arrex

Release v0.5

jimy byerley

Aug 07, 2023

CONTENTS

Arrex is a module that allows to create typed arrays much like numpy.ndarray and array.array, but resizeable and using any kind of element.

The elements must be extension-types (eg. class created in compiled modules) and must have a packed and copyable content: a fixed size and no reference or pointers This is meant to ensure that the content of those objects can be copied from the object to the array and back to any object after, or even deleted without any need of calling a constructor or destructor function.

basic usage:

```
>>> from arrex import *
>>> a = typedlist([
...             myclass(...),
...             myclass(...),
...             ], dtype=myclass)
>>> a[0]
myclass(...)
```

in that example, `myclass` can be a primitive numpy type, like `np.float64`

```
>>> import arrex.numpy          # this is enabling numpy dtypes for arrex
>>> typedlist(dtype=np.float64)
```

it can be a more complex type, from module `pyglm` for instance

```
>>> import arrex.glm            # this is enabling glm dtypes for arrex
>>> typedlist(dtype=glm.vec4)
```

```
>>> a = typedlist(dtype=vec3)
```

use it as a list

```
>>> # build from an iterable
>>> a = typedlist([], dtype=vec3)
>>>
>>> # append some data
>>> a.append(vec3(1,2,3))
>>>
>>> # extend with an iterable
>>> a.extend(vec3(i)  for i in range(5))
>>>
>>> len(a)      # the current number of elements
6
>>> a.owner     # the current data buffer
b'........'
>>> a[0]
vec3(1,2,3)
```

Use it as a slice:

```
>>> myslice = a[:5]             # no data is copied
typedlist(....)
```

Use it as a view on top of a random buffer

```
>>> a = np.ones((6,3), dtype='f4')
>>> myslice = typedlist(a, dtype=vec3)
```

It does support the buffer protocol, so it can be converted into a great variety of well known arrays, even without any copy

```
>>> np.array(typedlist([....]))
```

CHAPTER

ONE

SUBMODULES:

1.1 list

class typedlist

list-like array that stores objects as packed data. The objects added must necessarily be packed objects (builtin objects with no references).

This is a dynamically sized, borrowing array, which mean the internal buffer of data is reallocated on insertion, but can be used to view and extract from any buffer.

Methods added to the signature of list:

reserve(n) reallocate if necessary to make sure n elements can
be inserted without reallocation

capacity() -> int return the current number of elements that can be
contained without reallocation

shrink() shorten the allocated memory to fit the current content

also the slices do not copy the content

Use it as a list:

```
>>> a = typedlist(dtype=vec3)
>>>
>>> # build from an iterable
>>> a = typedlist([], dtype=vec3)
>>>
>>> # append some data
>>> a.append(vec3(1,2,3))
>>>
>>> # extend with an iterable
>>> a.extend(vec3(i)  for i in range(5))
>>>
>>> len(a)      # the current number of elements
6
>>>
>>> a.owner     # the current data buffer
b'.........'
```

Use it as a slice:

```
>>> # no data is copied
>>> myslice = a[:5]
typedlist(....)
```

Use it as a view on top of a random buffer

```
>>> a = np.ones((6,3), dtype='f4')
>>> myslice = typedlist(a, dtype=vec3)
```

It does support the buffer protocol, so it can be converted in a great variety of well known arrays, even without any copy

```
>>> np.array(typedlist([....]))
```

Constructors:

typedlist()

typedlist(*dtype*, *reserve=None*)

typedlist(*iterable*, *dtype*, *reserve=None*)

typedlist(*buffer*, *dtype*)

size

byte size of the current content

Type
int

allocated

byte size of the memory allocated memory

Type
int

owner

object realy owning the data instead of the current `typedlist`

dtype

the python data type

Type
type

ddtype

the data type declaration

Type
DDType

static empty(*dtype*, *size*)

create a new typedlist with the given `size` and unitialized elements of type `dtype`

static full(*value*, *size*)

create a new typedlist with the given `size`, all elements initialized to `value`.

- Methods matching those from `list`

append(*value*)

append the given object at the end of the array

if there is not enough allocated memory, reallocate enough to amortize the realocation time over the multiple appends

extend(*iterable*)

append all elements from the other array

insert(*index*, *value*)

insert value at index

clear()

remove all elements from the array but does not deallocate, very fast operation

reverse()

reverse the order of elementd contained

index(*value*)

return the index of the first element binarily equal to the given one

__add__()

concatenation of two arrays

__mul__()

duplicate the sequence by a certain number

__getitem__()

self[index]

currently supports:

- indices
- negative indices
- slices with step=1

__setitem__(*key*, *value*, /)

Set self[key] to value.

__delitem__(*key*, /)

Delete self[key].

__iter__()

yield successive elements in the list

__copy__()

shallow copy will create a copy of that array referencing the same buffer

__deepcopy__(*memo*)

deep recursive copy, will duplicate the viewed data in the underlying buffer

- The following methods are added on top of python `list` signature, in order to manage memory in a more efficient way.

capacity()

return the total number of elements that can be stored with the current allocated memory

`reserve`(*amount: int*)

Make sure there is enough allocated memory to append the given amount of elements.

if there is not enough of allocated memory, the memory is reallocated immediately.

`shrink`()

reallocate the array to have allocated the exact current size of the array

1.2 dtypes

1.2.1 general

The *dtype* is the type of the elements in a buffer. Thanks to the ddtype system, it is very easy to create new dtypes on top of pretty much everything.

Definitions:

type
a python type object (typically a class or a builtin type)

dtype
data dtype, meaning the type of the elements in an array, it can be a type, but more generally anything that define a data format.

ddtype
declaration of data type, meaning a packet of things decribing how to pack/unpack that dtype from/to an array

a ddtype always inherits from base class `DDType` which content is implemented at C level.

`class DDType`

base class for a declaration of data type (ddtype) DO NOT INSTANTIATE THIS CLASS FROM PYTHON, use on of its specialization instead

`dsize`

byte size of the dtype when packed

Type
int

`layout`

layout of the packed data such as defined in module `struct`, or `None` if not defined

Type
bytes

`key`

the python dtype itself if this DDType is declared, `None` if not declared

`declare`(*dtype*, *ddtype*)

declare a new dtype

`declared`(*key*)

return the content of the declaration for the givne dtype

1.2.2 specialized dtypes

class DDTypeFunctions(*dsize*, *pack*, *unpack*, *layout=None*)

create a dtype from pure python pack and unpack functions

Example

```
>>> enum_pack = {'apple':b'a', 'orange':b'o', 'cake':b'c'}
>>> enum_unpack = {v:k   for k,v in enum_direct.items()}
>>> enum_dtype = DDTypeFunctions(
...                     dsize=1,                          # 1 byte storage
...                     pack=enum_pack.__getitem__,       # this takes the python␣
↪object and gives a bytes to dump
...                     unpack=enum_unpack.__getitem__,   # this takes the bytes and␣
↪return a python object
...                     )
...
>>> a = typedlist(dtype=enum_dtype)             # declaration is not necessary
```

class DDTypeClass(*type*)

Create a dtype from a python class (can be a pure python class)

the given type must have the following attributes:

- `frombytes` or `from_bytes` or `from_buffer`

  static method that initialize the type from bytes

- `__bytes__` or `tobytes` or `to_bytes`

  method that converts to bytes, the returned byte must always be of the same size

- `__packlayout__` (optional) string or bytes giving binary format returned by `__bytes__`, it must follow the specifications of module `struct`

- `__packsize__` (optional) defines the byte size returned by `__bytes__`, optional if `__packlayout__` is provided

Example

```
>>> class test_class:
...     __packlayout__ = 'ff'
...     _struct = struct.Struct(__packlayout__)
...
...     def __init__(self, x, y):
...             self.x = x
...             self.y = y
...
...     def __bytes__(self):
...             return self._struct.pack(self.x, self.y)
...     @classmethod
...     def frombytes(cls, b):
...             return cls(*cls._struct.unpack(b))
...
```

(continues on next page)

(continued from previous page)

```
...     def __repr__(self):
...             return '(x={}, y={})'.format(self.x, self.y)
...
>>> a = typedlist(dtype=test_class)    # no declaration needed
```

class DDTypeStruct(*struct*)

create a dtype from a Struct object from module `struct`

Example

```
>>> a = typedlist(dtype='fxBh')   # no declaration needed
```

class DDTypeCType(*type*)

Create a dtype from a ctype

Example

```
>>> class test_structure(ctypes.Structure):
...     _fields_ = [
...             ('x', ctypes.c_int),
...             ('y', ctypes.c_float),
...             ]
...     def __repr__(self):
...             return '(x={}, y={})'.format(self.x, self.y)
...
>>> a = typedlist(dtype=test_structure)
```

class DDTypeExtension

DDTypeTypeExtension(type, layout=None, constructor=None)

Create a dtype for a C extension type.

This is the most efficient kind of dtype in term of access/assignation time.

In order to put an extension object into an array, it satisfy the following conditions:

- have fixed size known at the time of dtype creation (so any array element has the same)
- contain only byte copiable data (so nothing particular is done when copying/destroying the objects)

> **Warning:** These conditions MUST be ensured by the user when declaring an extension type as a dtype, or it will result in memory corruption and crash of the program

Example

```
>>> arrex.declare(vec3, DDTypeExtension(vec3, 'fff', vec3))
```

PYTHON MODULE INDEX

a

INDEX