mmappickle Documentation

Release 1.0.1

Laurent Fasnacht

Jun 18, 2018

Contents

This Python 3 module enables to store large structures in a python `pickle`, in a way that the array can be memory-mapped instead of being copied into the memory. This module is licensed under the LGPL3 license.

Currently, the container is a dictionnary (`mmappickle.mmapdict`), which keys are unicode strings of less than 256 bytes.

It supports any type of value, but it is only possible to memory map `numpy.ndarray` and `numpy.ma.MaskedArray` at present.

It also supports concurrent access (i.e. you can pass a `mmappickle.mmapdict` as an argument which is called using the `multiprocessing` Python module).

CHAPTER 1

Why use mmappickle?

`Mmappickle` is designed for "unstructured" parallel access, with a strong emphasis on adding new data.

Here are two example uses cases:

- Timelapse hyperspectral scanning

  At each time interval, an hyperspectral image (which is numpy array of a few hundreds of megabytes) is added to the file, along with the related metadata.

  The key features for which `mmappickle` is useful are:

  - is that it is possible to have an independant viewer process, which is monitoring the file as it is appended, and allows the operator to stop the capture once sufficient data was acquired.
  - all the information is kept in one file, which makes it easier to archive and distribute.
  - the images are not stored in RAM, which would be a problem due to their size.

  Moreover, as the file is a normal `pickle`, it is not required to install anything more than a standard distribution of Python to have it work. This is useful when distributing files to occasional users, which may be less familiar with Python, and may not be able to use `pip` to install libraries.

- Image registration

  When having multiples (hyperspectral) images of the same subject, a common requirement is to register these images to allow creating a combined image containing the information of all of them. Depending on the images, this could be for example HDR imaging, or simply stitching the images together.

  Commonly, a set of files are provided to the algorithm, which computes keypoints, then the transformation parameters, and finally creates an output file. Due to practical reasons intermediate data is rarely kept, as it means more files, and also that the references to the original files must be strictly kept.

  With `mmappickle`, all the input images are in one file. The registration algorithm can simply add the keypoints and the transformation parameters to the original file. This simplifies debugging and also serves as a cache when using successively different combining algorithms.

CHAPTER 2

Quick start

`mmappickle.mmapdict` behaves like a dictionary. For example:

```
>>> from mmappickle import mmapdict
>>> m = mmapdict('/tmp/test.mmdpickle') #could be an existing file
>>> m['a_sample_key'] = 'value'
>>> m['other_key'] = [1,2,3]
>>> print(m['a_sample_key'])
value
>>> m['other_key'][2]
3
>>> del m['a_sample_key']
>>> print(m.keys())
['other_key']
```

The contents of the dictionary are stored to disk. For example, in another python interpreter:

```
>>> from mmappickle import mmapdict
>>> m = mmapdict('/tmp/test.mmdpickle')
>>> print(m['other_key'])
[1, 2, 3]
```

It is also possible to open the file in read-only mode, in which case any modification will fail:

```
>>> from mmappickle import mmapdict
>>> m = mmapdict('/tmp/test.mmdpickle', True)
>>> m['other_key'] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/laurent/git/mmappickle/mmappickle/utils.py", line 22, in require_
↪writable_wrapper
    raise io.UnsupportedOperation('not writable')
io.UnsupportedOperation: not writable
```

Of course, the main interest is to store numpy arrays:

```
>>> import numpy as np
>>> from mmappickle.dict import mmapdict
>>> m = mmapdict('/tmp/test.mmdpickle')
>>> m['test'] = np.array([1,2,3], dtype=np.uint8)
>>> m['test'][1] = 4
>>> print(m['test'])
[1 4 3]
>>> print(type(m['test']))
<class 'numpy.core.memmap.memmap'>
```

As you can see, the `m['test']` is now memory-mapped. This means that its content is not loaded in memory, but instead accessed directly from the file.

Unfortunately, the array has to exist in order to serialize it to the `mmapdict`. If the array exceed the available memory, this won't work. Instead one should use stubs:

```
>>> from mmappickle.stubs import EmptyNDArray
>>> m['test_large'] = EmptyNDArray((300,300,300))
>>> print(type(m['test_large']))
<class 'numpy.core.memmap.memmap'>
```

The matrix in `m['test_large']` uses 216M of memory, but it was at no point allocated in RAM. This way, it is possible to allocate arrays larger than the size of the memory. One could have written `m['test_large'] = np.empty((300,300,300))`, but unfortunately the memory is allocated when calling `numpy.empty()`.

Finally, one last useful trick is the `mmappickle.mmapdict.vacuum()` method. It allows reclaiming the disk space:

```
>>> del m['test_large']
>>> #Here, /tmp/test.mmdpickle still occupies ~216M of hard disk
>>> m.vacuum()
>>> #Now the disk space has been reclaimed.
```

Warning: When running `mmappickle.mmapdict.vacuum()`, it is crucial that there are no other references to the file content, either in this process or in other. In particular, no memory-mapped array. If this rule is not followed, unfortunate outcomes are anticipated! (crash, data corruption, etc.)

CHAPTER 3

Getting help

Please use mmappickle issue tracker on GitHub to ask any question.

To report bugs, please see reporting-bugs.