a', 'c']
getKeysForValueComp(adict, 10) # ['c', 'a'] - dito
getKeysForValueComp(adict, 20) # ['b']
getKeysForValueComp(adict, 25) # []
```

The other one will only return one key:

```
getOneKeyForValue(adict, 10)   # 'c'  - depending on the circumstances this could also be 'a'
getOneKeyForValue(adict, 20)   # 'b'
```

and raise a `StopIteration` - `Exception` if the value is not in the `dict` :

```
getOneKeyForValue(adict, 25)
```

```
StopIteration
```

Searching nested sequences

Searching in nested sequences like a list of tuple requires an approach like searching the keys for values in dict but needs customized functions.

The index of the outermost sequence if the value was found in the sequence:

```
def outer_index(nested_sequence, value):
    return next(index for index, inner in enumerate(nested_sequence)
                      for item in inner
                      if item == value)

alist_of_tuples = [(4, 5, 6), (3, 1, 'a'), (7, 0, 4.3)]
outer_index(alist_of_tuples, 'a')  # 1
outer_index(alist_of_tuples, 4.3)  # 2
```

or the index of the outer and inner sequence:

```
def outer_inner_index(nested_sequence, value):
    return next((oindex, iindex) for oindex, inner in enumerate(nested_sequence)
                                 for iindex, item in enumerate(inner)
                                 if item == value)

outer_inner_index(alist_of_tuples, 'a') # (1, 2)
alist_of_tuples[1][2]  # 'a'

outer_inner_index(alist_of_tuples, 7)   # (2, 0)
alist_of_tuples[2][0]  # 7
```

In general (*not always*) using next and a **generator expression** with conditions to find the first occurrence of the searched value is the most efficient approach.

Syntax

Parameters

Remarks

All searching algorithms on iterables containing n elements have O(n) complexity. Only specialized algorithms like bisect.bisect_left() can be faster with O(log(n)) complexity.