Wendelin.core tutorial (or how to get started with Out-of-core NumPy arrays)

Wendelin.core provides *numpy.ndarray* compatible objects which

1. can be transparently persisted,
2. you can work with them using all the usual libraries including C/Fortran/Cython code[1], and
3. their size can be bigger than RAM and bigger than local disk.

Let's see how to get started and work with them.

Setup

First we create test virtualenv Python instance and install wendelin.core

```
# (don't forget to first have virtualenv and Python headers installed, e.g. apt-get install virtualenv python-dev on Debian)
$ virtualenv test
Running virtualenv with interpreter /usr/bin/python2
New python executable in /home/test/test/bin/python2
Also creating executable in /home/test/test/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
$ . test/bin/activate
# install ZODB + wendelin.core
(test) $ pip install wendelin.core
Collecting wendelin.core
  Downloading wendelin.core-0.6.tar.gz (2.6MB)
    100% |████████████████████████████████| 2.6MB 287kB/s
Collecting numpy (from wendelin.core)
  Downloading numpy-1.11.0-cp27-cp27mu-manylinux1_x86_64.whl (15.3MB)
    100% |████████████████████████████████| 15.3MB 47kB/s
Collecting ZODB3>=3.10 (from wendelin.core)
  Downloading ZODB3-3.11.0.tar.gz (55kB)
    100% |████████████████████████████████| 61kB 5.2MB/s
...
Successfully installed BTrees-4.3.1 ZConfig-3.1.0 ZEO-4.2.0b1 ZODB-4.3.1 ZODB3-3.11.0
numpy-1.11.0 persistent-4.2.1 psutil-4.2.0 six-1.10.0 transaction-1.6.1 wendelin.core-0.6
zc.lockfile-1.2.0 zdaemon-4.1.0 zodbpickle-0.6.0 zope.interface-4.2.0

(test) $ pip install ipython   # for interactive testing
```

Now all is installed and we are ready to play.

Persistence

To start let's create some database and array in it:

```
$ ipython
Python 2.7.11+ (default, Jun  2 2016, 19:34:15)
...

# imports
In [1]: from wendelin.bigarray.array_zodb import ZBigArray
In [2]: from wendelin.lib.zodb import dbopen, dbclose
In [3]: import transaction
In [4]: import numpy as np

# open/create database for tests (on local disk for now)
In [5]: root = dbopen('test.fs')

# create 10 items 1d array object
In [6]: root['A'] = A = ZBigArray((10,), np.int)
In [7]: transaction.commit()

# see what it is
In [8]: A
Out[8]: <wendelin.bigarray.array_zodb.ZBigArray at 0x7fcd7dcea5d0>

In [9]: a = A[:]

In [10]: a
Out[10]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [11]: type(a)
Out[11]: numpy.ndarray
```

So we initialized *test.fs* ZODB database and setup top-level object in it keyed *A* to be a *ZBigArray* instance. We can see in

[10] a is *numpy.ndarray* view of *A*, which is initially all zeros.

Let's initialize *A* to some data:

```
In [12]: a[:] = np.arange(10)
In [12]: a[7] = 22

In [13]: a
Out[13]: array([0, 1, 2, 3, 4, 5, 6, 22, 8, 9])
```

above we are just working with *a*, which is usual *numpy.ndarray* as we would work in classical NumPy case. However since *a* is view of a *ZBigArray* (A), even though we changed data directly on *a* -- *numpy.ndarray* object without explicitly notifying *A*, the system knows it and keeps track of the data and changes. Let's save the data to database: In [14]: **transaction.commit()** and finish current process:

```
In [15]: dbclose(root)
In [16]: exit
(test) $
```

After new process is started again we can see the data is there:

```
$ ipython
# imports as [1-4] in previous session
In [1]: from wendelin.bigarray.array_zodb import ZBigArray
In [2]: from wendelin.lib.zodb import dbopen, dbclose
In [3]: import transaction
In [4]: import numpy as np

In [5]: root = dbopen('test.fs')
In [6]: A = root['A']

In [7]: A
Out[7]: <wendelin.bigarray.array_zodb.ZBigArray at 0x7f313416b150>

In [8]: a = A[:] # notice the data is the same as we set it above
In [9]: a
Out[9]: array([ 0,  1,  2,  3,  4,  5,  6, 22,  8,  9])

In [10]: type(a)
Out[10]: numpy.ndarray
```

It is real *numpy.ndarray* for all the code

As was shown above, we can create *numpy.ndarray* views to *ZBigArray* objects. Thus, since views are real ndarrays, we can invoke any real code which accepts ndarray as input. Let's for example compute the mean:

```
In [11]: np.mean(a)
Out[11]: 6.0
```

Notice, above we call C-function *mean()* implemented in NumPy library. It does not know it works on a *ZBigArray* data - all it sees is regular *numpy.ndarray* object with memory.

Let's also see how we can use regular Cython code, which expects only ndarrays, to work on *ZBigArray* data:

```
# in another terminal
$ cat myeven.pyx
cimport numpy as np

def counteven(np.ndarray[np.int_t, ndim=1] a):
    cdef int n = a.shape[0]
    cdef int i, neven=0

    i = 0
    while i < n:
        if a[i] % 2 == 0:
            neven += 1
        i += 1

    return neven

$ cat setup.py
from numpy.distutils.core import setup
from Cython.Build import cythonize
setup(ext_modules = cythonize("myeven.pyx"))

$ pip install cython
$ python setup.py build_ext -i
running build_ext
building 'myeven' extension
... myeven.so extension is built
```

now back to ipython session:

```
In [12]: from myeven import counteven

# verify counteven() accepts only ndarrays
In [13]: counteven([1, 2, 3])
...
TypeError: Argument 'a' has incorrect type (expected numpy.ndarray, got list)

# verify counteven() accepts only ndarrays with dtype numpy.int
In [14]: counteven(np.arange(2, dtype=np.int16))
...
ValueError: Buffer dtype mismatch, expected 'int_t' but got 'short'

In [15]: counteven(np.arange(2, dtype=np.int))
Out[15]: 1

# call counteven() on ZBigArray data - it works correctly
In [16]: counteven(a)
Out[16]: 6
```

Arrays bigger than RAM

First of all, before doing computations on arrays bigger than RAM, we should be able to somehow produce them. Wendelin.core allows both

- to create an array with large initial number of elements, and
- to increase size of existing arrays, appending data to their tail.

in all cases the amount of modifications to array's data in one transaction have to be less than available RAM. The strategy to ingest data into an array is thus to do so in parts.

For reading, on the other hand, the amount of data which can be read in one transaction is limited only by virtual address-space size, which is ~ 127TB on Linux/amd64.

Let's append some data blocks to our array (for real example see demo_zbigarray.py which ingests signal data twice more than RAM)

```
In [17]: for i in range(4):
   ....:     A.append( np.arange(4*1024*1024) )
   ....:     transaction.commit()

In [18]: A.shape
Out[18]: (16777226,)

In [19]: A[:]
Out[19]: array([      0,       1,       2, ..., 4194301, 4194302, 4194303])
```

Doing enough such iterations we can eventually get to array whose size is bigger than local RAM.

Notice: contrary to NumPy, where *numpy.append()* works by copying data to newly allocated array (thus working in O(len(array) + δ) time), ZBigArray objects can be appended in-place without copying, thus *ZBigArray.append()* works in O(δ) time.

After we have array bigger than RAM, we can still call existing functions on it to do computations [1]. In particular *numpy.mean()* and *counteven()* continue to work:

```
In [20]: a = A[:]
In [21]: np.mean(a)
Out[21]: # works -> some value depending on size of A

In [22]: counteven(a)
Out[22]: # works -> some value depending on size of A
```

When, for example *np.mean(a)* runs, the system will be loading array parts from database and reclaiming least-recently-accessed memory pages for new accesses when the amount of loaded data is close to amount of local RAM. All this happens transparent to client code, so it thinks it just accesses plain memory.

Arrays bigger than disk

In the previous section we already saw how to work with arrays bigger than local RAM by using ZODB database located on local disk. Going beyond local disk is possible by using distributed ZODB storage which shards data in a cluster of nodes.

This can be achieved with NEO ZODB storage. Required changes on client side is to adjust *dbopen()* call as follows:

```
In [5]: root = dbopen('neo://dbname@master')
```

And install NEO client library (via **pip install neoppod[client]**).

The system will be then reading and writing data from/to networked distributed database, which scales in size the more storage nodes we add to its cluster.

Full cluster setup is documented in NEO readme. One can setup a simple NEO cluster to play with *neosimple* command (after **pip install neoppod**).

Summary

In this tutorial we briefly overviewed how to use *wendelin.core* library for working with NumPy compatible arrays, which

1. can be transparently persisted,
2. you can work with them using all the usual libraries including C/Fortran/Cython code[1], and
3. their size can be bigger than RAM and bigger than local disk.

wendelin.core is Free Software; The project home page is here: https://lab.nexedi.com/nexedi/wendelin.core

[1] *(1, 2, 3)* provided called functions do not create temporary arrays proportinal to input in size.