slice-aggregator Documentation

Release 0.1.0

Bartosz Marcinkowski

Mar 15, 2018

Contents:

It is a library for aggregating values assigned to indices by slices

```
>>> import slice_aggregator
>>> a = slice_aggregator.ixs_by_slices()
>>> a[-5] += 1
>>> a[10] -= 2.5
>>> a[-10:]
-1.5
```

and the other way around

```
>>> import slice_aggregator
>>> a = slice_aggregator.slices_by_ixs()
>>> a[:-5] += 1
>>> a[-10:10] -= 2.5
>>> a[-10]
-1.5
```

CHAPTER 1

API

slice_aggregator.**slices_by_ixs**(*, *zero_factory: typing.Callable[[], V] = None*, *zero_test: typing.Callable[[V], bool] = None*) → slice_aggregator.by_ixs.Aggregator[V]

Returns an object that allows assigning values to slices and aggregating them by indices

Parameters

- **zero_factory** – callable returning additive identity
- **zero_test** – test for equality to zero

Returns a new instance of *slice_aggregator.by_ixs.Aggregator*

slice_aggregator.**ixs_by_slices**(*, *zero_factory: typing.Callable[[], V] = None*, *zero_test: typing.Callable[[V], bool] = None*) → slice_aggregator.by_slices.Aggregator[V]

Returns an object that allows assigning values to indices and aggregating them by slices

Parameters

- **zero_factory** – callable returning additive identity
- **zero_test** – test for equality to zero

Returns a new instance of *slice_aggregator.by_slices.Aggregator*

class slice_aggregator.by_ixs.**Aggregator**(*, *dual: slice_aggregator.by_slices.Aggregator*, *zero_factory: typing.Callable[[], V] = None*)

A data structure for assigning values to slices and aggregating them by indices

It provides a method-based interface and an alternative based on *__getitem__* and slices.

Warning: Only the method-based interface is suitable for custom values handling inplace operators. Read the documentation on advances usage for more details.

get(*ix: int*) → V

Get the aggregated value of all slices containing the specified index

inc(*start: typing.Union[int, NoneType], stop: typing.Union[int, NoneType], value: V*) → None

Increment the value assigned to a slice

dec(*start: typing.Union[int, NoneType], stop: typing.Union[int, NoneType], value: V*) → None
Decrement the value assigned to a slice

class slice_aggregator.by_slices.**Aggregator**
A data structure for assigning values to indices and aggregating them by slices

It provides a method-based interface and an alternative based on *__getitem__* and slices.

Warning Only the method-based interface is suitable for custom values handling inplace operators. Read the documentation on advances usage for more details.

get(*start: typing.Union[int, NoneType], stop: typing.Union[int, NoneType]*) → V
Get the aggregated value of all indices contained by the specified slice

inc(*ix: int*, *value: V*) → None
Increment the value assigned to an index

dec(*ix: int*, *value: V*) → None
Decrement the value assigned to an index

set(*ix: int*, *value: V*) → None
Set the value assigned to an index

CHAPTER 2

Advanced usage

The library "just works" with value types that:

1. implement addition-like binary operation via Python magic-methods (`__add__`, `__sub__`, `__sub__`, `__pos__`)
2. use `0` as the neutral element for their addition implementation
3. implement `__eq__` that allows testing for equality to zero
4. do not implement inplace addition/subtraction (`__iadd__`, `__isub__`)

All Python's numeric types (`int`, `float`, `long`, `complex`) fall into that category.

The first condition is a hard requirement for any type to be used for values, but the others are not. You can use another value as a neutral element by using the `zero_factory` parameter, you don't have to worry about `__eq__` if you supply `zero_test` and you can have `__iadd__` and `__isub__` if you use the method-based interface.

Example:

```
>>> import numpy as np
>>>
>>> def zero_factory():
...     return np.zeros(3)
>>>
>>> zero = zero_factory()
>>>
>>> def zero_test(v):
...     return np.array_equal(v, zero)
>>>
>>> import slice_aggregator
>>>
>>> a = slice_aggregator.ixs_by_slices(zero_factory=zero_factory, zero_test=zero_test)
>>> a.inc(-5, np.array([1, 0, 3.5]))
>>> a.dec(10, np.array([2.5, -1, 0]))
>>> tuple(a.get(-10, None))  # a[-10:]
(-1.5, 1.0, 3.5)
```

CHAPTER 3

Time and memory complexity

After assigning values to `n` unique indices (we treat a slice as, up to two, indices) that are all within a `(-v, v)` interval:

Reading (aggregating) time	O(log v)
Writing (assigning) time	O(log v + log n)
Memory	O(n log v)

Assumptions:

- values and indices are constant-size and basic arithmetic operations on them are constant-time
- set item and get item on a `dict` are constant-time (which is true on average)

CHAPTER 4

Underlying data structure

4.1 by_slice

The core concept is a data structure similar to a Fenwick tree that allows assigning values to nonnegative indices and efficiently computing suffix sums. Where a Fenwick tree would store an aggregate for *[a, b]*, it stores an aggregate for *[b, b + b - a]*. With that change, while modifying the value for index *ix* it goes along decreasing indices, so it doesn't need to know the size of the internal table (maximum allowed value). So all values above the biggest index modified by the user are zeroes. That's useful for computing suffix sums - moving along increasing indices, the biggest one the user has set to a non-zero value is where one can stop. A max heap with an index is used to efficiently track these upper bounds.

The unbounded variant is just a combination of two such left-bounded data structures.

4.2 by_ix

This a thin layer on top of the previous data structure. Incrementing *[a, b)* translates to decrementing *a - 1* and incrementing *b - 1* of the underlying *by_slice* aggregator, and aggregating slices translates to a suffix sum.

CHAPTER 5

Indices and tables

- genindex

Index