Typed list

This lesson is a brief introduction to Typed list and how it is implemented using the NumPy library.

We'll cover the following

Introduction

One of the strengths of NumPy is that it can be used to build new objects or to subclass the ndarray object. This later process is a bit tedious but it is worth the effort because it allows you to improve the `ndarray` object to suit your problem.

We'll examine in the following section two real-world cases (typed list and memory-aware array) that are extensively used in the glumpy project (that I maintain).

What is a Typed list?

Typed list (also known as ragged array) is a list of items that all have the same data type (in the sense of NumPy). They offer both the list and the `ndarray` API (with some restriction of course) but because their respective APIs may not be compatible in some cases, we have to make choices. For example, concerning the `+` operator, we'll choose to use the NumPy API where the value is added to each individual item instead of expanding the list by appending a new item (1).

```
l = TypedList([[1,2], [3]])
print(l)
#[1, 2], [3]
print(l+1)
#[2, 3], [4]
```

From the list API, we want our new object to offer the possibility of inserting, appending and removing items seamlessly.

Creation

Since the object is dynamic by definition, it is important to offer a general-purpose creation method powerful enough to avoid having to do later manipulations. Such manipulations, for example, insertion/deletion, cost a lot of operations and we want to avoid them.

Here is a proposal (among others) for the creation of a `TypedList` object.

```
def __init__(self, data=None, sizes=None, dtype=float)
    """
    Parameters
    ----------

    data : array_like
        An array, any object exposing the array interface, an object
        whose __array__ method returns an array, or any (nested) sequence.

    sizes:  int or 1-D array
        If `itemsize` is an integer, N, the array will be divided
        into elements of size N. If such partition is not possible,
        an error is raised.

        If `itemsize` is a 1-D array, the array will be divided into
        elements whose successive sizes will be picked from itemsize.
        If the sum of itemsize values is different from array size,
        an error is raised.

    dtype: np.dtype
        Any object that can be interpreted as a NumPy data type.
    """
```

Implementation

This API allows creating an empty list or creating a list from some external data. Note that in the latter case, we need to specify how to partition the data into several items or they will split into `1-size` items. It can be a regular partition (i.e., each item is 2 data long) or a custom one (i.e., data must be split in items of size 1, 2, 3, and 4)

```
L = TypedList([[0], [1,2], [3,4,5], [6,7,8,9]])
print(L)
# [ [0] [1 2] [3 4 5] [6 7 8 9] ]

L = TypedList(np.arange(10), [1,2,3,4])
# [ [0] [1 2] [3 4 5] [6 7 8 9] ]
```

At this point, the question is whether to subclass the `ndarray` class or to use an internal `ndarray` to store our data. In our specific case, it does not really make sense to subclass `ndarray` because we don't really want to offer the `ndarray` interface. Instead, we'll use an `ndarray` for storing the list data and this design choice will offer us more flexibility.

```
-----┌───┐┌───┬───┐┌───┬───┬───┐┌───┬───┬───┬───┐------
     │ 0 ││ 1 │ 2 ││ 3 │ 4 │ 5 ││ 6 │ 7 │ 8 │ 9 │
  ---└───┘└───┴───┘└───┴───┴───┘└───┴───┴───┴───┘-------

   item 1   item 2     item 3           item 4
```

To store the limit of each item, we'll use an items array that will take care of storing the position (`start` and `end`) for each item.

For the creation of a list, there are two distinct cases:

- No data is given
- Some data is given

The first case is easy and requires only the creation of the `_data` and `_items` arrays. Note that their size is not null since it would be too costly to resize the array each time we insert a new item. Instead, it's better to reserve some space.

First case

No data has been given, only the `dtype`:

```
self._data = np.zeros(512, dtype=dtype)
self._items = np.zeros((64,2), dtype=int)
self._size = 0
self._count = 0
```

Second case

Some data has also been given as a list of item sizes (for other cases, see full code below):

```
self._data = np.array(data, copy=False)
self._size = data.size
self._count = len(sizes)
indices = sizes.cumsum()
self._items = np.zeros((len(sizes),2),int)
self._items[1:,0] += indices[:-1]
self._items[0:,1] += indices
```

Access

Once this is done, every list method requires only a bit of computation and playing with the different key when getting, inserting or setting an item.

Here is the code for the `__getitem__` method. No real difficulty but the possible negative step:

```
def __getitem__(self, key): #get items from a typed list
    if type(key) is int:
        if key < 0:
            key += len(self)
        if key < 0 or key >= len(self):
            raise IndexError("Tuple index out of range")
        dstart = self._items[key][0]
        dstop  = self._items[key][1]
        return self._data[dstart:dstop]

    elif type(key) is slice:
        istart, istop, step = key.indices(len(self))
        if istart > istop:
            istart,istop = istop,istart
        dstart = self._items[istart][0]
        if istart == istop:
            dstop = dstart
        else:
            dstop  = self._items[istop-1][1]
        return self._data[dstart:dstop]
    elif isinstance(key,str):
        return self._data[key][:self._size]

    elif key is Ellipsis:
        return self.data

    else:
        raise TypeError("List indices must be integers")
```

Solve this Quiz!

Q Why makes `Typed List` better than a list while using NumPy?

○ A) All items in a `Typed List` have the same data type unlike list

○ B) `Typed List` takes less computation time unlike list

COMPLETED 0%

1 of 1