Sorted List

Sorted Containers is an Apache2 licensed Python sorted collections library, written in pure-Python, and fast as C-extensions. The introduction is the best way to get started.

Sorted list implementations:

- **SortedList**
- **SortedKeyList**

SortedList

class sortedcontainers.**SortedList**(*iterable=None*, *key=None*) [source]

Bases: **MutableSequence**

Sorted list is a sorted mutable sequence.

Sorted list values are maintained in sorted order.

Sorted list values must be comparable. The total ordering of values must not change while they are stored in the sorted list.

Methods for adding values:

- **SortedList.add()**
- **SortedList.update()**
- **SortedList.__add__()**
- **SortedList.__iadd__()**
- **SortedList.__mul__()**
- **SortedList.__imul__()**

Methods for removing values:

- **SortedList.clear()**
- **SortedList.discard()**
- **SortedList.remove()**
- **SortedList.pop()**
- **SortedList.__delitem__()**

Methods for looking up values:

- **SortedList.bisect_left()**
- **SortedList.bisect_right()**
- **SortedList.count()**
- **SortedList.index()**
- **SortedList.__contains__()**
- **SortedList.__getitem__()**

Methods for iterating values:

- **SortedList.irange()**
- **SortedList.islice()**
- **SortedList.__iter__()**
- **SortedList.__reversed__()**

Methods for miscellany:

- **SortedList.copy()**
- **SortedList.__len__()**
- **SortedList.__repr__()**
- **SortedList._check()**
- **SortedList._reset()**

Sorted lists use lexicographical ordering semantics when compared to other sequences.

Some methods of mutable sequences are not supported and will raise not-implemented error.

static **__new__**(*cls*, *iterable=None*, *key=None*) [source]

Create new sorted list or sorted-key list instance.

Optional *key*-function argument will return an instance of subtype **SortedKeyList**.

```
>>> sl = SortedList()
>>> isinstance(sl, SortedList)
True
>>> sl = SortedList(key=lambda x: -x)
>>> isinstance(sl, SortedList)
True
>>> isinstance(sl, SortedKeyList)
True
```

Parameters:
- **iterable** – initial values (optional)
- **key** – function used to extract comparison key (optional)

Returns: sorted list or sorted-key list instance

__init__(*iterable=None*, *key=None*) [source]

Initialize sorted list instance.

Optional *iterable* argument provides an initial iterable of values to initialize the sorted list.

Runtime complexity: *O(n*log(n))*

```
>>> sl = SortedList()
>>> sl
SortedList([])
>>> sl = SortedList([3, 1, 2, 5, 4])
>>> sl
SortedList([1, 2, 3, 4, 5])
```

Parameters: **iterable** – initial values (optional)

add(*value*) [source]

Add *value* to sorted list.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList()
>>> sl.add(3)
>>> sl.add(1)
>>> sl.add(2)
>>> sl
SortedList([1, 2, 3])
```

Parameters: **value** – value to add to sorted list

`update(iterable)` [source]

Update sorted list by adding all values from *iterable*.

Runtime complexity: *O(k*log(n))* – approximate.

```
>>> sl = SortedList()
>>> sl.update([3, 1, 2])
>>> sl
SortedList([1, 2, 3])
```

Parameters: **iterable** – iterable of values to add

`clear()` [source]

Remove all values from sorted list.

Runtime complexity: *O(n)*

`discard(value)` [source]

Remove *value* from sorted list if it is a member.

If *value* is not a member, do nothing.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList([1, 2, 3, 4, 5])
>>> sl.discard(5)
>>> sl.discard(0)
>>> sl == [1, 2, 3, 4]
True
```

Parameters: **value** – *value* to discard from sorted list

`remove(value)` [source]

Remove *value* from sorted list; *value* must be a member.

If *value* is not a member, raise ValueError.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList([1, 2, 3, 4, 5])
>>> sl.remove(5)
>>> sl == [1, 2, 3, 4]
True
>>> sl.remove(0)
Traceback (most recent call last):
  ...
ValueError: 0 not in list
```

Parameters: **value** – *value* to remove from sorted list
Raises: **ValueError** – if *value* is not in sorted list

`pop(index=-1)` [source]

Remove and return value at *index* in sorted list.

Raise `IndexError` if the sorted list is empty or index is out of range.

Negative indices are supported.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList('abcde')
>>> sl.pop()
'e'
>>> sl.pop(2)
'c'
>>> sl
SortedList(['a', 'b', 'd'])
```

Parameters: **index** (*int*) – index of value (default -1)
Returns: value
Raises: **IndexError** – if index is out of range

`bisect_left(value)` [source]

Return an index to insert *value* in the sorted list.

If the *value* is already present, the insertion point will be before (to the left of) any existing values.

Similar to the *bisect* module in the standard library.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList([10, 11, 12, 13, 14])
>>> sl.bisect_left(12)
2
```

Parameters: **value** – insertion index of value in sorted list
Returns: index

`bisect_right(value)` [source]

Return an index to insert *value* in the sorted list.

Similar to *bisect_left*, but if *value* is already present, the insertion point will be after (to the right of) any existing values.

Similar to the *bisect* module in the standard library.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList([10, 11, 12, 13, 14])
>>> sl.bisect_right(12)
3
```

Parameters: **value** – insertion index of value in sorted list
Returns: index

`count(value)` ¶ [source]

Return number of occurrences of *value* in the sorted list.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList([1, 2, 2, 3, 3, 3, 4, 4, 4, 4])
>>> sl.count(3)
3
```

Parameters: **value** – value to count in sorted list
Returns: count

index(*value, start=None, stop=None*) [source]

Return first index of value in sorted list.

Raise ValueError if *value* is not present.

Index must be between *start* and *stop* for the *value* to be considered present. The default value, None, for *start* and *stop* indicate the beginning and end of the sorted list.

Negative indices are supported.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList('abcde')
>>> sl.index('d')
3
>>> sl.index('z')
Traceback (most recent call last):
  ...
ValueError: 'z' is not in list
```

Parameters:
- **value** – value in sorted list
- **start** (*int*) – start index (default None, start of sorted list)
- **stop** (*int*) – stop index (default None, end of sorted list)

Returns: index of value
Raises: **ValueError** – if value is not present

irange(*minimum=None, maximum=None, inclusive=(True, True), reverse=False*) [source]

Create an iterator of values between *minimum* and *maximum*.

Both *minimum* and *maximum* default to *None* which is automatically inclusive of the beginning and end of the sorted list.

The argument *inclusive* is a pair of booleans that indicates whether the minimum and maximum ought to be included in the range, respectively. The default is `(True, True)` such that the range is inclusive of both minimum and maximum.

When *reverse* is *True* the values are yielded from the iterator in reverse order; *reverse* defaults to *False*.

```
>>> sl = SortedList('abcdefghij')
>>> it = sl.irange('c', 'f')
>>> list(it)
['c', 'd', 'e', 'f']
```

Parameters:
- **minimum** – minimum value to start iterating
- **maximum** – maximum value to stop iterating
- **inclusive** – pair of booleans
- **reverse** (*bool*) – yield values in reverse order

Returns: iterator

islice(*start=None, stop=None, reverse=False*) [source]

Return an iterator that slices sorted list from *start* to *stop*.

The *start* and *stop* index are treated inclusive and exclusive, respectively.

Both *start* and *stop* default to *None* which is automatically inclusive of the beginning and end of the sorted list.

When *reverse* is *True* the values are yielded from the iterator in reverse order; *reverse* defaults to *False*.

```
>>> sl = SortedList('abcdefghij')
>>> it = sl.islice(2, 6)
>>> list(it)
['c', 'd', 'e', 'f']
```

Parameters:
- **start** (*int*) – start index (inclusive)
- **stop** (*int*) – stop index (exclusive)
- **reverse** (*bool*) – yield values in reverse order

Returns: iterator

`__iter__()` [source]

Return an iterator over the sorted list.

`sl.__iter__()` <==> `iter(sl)`

Iterating the sorted list while adding or deleting values may raise a **`RuntimeError`** or fail to iterate over all values.

`__reversed__()` [source]

Return a reverse iterator over the sorted list.

`sl.__reversed__()` <==> `reversed(sl)`

Iterating the sorted list while adding or deleting values may raise a **`RuntimeError`** or fail to iterate over all values.

`__contains__(value)` [source]

Return true if *value* is an element of the sorted list.

`sl.__contains__(value)` <==> `value in sl`

Runtime complexity: *O(log(n))*

```
>>> sl = SortedList([1, 2, 3, 4, 5])
>>> 3 in sl
True
```

Parameters: **value** – search for value in sorted list
Returns: true if *value* in sorted list

`__getitem__(index)` [source]

Lookup value at *index* in sorted list.

`sl.__getitem__(index)` <==> `sl[index]`

Supports slicing.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList('abcde')
>>> sl[1]
'b'
>>> sl[-1]
'e'
>>> sl[2:5]
['c', 'd', 'e']
```

Parameters: **index** – integer or slice for indexing
Returns: value or list of values
Raises: **IndexError** – if index out of range

`__delitem__`(*index*) [source]

Remove value at *index* from sorted list.

`sl.__delitem__(index)` <==> `del sl[index]`

Supports slicing.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sl = SortedList('abcde')
>>> del sl[2]
>>> sl
SortedList(['a', 'b', 'd', 'e'])
>>> del sl[:2]
>>> sl
SortedList(['d', 'e'])
```

Parameters: **index** – integer or slice for indexing
Raises: **IndexError** – if index out of range

`__add__`(*other*) [source]

Return new sorted list containing all values in both sequences.

`sl.__add__(other)` <==> `sl + other`

Values in *other* do not need to be in sorted order.

Runtime complexity: *O(n*log(n))*

```
>>> sl1 = SortedList('bat')
>>> sl2 = SortedList('cat')
>>> sl1 + sl2
SortedList(['a', 'a', 'b', 'c', 't', 't'])
```

Parameters: **other** – other iterable
Returns: new sorted list

`__iadd__`(*other*) [source]

Update sorted list with values from *other*.

`sl.__iadd__(other)` <==> `sl += other`

Values in *other* do not need to be in sorted order.

Runtime complexity: *O(k*log(n))* – approximate.

```
>>> sl = SortedList('bat')
>>> sl += 'cat'
>>> sl
SortedList(['a', 'a', 'b', 'c', 't', 't'])
```

Parameters: **other** – other iterable
Returns: existing sorted list

__mul__(*num*) [source]

Return new sorted list with *num* shallow copies of values.

`sl.__mul__(num)` <==> `sl * num`

Runtime complexity: *O(n*log(n))*

```
>>> sl = SortedList('abc')
>>> sl * 3
SortedList(['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c'])
```

Parameters: **num** (*int*) – count of shallow copies
Returns: new sorted list

__imul__(*num*) [source]

Update the sorted list with *num* shallow copies of values.

`sl.__imul__(num)` <==> `sl *= num`

Runtime complexity: *O(n*log(n))*

```
>>> sl = SortedList('abc')
>>> sl *= 3
>>> sl
SortedList(['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c'])
```

Parameters: **num** (*int*) – count of shallow copies
Returns: existing sorted list

__eq__(*other*)

Return true if and only if sorted list is equal to *other*.

`sl.__eq__(other)` <==> `sl == other`

Comparisons use lexicographical order as with sequences.

Runtime complexity: *O(n)*

Parameters: **other** – *other* sequence
Returns: true if sorted list is equal to *other*

__ne__(*other*)

Return true if and only if sorted list is not equal to *other*.

`sl.__ne__(other)` <==> `sl != other`

Comparisons use lexicographical order as with sequences.

Runtime complexity: *O(n)*

Parameters: **other** – *other* sequence
Returns: true if sorted list is not equal to *other*

`__lt__(other)`

Return true if and only if sorted list is less than *other*.

```
sl.__lt__(other) <==> sl < other
```

Comparisons use lexicographical order as with sequences.

Runtime complexity: *O(n)*

Parameters: **other** – *other* sequence
Returns: true if sorted list is less than *other*

`__le__(other)`

Return true if and only if sorted list is less than or equal to *other*.

```
sl.__le__(other) <==> sl <= other
```

Comparisons use lexicographical order as with sequences.

Runtime complexity: *O(n)*

Parameters: **other** – *other* sequence
Returns: true if sorted list is less than or equal to *other*

`__gt__(other)`

Return true if and only if sorted list is greater than *other*.

```
sl.__gt__(other) <==> sl > other
```

Comparisons use lexicographical order as with sequences.

Runtime complexity: *O(n)*

Parameters: **other** – *other* sequence
Returns: true if sorted list is greater than *other*

`__ge__(other)`

Return true if and only if sorted list is greater than or equal to *other*.

```
sl.__ge__(other) <==> sl >= other
```

Comparisons use lexicographical order as with sequences.

Runtime complexity: *O(n)*

Parameters: **other** – *other* sequence
Returns: true if sorted list is greater than or equal to *other*

`copy()` [source]

Return a shallow copy of the sorted list.

Runtime complexity: *O(n)*

Returns: new sorted list

`__len__()` [source]

Return the size of the sorted list.

`sl.__len__()` <==> `len(sl)`

Returns: size of sorted list

`__repr__()` [source]

Return string representation of sorted list.

`sl.__repr__()` <==> `repr(sl)`

Returns: string representation

`_check()` [source]

Check invariants of sorted list.

Runtime complexity: *O(n)*

`_reset(load)` [source]

Reset sorted list load factor.

The *load* specifies the load-factor of the list. The default load factor of 1000 works well for lists from tens to tens-of-millions of values. Good practice is to use a value that is the cube root of the list size. With billions of elements, the best load factor depends on your usage. It's best to leave the load factor at the default until you start benchmarking.

See Implementation Details and Performance at Scale for more information.

Runtime complexity: *O(n)*

Parameters: **load** (*int*) – load-factor for sorted list sublists

`append(value)` [source]

Raise not-implemented error.

Implemented to override *MutableSequence.append* which provides an erroneous default implementation.

Raises: **NotImplementedError** – use `sl.add(value)` instead

`extend(values)` [source]

Raise not-implemented error.

Implemented to override *MutableSequence.extend* which provides an erroneous default implementation.

Raises: **NotImplementedError** – use `sl.update(values)` instead

`insert(index, value)` [source]

Raise not-implemented error.

Raises: **NotImplementedError** – use `sl.add(value)` instead

`reverse()` [source]

Raise not-implemented error.

Sorted list maintains values in ascending sort order. Values may not be reversed in-place.

Use `reversed(sl)` for an iterator over values in descending sort order.

Implemented to override *MutableSequence.reverse* which provides an erroneous default implementation.

Raises: **NotImplementedError** – use `reversed(sl)` instead

`__setitem__(index, value)` [source]

Raise not-implemented error.

`sl.__setitem__(index, value)` <==> `sl[index] = value`

Raises: **NotImplementedError** – use `del sl[index]` and `sl.add(value)` instead

SortedKeyList

class sortedcontainers.**SortedKeyList**(*iterable=None, key=<function identity>*)

Bases: **SortedList** [source]

Sorted-key list is a subtype of sorted list.

The sorted-key list maintains values in comparison order based on the result of a key function applied to every value.

All the same methods that are available in **SortedList** are also available in **SortedKeyList**.

Additional methods provided:

- **SortedKeyList.key**
- **SortedKeyList.bisect_key_left()**
- **SortedKeyList.bisect_key_right()**
- **SortedKeyList.irange_key()**

Some examples below use:

```
>>> from operator import neg
>>> neg
<built-in function neg>
>>> neg(1)
-1
```

`__init__(iterable=None, key=<function identity>)` [source]

Initialize sorted-key list instance.

Optional *iterable* argument provides an initial iterable of values to initialize the sorted-key list.

Optional *key* argument defines a callable that, like the *key* argument to Python's *sorted* function, extracts a comparison key from each value. The default is the identity function.

Runtime complexity: *O(n*log(n))*

```
>>> from operator import neg
>>> skl = SortedKeyList(key=neg)
>>> skl
SortedKeyList([], key=<built-in function neg>)
>>> skl = SortedKeyList([3, 1, 2], key=neg)
```

```
>>> skl
SortedKeyList([3, 2, 1], key=<built-in function neg>)
```

Parameters:
- **iterable** – initial values (optional)
- **key** – function used to extract comparison key (optional)

key

Function used to extract comparison key from values.

bisect_key_left(*key*) [source]

Return an index to insert *key* in the sorted-key list.

If the *key* is already present, the insertion point will be before (to the left of) any existing keys.

Similar to the *bisect* module in the standard library.

Runtime complexity: *O(log(n))* – approximate.

```
>>> from operator import neg
>>> skl = SortedKeyList([5, 4, 3, 2, 1], key=neg)
>>> skl.bisect_key_left(-1)
4
```

Parameters: **key** – insertion index of key in sorted-key list
Returns: index

bisect_key_right(*key*) [source]

Return an index to insert *key* in the sorted-key list.

Similar to *bisect_key_left*, but if *key* is already present, the insertion point will be after (to the right of) any existing keys.

Similar to the *bisect* module in the standard library.

Runtime complexity: *O(log(n))* – approximate.

```
>>> from operator import neg
>>> skl = SortedList([5, 4, 3, 2, 1], key=neg)
>>> skl.bisect_key_right(-1)
5
```

Parameters: **key** – insertion index of key in sorted-key list
Returns: index

irange_key(*min_key=None, max_key=None, inclusive=(True, True), reverse=False*)

Create an iterator of values between *min_key* and *max_key*. [source]

Both *min_key* and *max_key* default to *None* which is automatically inclusive of the beginning and end of the sorted-key list.

The argument *inclusive* is a pair of booleans that indicates whether the minimum and maximum ought to be included in the range, respectively. The default is `(True, True)` such that the range is inclusive of both minimum and maximum.

When *reverse* is *True* the values are yielded from the iterator in reverse order; *reverse* defaults to *False*.

```
>>> from operator import neg
>>> skl = SortedKeyList([11, 12, 13, 14, 15], key=neg)
>>> it = skl.irange_key(-14, -12)
>>> list(it)
[14, 13, 12]
```

Parameters:
- **min_key** – minimum key to start iterating
- **max_key** – maximum key to stop iterating
- **inclusive** – pair of booleans
- **reverse** (*bool*) – yield values in reverse order

Returns: iterator

sortedcontainers.**SortedListWithKey**

alias of **SortedKeyList**