Python Sorted Collections

Grant Jenks
PyCon 2016

1

Hello, [smile], I’m here today to talk about Python sorted collections. I’m excited and a bit nervous to be up here in front of all you smart people. But I’m a pretty smart guy myself and I really admire Python so let’s get started.

Python Sordid Collections

Grant Jenks
PyCon 2016

2

Every time I talk about sorted collections, my wife hear's sordid. We'll see today how many times I can confuse the on-site captioners. You're probably more familiar with sorted collections than you realize.

A Short Argument for Sorted Collections

3

Let me make a short argument for sorted collections types.

8.5. heapq — Heap queue algorithm

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

8.6. bisect — Array bisection algorithm

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion.

17.7. queue — A synchronized queue class

The queue module implements multi-producer, multi-consumer queues.

class `queue.`**`PriorityQueue`**`(`*maxsize=0*`)`

import heapq, bisect, queue

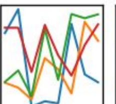
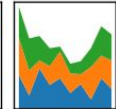
In the standard library we have `heapq`, `bisect` and `queue.PriorityQueue` but they don't quite fill the gap.
Behind the scenes, priority queue uses a heap implementation.
Another common mistake is to think that `collections.OrderedDict` is a dictionary that maintains sort order but that’s not the case.

I don't always import sorted types. But when I do, I expect them in the standard library.

May 2016	Programming Language
1	Java
2	C
3	C++
4	C#
5	Python

TIOBE Index

And here's why. Java, C++ and .NET have them. Python has broken into the top five of the TIOBE index but feels a bit more like PHP or Javascript in this regard.

Third-Party Solutions

We also depend on external solutions: Sqlite in-memory indexes, `pandas.DataFrame` indexes, and Redis sorted sets. If you've ever issued a "zadd" command to Redis then you used a sorted collection.

What are sorted collections types?

8

So what should be the API of sorted collection types in Python?

SortedList

```
class SortedList(collections.MutableSequence):

    def __init__(self, iterable=(), key=None):

        ...

    def bisect(self, value):

        ...
```


Well a SortedList should be a MutableSequence. Pretty close to the “list” API.
But there’s a sort order constraint that must be satisfied by “setitem” and “insert” methods.
Also should support a “key” argument like the “sorted” builtin function.
Given sorted order, “bisect_right” and “bisect_left” methods make sense.
You could also imagine an “add”

method and “discard” method for elements. Kind of like a multi-set in other languages.
I’d also expect “getitem”, “contains”, “count”, etc. to be faster than linear time.

SortedDict

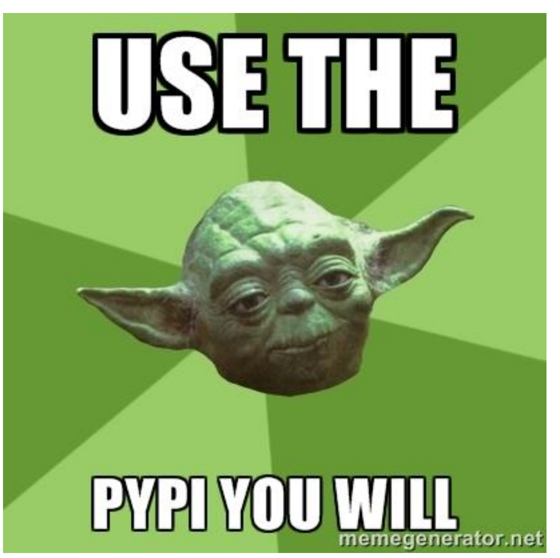
```
class SortedDict(collections.MutableMapping):
    def __init__(self, [key,] *args, **kwargs):
        ...
    def bisect(self, key):
        ...
```


A sorted dictionary should be a MutableMapping. Pretty close to the dictionary API.
But iteration yields items in sorted order.
Also should support efficient positional indexing, something like a SequenceView.

SortedSet

```
class SortedSet(collections.MutableSet, collections.Sequence):

    def __init__(self, iterable=(), key=None):

        ...

    def bisect(self, value):

        ...
```


SortedSet should be a MutableSet. Pretty close to the “set” API. SortedSet should also be a Sequence like the “tuple” API to support efficient positional indexing.



The chorus and the refrain from core developers is: "Look to the PyPI." Which is good advice.

A Brief History Of Sorted Collections

13

So let’s talk about your options with a bit of software archaeology.

blist

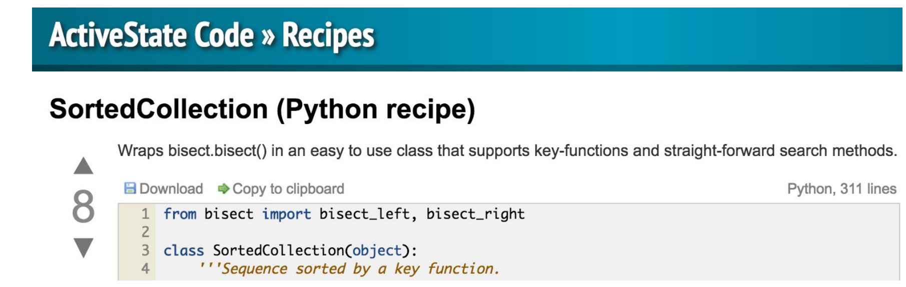
- Daniel Stutzbach; 2006 start, 2014 last PyPI update.
- `blist.blist` B-tree based replacement for `list`.
- Sorted collections based on `blist.blist` type.
- Full-featured, long-standing API.

Blist is the genesis of our story but it wasn't really designed for sorted collections. It's written in C and the innovation here is the "blist" data type. That's a B-tree based replacement for CPython's built-in list. Sorted list, sorted dictionary, and sorted set were built on top of this "blist" data type and it became the incumbent to beat. Also noteworthy is that the API was rather well thought out.

There were some quirks, for example: the “pop” method returns the first element rather than the last element in the sorted list.

sortedcollection

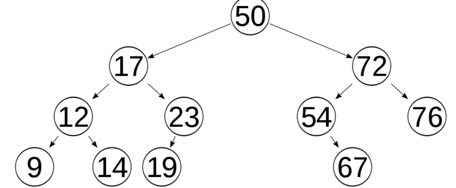
- Raymond Hettinger; published on ActiveState, 2010.
- Linked from the Python Standard Library docs.
- Mostly meant for read-only workloads.



SortedCollection is not a package. You can’t install this with “pip”. It’s simply a Python recipe that Raymond Hettinger linked from the Python docs. Couple innovations here though: it’s simple, it’s written in pure-Python, and maintains a parallel list of keys. So we have efficient support for that key-function parameter.

bintrees

- Manfred Moitzi; 2010 start, 2015 last PyPI update.
- Multiple tree implementations: Binary, AVL, Red-Black.
- API extends `blist` with tree traversal for slicing by value.



16

This is bintrees. Still alive and kicking today. A few innovations here: it’s written with Cython support to improve performance and has a few different tree “backends.” You can create a red-black or AVL tree depending on your needs. There’s also some notion of accessing the nodes themselves and customizing the tree traversal to slice by value rather than by index.

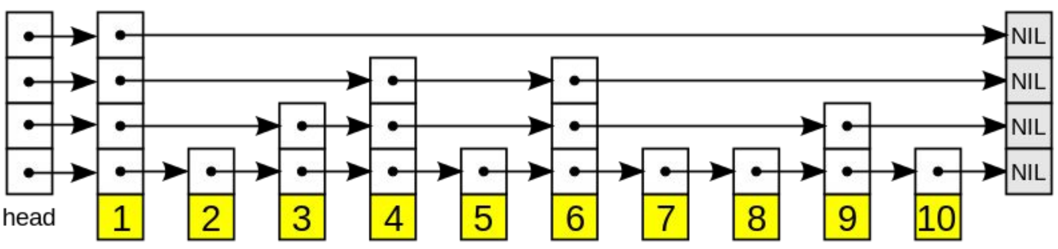
banyan

- Ami Tavory; 2013 start, 2013 last PyPI update.
- Highly optimized C++ implementation.
- Supports tree-augmentation with metadata.

Banyan had a very short life but adds another couple innovations: it’s incredibly fast and achieves that through C++ template meta-programming. It also has a feature called tree-augmentation that will let you store metadata at tree nodes. You can use this for interval trees if you need those.

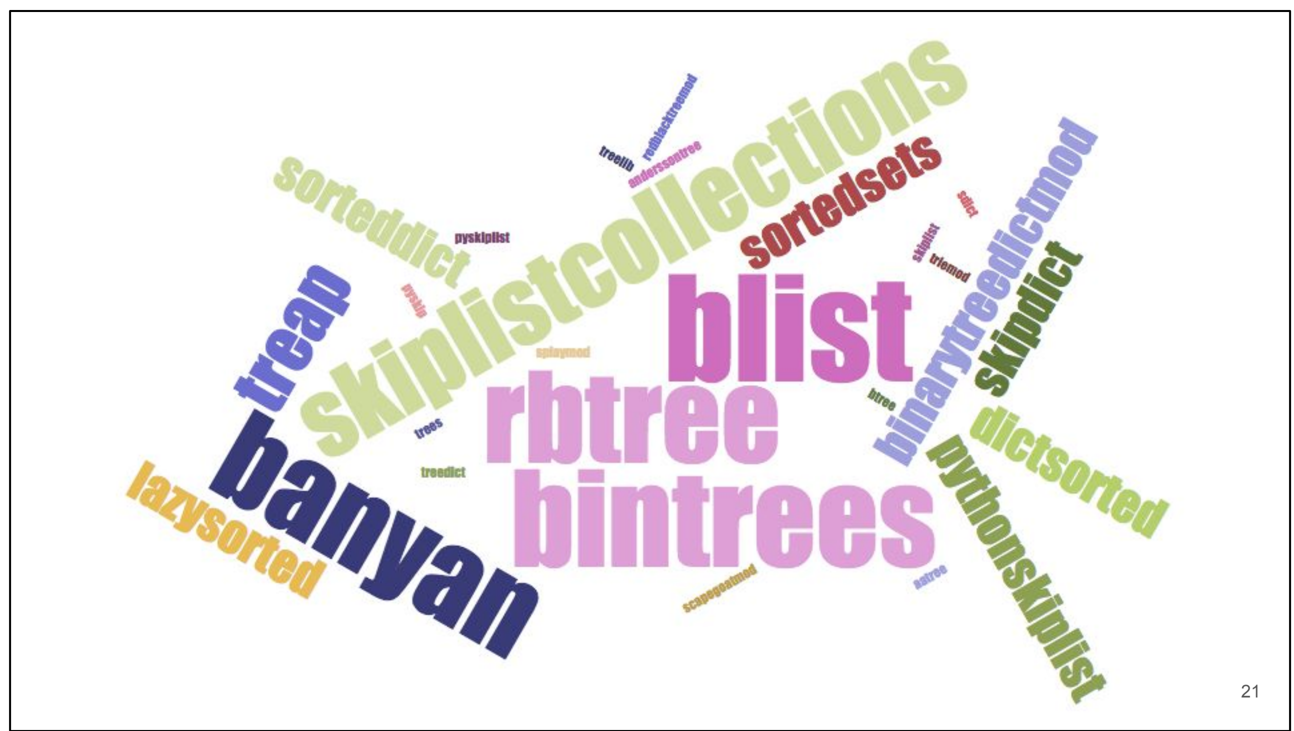
skiplistcollections

- Jakub Stasiak; 2013 start, 2014 last PyPI update.
- Pure-Python with competitive performance.



18

Finally there’s “skiplistcollections.” Couple significant things here: it’s pure-Python but fast, even for large collections, and it uses a skip-list data type rather than a binary tree.



Altogether, you go on PyPI and try to figure this out and it’s kind of like this. It’s a mess. PyPI has really got to work better than using Google with the site operator.

Couple others worth calling out: rbtree is another fast C-based implementation. And there’s a few like treap, splay and scapegoat that are contributions and experiments by Dan

Stromberg. He’s also done some interesting benchmarking of the various tree types. There’s no silver bullet when it comes to trees.

I love Python because there's one right way to do things. If I just want sorted types, what’s the right answer?

The Missing Battery: SortedContainers

20

I couldn't find the right answer so I built it. The missing battery: Sorted Containers.

Star 514

SortedContainers provides sorted container types, written in pure-Python and fast as C-extensions.

Give Support

If you or your organization uses SortedContainers, consider financial support:

SortedContainers

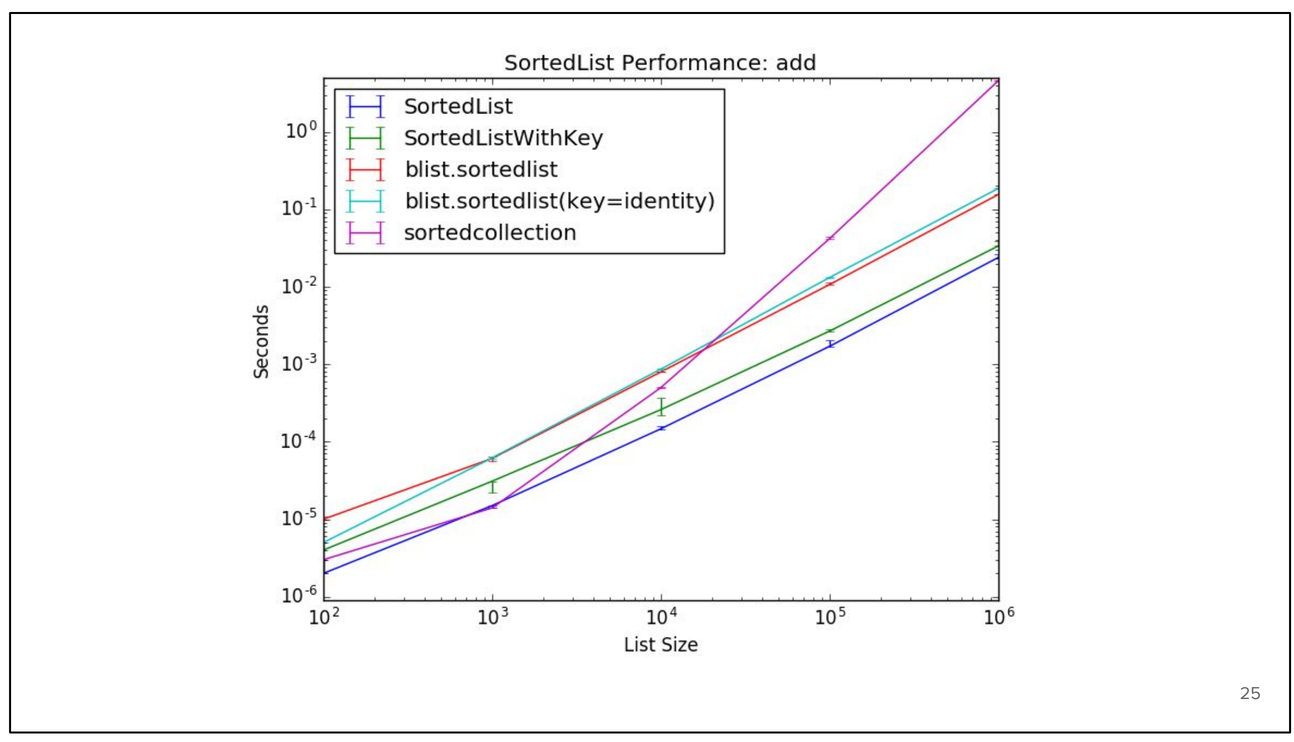
SortedContainers is an *Apache2 Licensed* containers library, written in pure-Python, and fast as C-extensions.

Python's standard library is great until you need a sorted container type. Many will attest that you can get really far without one, but the moment you **really need** a sorted list, dict, or set, you're faced with a dozen different implementations, most using C-extensions without great documentation and benchmarking.

Things shouldn't be this way. Not in Python.

```
>>> sl = sortedcontainers.SortedList(xrange(10000000))
>>> 1234567 in sl
True
>>> sl[7654321]
7654321
>>> sl.add(1234567)
>>> sl.count(1234567)
2
>>> sl *= 3
>>> len(sl)
30000003
```


Here it is. This is the project home page. SortedContainers is a Python sorted collections library with sorted list, sorted dictionary, and sorted set implementations. It's pure-Python but it's as fast as C-extensions. It's Python 2 and Python 3 compatible. It's fully-featured. And it's extensively tested with 100% coverage and hours of stress.



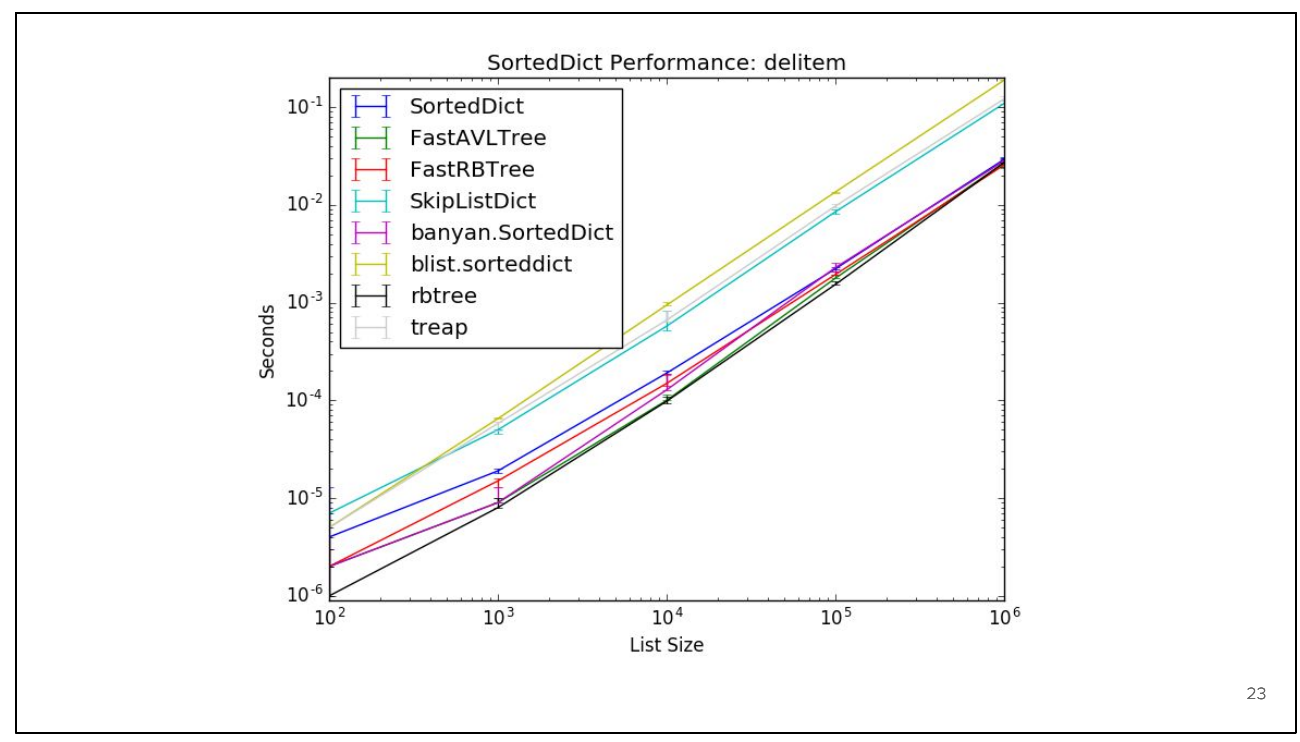
Performance is a feature. That means graphs. Lot's of them. There are 189 performance graphs in total. Let's look at a few of them together.

Here's the performance of adding a random value to a sorted list. I'm comparing SortedContainers with other competing implementations.

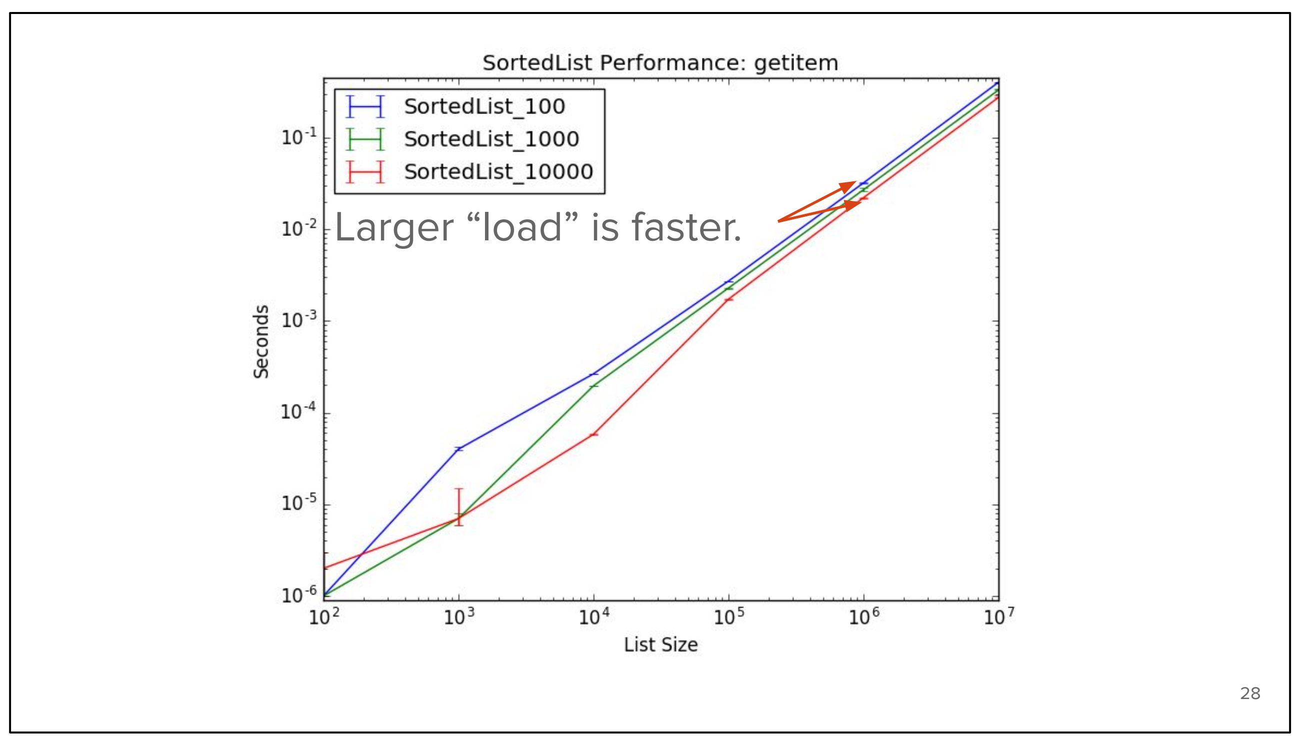
Notice the axes are log-log. So if performance differs by major tick

marks then one is actually ten times faster than the other.
We see here that SortedContainers is in fact about ten times faster than blist when it comes to adding random values to a sorted list.
Notice also Raymond’s recipe is just a list and that displays order n-squared runtime complexity. That’s why it curves upwards.

Of all the sorted collections libraries, SortedContainers is also fastest at initialization. We’ll look at why soon.



SortedContainers is not always fastest. But notice here the performance improves with scale. You can see it there in blue. It starts in the middle of the pack and has a lesser slope than competitors.

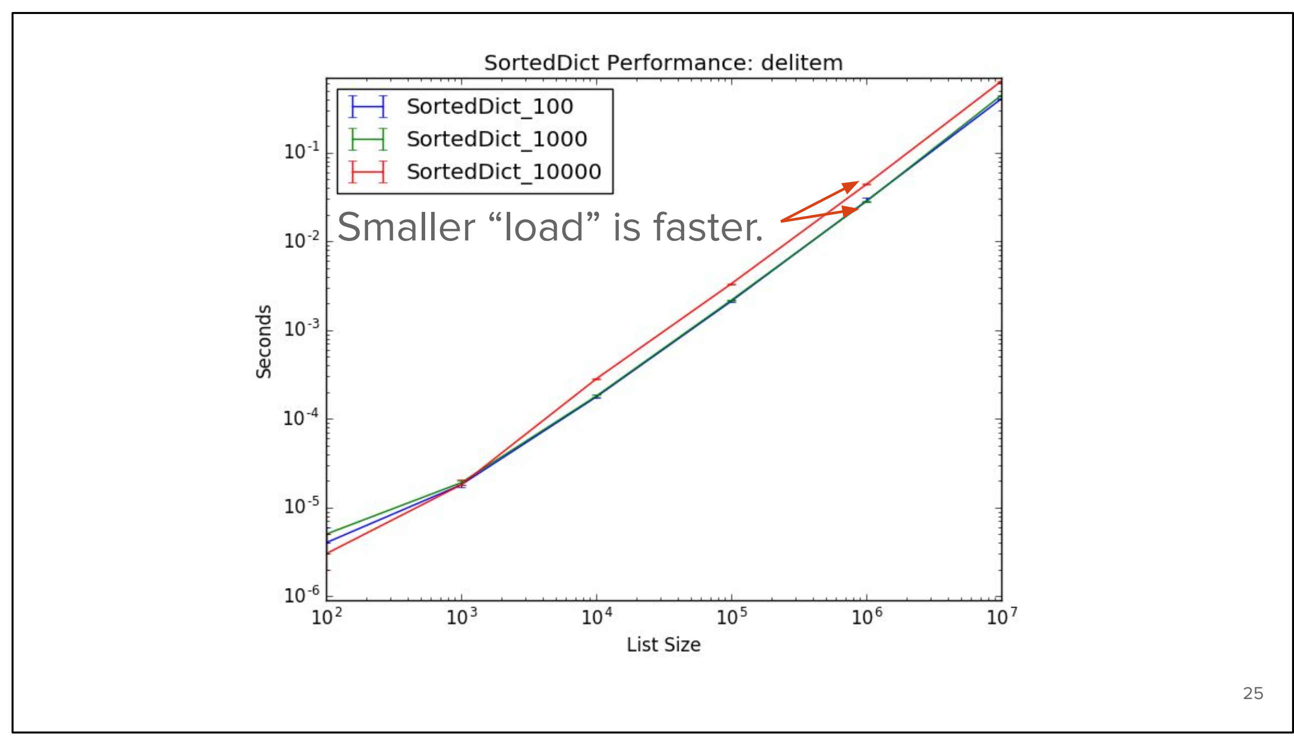


In short, SortedContainers is kind of like a B-tree implementation. That means you can configure the the fan-out of nodes in the tree. We call that the load parameter and there are extensive performance graphs of three different load parameters.

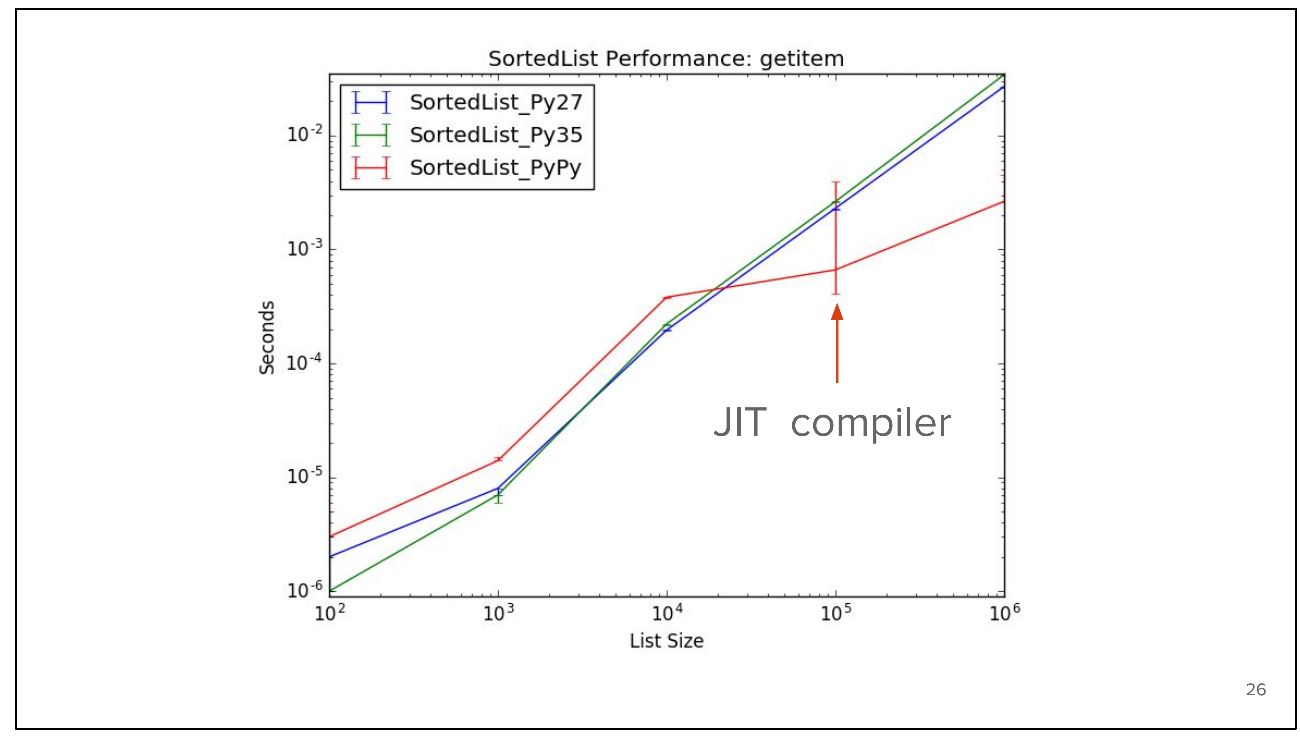
Here we see that a load factor of ten thousand is fastest for indexing a sorted list.

Notice the axes now go up to ten million elements.

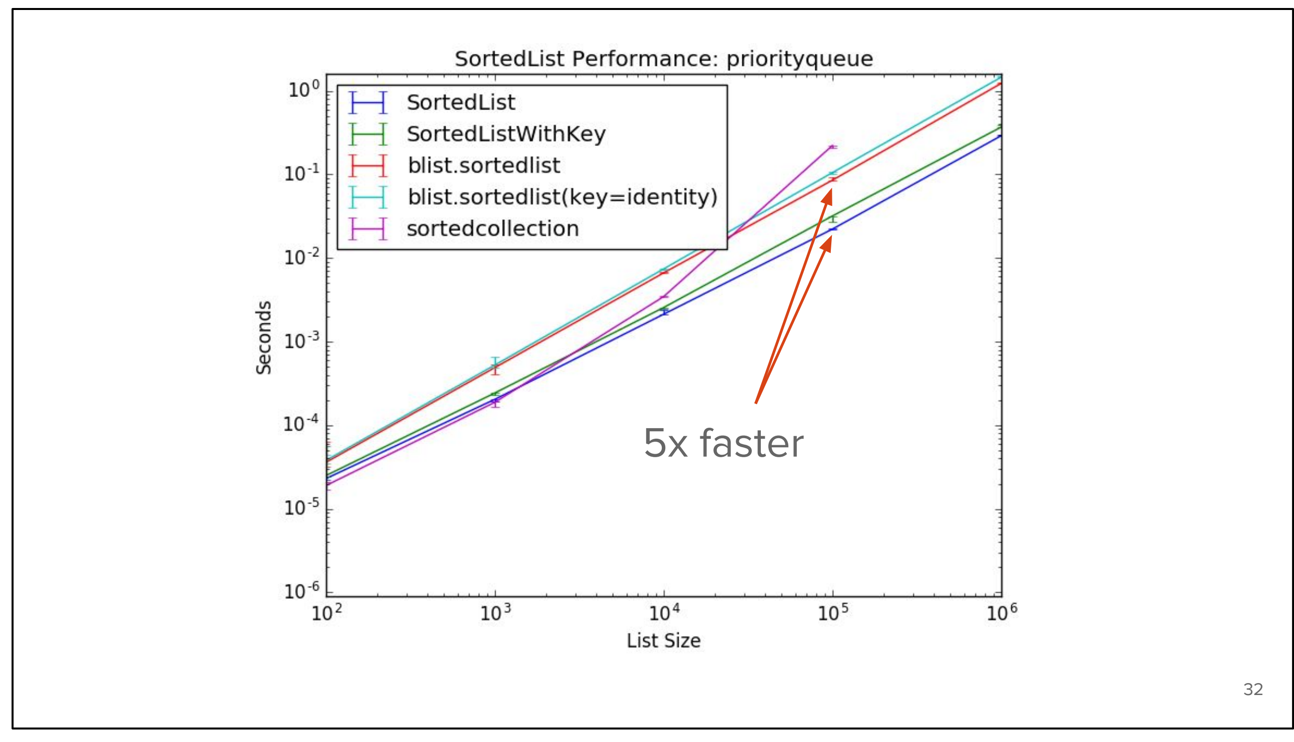
I’ve actually scaled SortedList all the way to ten billion elements. It was a really incredible experiment. I had to rent the largest high-memory instance available from Google Compute Engine. That benchmark required about 128 gigabytes of memory and cost me about thirty dollars.



This is the performance of deleting a key from a sorted dictionary. Now the smaller load-factor is fastest. The default load-factor is 1,000 and works well for most scenarios. It’s a very sane default.



In addition to comparisons and load-factors, I also benchmark runtimes. Here's CPython 2.7, CPython 3.5 and PyPy version 5. You can see where the the just-in-time compiler, the jit-compiler, kicks in. That'll make SortedContainers another ten times faster.



Finally, I made a survey in 2015 on Github as to how people were using sorted collections. I noticed patterns like priority queues, mutli-sets, nearest-neighbor algorithms, etc.

This is the priority queue workload which spends 40% of its time adding elements, 40% popping elements, 10% discarding elements, and has a couple other methods.

SortedContainers is two to ten times faster in all of these scenarios.

Features

```
1  sorted_set.pop()
2  sorted_list.bisect_right('carol')
3  sorted_dict.irange('bob', 'eve')
4  sorted_dict.iloc[-5:]
5  sorted_set.islice(10, 50)
```


We also have a lot of features. The API is nearly a drop-in replacement for the "blist" and "rbtree" modules. But the quirks have been fixed so the "pop" method returns the last element rather than the first.
Sorted lists are sorted so you can bisect them. Looking up the index of an element is also very fast.
Bintrees introduced methods for tree traversal. And I've boiled those down to

a couple API methods. On line 3, we see “irange”. Irange iterates all keys from bob to eve in sorted order. Sorted dictionaries also have a sequence-like view called iloc. If you’re coming from Pandas that should look familiar. Line 4 creates a list of the five largest keys in the dictionary. Similar to “irange” there is an “islice” method. Islice does positional index slicing. In line 5 we create an iterator over the indexes 10 through 49 inclusive.

Recipes

- ValueSortedDict - dictionary sorted by item value.
- ItemSortedDict - key, value sort order function.
- OrderedDict - insertion order with positional indexing.
- IndexableSet - supports positional indexing.
- `$ pip install sortedcollections`

One of the benefits of being pure-Python: it's easy to hack on. Over the years, a few patterns have emerged and become recipes. All of these are available from PyPI with pip install sortedcollections.

Alex Martelli, Wikipedia

Good stuff! ... I like the *simple, effective implementation* idea of splitting the sorted containers into smaller "fragments" to avoid the O(N) insertion costs.

Jeff Knupp, Review of SortedContainers

That last part, "fast as C-extensions," was difficult to believe. I would need some sort of *performance comparison* to be convinced this is true. The author includes this in the docs. It is.

Kevin Samuel, Formations Python

I'm quite amazed, not just by the code quality (it's incredibly readable and has more comment than code, wow), but the actual amount of work you put at stuff that is *not* code: documentation, benchmarking, implementation explanations. Even the git log is clean and the unit tests run out of the box on Python 2 and 3.

Testimonials

37

If all that didn't convince you that Sorted Containers is great then listen to what other smart people say about it:
Alex Martelli says: Good stuff! ... I like the simple, effective implementation idea of splitting the sorted containers into smaller "fragments" to avoid the O(N) insertion costs.
Jeff Knupp writes: That last part, "fast as C-extensions," was difficult to

believe. I would need some sort of performance comparison to be convinced this is true. The author includes this in the docs. It is.
Kevin Samuel says: I’m quite amazed, not just by the code quality (it’s incredibly readable and has more comment than code, wow), but the actual amount of work you put at stuff that is not code: documentation, benchmarking, implementation explanations. Even the git log is clean and the unit tests run out of the box on Python 2 and 3.

Under The Hood: SortedContainers

31

If you're new to sorted collections, I hope I've piqued your interest.
Think about the achievement here.
SortedContainers is pure-Python but as fast as C-implementations.
Let's look under the hood of SortedContainers at what makes it so fast.

8.6. bisect — Array bisection algorithm

Source code: Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called `bisect` because it uses a basic bisection algorithm to do its work. The source code may be most useful as a working example of the algorithm (the boundary conditions are already right!).

The following functions are provided:

`bisect.`**`bisect_left`**(*a, x, lo=0, hi=len(a)*)

Locate the insertion point for *x* in *a* to maintain sorted order. The parameters *lo* and *hi* may be used to specify a subset of the list which should be considered; by default the entire list is used. If *x* is already present in *a*, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()` assuming that *a* is already sorted.

The returned insertion point *i* partitions the array *a* into two halves so that `all(val < x for val in a[lo:i])` for the left side and `all(val >= x for val in a[i:hi])` for the right side.

`bisect.`**`bisect_right`**(*a, x, lo=0, hi=len(a)*)
`bisect.`**`bisect`**(*a, x, lo=0, hi=len(a)*)

bisect module

It really comes down to bisect for the heavy lifting. Bisect is a module in the standard library that implements binary search on lists. There's also a handy method called insort that does a binary search and insertion for us in one call. There's no magic here, it's just implemented in C and part of the standard library.

List of Sublists

```
[   # _lists
    [ 0,  1,  2, 3],
    [ 4,  5,  6],
    [ 7,  8,  9, 10, 11, 12],
    [13, 14, 15, 16, 17],
]
```


Here’s the basic structure. It’s just a list of sublists. So there’s a member variable called “lists” that points to sublists. Each of those is maintained in sorted order. You’ll sometimes hear me refer to these as the top-level list and its sublists.

There’s no need to wrap sublists in their own objects. They are just lists. Simple is fast and efficient.

List of Maxes

```
[    # _lists                        [    # _maxes
    [ 0,  1,  2, 3],                      3,
    [ 4,  5,  6],                         6,
    [ 7,  8,  9, 10, 11, 12],             12,
    [13, 14, 15, 16, 17],                 17,
]                                    ]
```

42

In addition to the list of sublists. There's an index called the maxes index. That simply stores the maximum value in each sublist. Now lists in CPython are simply arrays of pointers so we're not adding much overhead with this index.

Let's walk through testing membership with contains. Let's look for element 14.

Let’s also walk through adding an element. Let’s add 5 to the sorted list.

“Jenks” Index

```
1  lengths = [  4,  3,  6,  5 ]
2  pair_wise_sums1 = [  7, 11 ]
3  pair_wise_sums2 = [ 18 ]
4  _index = [ 18,  7, 11,  4,  3,  6,  5 ]
5  _offset = 3
```


Now numeric indexing is a little more complex. Numeric indexing uses a tree packed densely into another list. I haven’t seen this structure described in textbooks or research so I’d like to call it a “Jenks” index. But I’ll also refer to it as the positional index.

Let’s build the positional index together …

Positional Indexing

```
_index = [        18,
             7,          11,
          4,  3,      6,     5 ]
```

```
_offset = 3
```

```
1  @18, index = 8, position = 0
2  @11, index = 1, position = 2
3   @6, index = 1, position = 5, topindex = 2
```


Remember the positional index is a tree stored in a list, kind of like a heap.

Let's use this to lookup index 8.
Starting at the root, 18, compare index to the left-child node.
8 is greater than 7 so we subtract 7 from 8 and move to the right-child node.
Again, now at node 11, compare index again to the left-child node.

1 is less than 6, so we simply move to the left-child node.
We terminate at 6 because it’s a leaf node.
Our final index is 1 and our final position is 5. We calculate the top-level list index as the position minus the offset.
So our final coordinates are index 2 in the top-level list and index 1 in the sublist.

That’s it. Three lists maintain the elements, the maxes index, and the positional index. We’ve used simple built-in types to construct complex behavior.

Altogether that gets us to our first performance lesson.

Builtin types are ***fast***.

37

Builtin types are fast. Like really fast. Builtin types are as fast as C and benefit from years of optimizations.

SortedList.__contains__

```
1  def __contains__(self, val):
2      _lists = self._lists
3      pos = bisect_left(self._maxes, val)
4      idx = bisect_left(_lists[pos], val)
5      return _lists[pos][idx] == val
```


Ok, let's look at the contains method for a sorted list. This is the majority of the code. We bisect the maxes index for the sublist index. Then we bisect the sublist for the element index.
How many lines of Python code execute? 4.
How many instructions execute? Hundreds of lines of C-code.
Rather than programming in Python, I programmed against my interpreter.

That’s our next lesson.

Program ~~in Python~~ your interpreter.

39

Program your interpreter. The operations provided by the interpreter and standard library are fast. They're implemented in C. When you program your interpreter, you write C code but in Python.

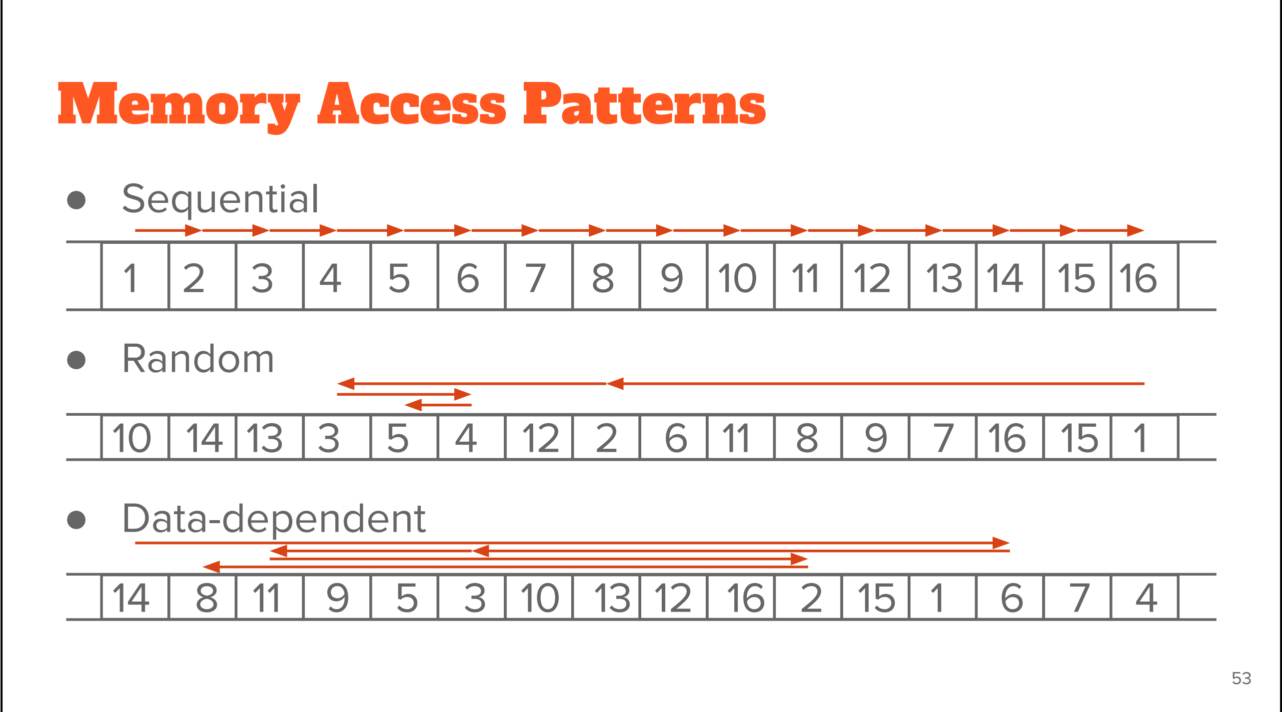
Memory is Tiered

- Registers - dozenish.
- L1 Instruction/Data Cache - 32 KB.
- L2 Cache - 256 KB.
- [L3 Cache (Shared) - 8 MB.]
- Main Memory - Gigabytes.

Now let's talk about memory. This is very simplified. My apologies to those who feel this is grossly simplified. Notice the limited sizes: a dozen registers, kilobytes of L1 cache, megabytes of L3 cache. Some machines don't even have an L3 cache.

So keep overhead low. Keep related data packed together. Our sublists add roughly one pointer per element. That's

all. It’s 66% less memory than binary tree implementations.



Also, each memory tier has different performance. Memory slows down by a factor of a thousand from registers to main memory. And the advertised price of memory lookups is often the average random lookup time. But that's only one common pattern.

Sequential memory access patterns are so fast you almost don't pay for them at all. The processor predicts the

memory you’ll need next and queues it for you.

Then there’s also data-dependent memory accesses which happen when you follow pointers. So the next memory location is dependent on the current one. This is typical in binary trees and it’s really slow. It’s as much as ten times slower than random memory access.

list.insert

```
1  for (i = n; --i >= where; )
2      items[i+1] = items[i];
3  Py_INCREF(v);
4  items[where] = v;
```


Let’s think about adding elements again. Add calls bisect.insort which does a binary search and then insert on the list.

Here is the code for insert in CPython. It is entirely sequential memory accesses.
Also the binary search process starts random but narrows the search range and so improves locality of memory

accesses.

By comparison, traditional binary trees use data-dependent memory access as they repeatedly dereference pointers.

We can sequentially shift a thousand elements in memory in the time it takes to access a couple of binary nodes from DRAM.

tiered.

Memory is tiered.

tiered.

tiered.

43

So memory is tiered. And caches are limited in size.
This is also why the slope of the performance curve for sorted list was less than that for binary tree implementations. At scale, binary trees do more data-dependent DRAM lookups than SortedContainers.

SortedList.__init__

```
1  values = sorted(iterable)
2  _lists = [values[pos:pos+load] for pos in
3             range(0, len(values), load)]
4  _maxes = [sub[-1] for sub in _lists]
```


I said that initializing a SortedContainer is fast. Let's look at why. Here's the initializer for a SortedList. Notice it simply calls the sorted builtin and then chops up the result into sublists and then initializes the maxes index.

I think of this as a cheat. I'm using the power of Timsort to initialize the container. And it turns out initialization is really common. The result is fast and

readable.

Also, how long does it take to initialize already sorted data? Linear time. It’s just a couple mem-copy like operations.

SortedSet.add

```
1 def add(self, value):
2     _set, _list = self._set, self._list
3     if value not in _set:
4         _set.add(value)
5         _list.add(value)
```

60

Here’s another cheat. When we add an element to a sorted set, we add it to both a set object and sorted list. This preserves the fast set membership tests.

Some purists will argue that hashing should not be necessary. They are correct, but, if you can define comparisons, then you can probably define hash. Remember that we’re

solving real problems, not theoretical ones. If you can reuse the builtin types, then cheat and do it.

Cheat, if you can.

46

So, if you can, cheat. The way to make things faster is to do less work. There's no way around that.
Another cheat I've mentioned regards the positional index. If you don't need numerical lookups, then don't build the index. That's a common scenario with sorted dictionaries. We use less memory and run faster.

Runtime Complexity

- Punchline: $O(\sqrt[3]{n})$
- Billion integers in CPython: 30 GBs.
- Timsort: comparisons are expensive.
- Memory is expensive.
- Performance at Scale: 10,000,000,000

When it comes to runtime complexity, here's the punchline: adding random elements has an amortized cost proportional to the cube root of the container size. That's an unusual runtime complexity but it works quite well.
The surprising thing is that "n" stays relatively small in practice. For example, creating a billion integers in CPython will take more than 30

gigabytes of memory which is already exceeding the limits of most machines. We've also seen that memory is expensive. Allocations are costly. In the common case, SortedContainers allocates no more memory when adding elements.

If you're doubtful about performance at scale, then I encourage you to read the project docs. There's a page called Performance at Scale and it talks extensively about theory with benchmarks up to ten billion elements.

A little PSA before I continue: If you claim to be fast, you've got to have measurements. Measure. Measure. Measure. Big-O notation is not a substitute for benchmarks. Quite often, constants and coefficients that are ignored in theory matter quite a lot in practice.

Measure.
Measure.
Measure.

65

So: Measure. Measure. Measure.
This whole project in fact started with a measurement. I was timing how long it took to add an element to a “blist” when I noticed that “bisect.insort” was actually faster for a list with one thousand elements. It was so much faster in fact, I thought “wow, I could do two inserts in a thousand-element list and still be faster than “blist.” That thought eventually became the list of

sublists implementation that we have today.

SortedContainers Performance

- Builtin types are ***fast***.
- Program ~~in Python~~ your interpreter.
- Memory is tiered.
- Cheat, if you can.
- Measure. Measure. Measure.

So here's the performance lessons:
Builtin types are fast.
Program your interpreter.
Memory is tiered.
Cheat, if you can.
Measure. Measure. Measure.

SortedContainers

SortedContainers is an *Apache2 Licensed* containers library, written in pure-Python, and fast as C-extensions.

Python's standard library is great until you need a sorted container type. Many will attest that you can get really far without one, but the moment you **really need** a sorted list, dict, or set, you're faced with a dozen different implementations, most using C-extensions without great documentation and benchmarking.

Things shouldn't be this way. Not in Python.

Star 514

SortedContainers provides sorted container types, written in pure-Python and fast as C-extensions.

Give Support

If you or your organization uses SortedContainers, consider financial support:

```
>>> sl = sortedcontainers.SortedList(xrange(10000000))
>>> 1234567 in sl
True
>>> sl[7654321]
7654321
>>> sl.add(1234567)
>>> sl.count(1234567)
2
>>> sl *= 3
>>> len(sl)
30000003
```


A couple closing thoughts. Everything related to SortedContainers is under an open-source Apache2 license. Contributors are very welcome. We've started to create a little community around sorted collections.

I think it's interesting to ask: is this worth a PEP? I'm personally on the fence. I think sorted collections would contribute to Python's maturity. But I

don’t know if any proposal could survive the inevitable bike-shedding. My contribution is a pure-Python implementation that’s fast-enough for most scenarios.

I’ll end with a quote from Mark Summerfield. Mark and a couple other authors have actually deprecated their modules in favor of SortedContainers. Mark says: “Python’s ‘batteries included’ standard library seems to have a battery missing. And the argument that ‘we never had it before’ has worn thin. It is time that Python offered a full range of collection classes out of the box, including sorted ones.”

Thanks for letting me share.

[Questions?]