ppmd-cffi

Aug 12, 2021

Contents:

CHAPTER 1

User Guide

PPM, Prediction by partial matching, is a wellknown compression technique based on context modeling and prediction. PPM models use a set of previous symbols in the uncompressed symbol stream to predict the next symbol in the stream.

PPMd is an implementation of PPMII by Dmitry Shkarin.

The ppmd-cffi package uses core C files from p7zip. The library has a bare function and no metadata/header handling functions. This means you should know compression parameters and input/output data sizes.

1.1 Getting started

1.1.1 Install

The ppmd-cffi is written by Python and C language bound with CFFI, and can be downloaded from PyPI(aka. Python Package Index) using standard 'pip' command as like follows;

```
$ pip install ppmd-cffi
```

1.2 Command line

ppmd-cffi provide command line script to hande *.ppmd* file.

To compress file

```
$ ppmd target.dat
```

To decompress ppmd file

```
$ ppmd -x target.ppmd
```

To decompress to STDOUT

```
$ ppmd -x -c target.ppmd
```

1.3 Programming Interfaces

1.4 *.ppmd* file comression/decompression

ppmd-cffi project provide two functions which compress and decompress *.ppmd* archive file. *PpmdCompressor* class provide compress function *compress()* and *PpmdDecompressor* class provide extraction function *decompress()*.

Both classes takes *version*= argument which default is 8, means PPMd Ver. I. Also classes takes *target*, *fname* and *ftime* arguments which is a target file and its properties. *target* should be a file-like object which has *write()* method. *fname* and *ftime* is a file property which is stored in archive as meta data. *fname* should be string, and *ftime* should be a datetime object.

order and *mem_in_mb* parameters will be vary.

1.4.1 Compression with PPMd ver. H

```
targetfile = pathlib.Path('target.dat')
fname = 'target.dat'
ftime = datetime.utcfromtimestamp(targetfile.stat().st_mtime)
archivefile = 'archive.ppmd'
order = 6
mem_in_mb = 16
with archivefile.open('wb') as target:
    with targetfile.open('rb') as src:
        with PpmdCompressor(target, fname, ftime, order, mem_in_mb, version=7) as␣
↪compressor:
            compressor.compress(src)
```

1.4.2 Compression with PPMd ver. I

```
targetfile = pathlib.Path('target.dat')
fname = 'target.dat'
ftime = datetime.utcfromtimestamp(targetfile.stat().st_mtime)
archivefile = 'archive.ppmd'
order = 6
mem_in_mb = 8
with archivefile.open('wb') as target:
    with targetfile.open('rb') as src:
        with PpmdCompressor(target, fname, ftime, order, mem_in_mb, version=8) as␣
↪compressor:
            compressor.compress(src)
```

1.4.3 Decompression

When construct *PpmdDecompressor* object, it read header from specified archive file. The header hold a compression parameters such as PPMd version, order and memory size. It also has a *filename* and *timestamp*. *PpmdDecompressor* select a proper decoder based on header data. You need to handle *filename* and *timestamp* by your self. A decompressor method will write data to specified file-like object, which should have *write()* method.

```
targetfile = pathlib.Path('target.ppmd')
with targetfile.open('rb') as target:
    with PpmdDecompressor(target, target_size) as decompressor:
        extractedfile = pathlib.Path(parent.joinpath(decompressor.filename))
        with extractedfile.open('wb') as ofile:
            decompressor.decompress(ofile)
            timestamp = datetime_to_timestamp(decompressor.ftime)
            os.utime(str(extractedfile), times=(timestamp, timestamp))
```

1.5 Bare encoding/decoding PPMd data

There are several classes to handle bare PPMd data. Note: mem parameter should be as bytes not MB.

- Ppmd7Encoder(dst, order, mem)
- Ppmd7Decoder(src, order, mem)
- Ppmd8Encoder(det, order, mem, restore)
- Ppmd8Decoder(src, order, mem, restore)

CHAPTER 2

Contributor guide

2.1 Development environment

If you're reading this, you're probably interested in contributing to ppmd. Thank you very much! The purpose of this guide is to get you to the point where you can make improvements to the py7zr and share them with the rest of the team.

2.1.1 Setup Python and C compiler

The ppmd is written in the Python and C languages bound with CFFI, C Foreign Function Interface. Python installation for various platforms with various ways. You need to install Python environment which support *pip* command. Venv/Virtualenv is recommended for development.

We have a test suite with python 3.7, 3.8 and pypy3. If you want to run all the test with these versions and variant on your local, you should install these versions. You can run test with CI environment on Github actions.

2.1.2 Get Early Feedback

If you are contributing, do not feel the need to sit on your contribution until it is perfectly polished and complete. It helps everyone involved for you to seek feedback as early as you possibly can. Submitting an early, unfinished version of your contribution for feedback in no way prejudices your chances of getting that contribution accepted, and can save you from putting a lot of work into a contribution that is not suitable for the project.

2.2 Code Contributions

2.2.1 Steps submitting code

When contributing code, you'll want to follow this checklist:

1. Fork the repository on GitHub.

2. Run the tox tests to confirm they all pass on your system. If they don't, you'll need to investigate why they fail. If you're unable to diagnose this yourself, raise it as a bug report.
3. Write tests that demonstrate your bug or feature. Ensure that they fail.
4. Make your change.
5. Run the entire test suite again using tox, confirming that all tests pass including the ones you just added.
6. Send a GitHub Pull Request to the main repository's master branch. GitHub Pull Requests are the expected method of code collaboration on this project.

2.2.2 Code review

Contribution will not be merged until they have been code reviewed. There are limited reviewer in the team, reviews from other contributors are also welcome. You should implemented a review feedback unless you strongly object to it.

2.2.3 Code style

The ppmd uses the PEP8 code style. In addition to the standard PEP8, we have an extended guidelines

- line length should not exceed 125 charactors.
- It also use MyPy static type check enforcement.

CHAPTER 3

Authors

ppmd-cffi is written and maintained by Hiroshi Miura <miurahr@linux.com>

Contributors, listed alphabetically, are:

CHAPTER 4

Glossary

binary file A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also *text file* for a file object able to read and write `str` objects.

bytes-like object An object that supports the *bufferobjects* and can export a C-*contiguous* buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as "read-write bytes-like objects". Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects ("read-only bytes-like objects"); examples of these include `bytes` and a `memoryview` of a `bytes` object.

contiguous A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

file object An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object A synonym for *file object*.

text file A *file object* able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the text encoding automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also *binary file* for a file object able to read and write *bytes-like objects*.

path-like object An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by **PEP 519**.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index