zmxtools

Release 0.1.5

Tom Vettenburg

Apr 08, 2023

CONTENTS:

CHAPTER

ONE

A TOOLKIT TO READ ZEMAX FILES.

Currently, this is limited to unpacking ZAR archives. For further processing of the archive's contents, e.g. ZMX or AGF glass files, please check the *list of related software* below.

1.1 Features

- Unpack a Zemax OpticStudio® Archive ZAR file using the `unzar` command.
- Repack a ZAR file as a standard zip file using the `unzar -z` command.
- Use as a pure Python 3 library.
- Fully typed with annotations and checked with mypy, PEP561 compatible

1.2 Installation

CHAPTER

TWO

PREREQUISITES

- Python 3.8 or higher
- pip, the Python package manager

To install `zmxtools`, just run the following command in a command shell:

```
pip install zmxtools
```

The `zmxtools` library will color-code test output when the `coloredlogs` package is installed. You can optionally install it with

```
pip install coloredlogs
```

2.1 Usage

This package can be used directly from a terminal shell or from your own Python code. Example files can be found on manufacturer's sites such as Thorlabs Inc.

CHAPTER

THREE

COMMAND LINE SHELL

The command `unzar` is added to the path upon installation. It permits the extraction of the zar-file to a sub-directory as well as its conversion to a standard zip-file. For example, extracting to the sub-directory `mylens` is done using

```
unzar mylens.zar
```

Repacking the same zar-archive as a standard zip-archive `mylens.zip` is done with:

```
unzar mylens.zar -z
```

Multiple input files and an alternative output directory can be specified:

```
unzar -i *.zar -o some/where/else/
```

Find out more information and alternative options using:

```
unzar -h
```

CHAPTER

FOUR

AS A PYTHON LIBRARY

Extraction and repacking can be done programmatically as follows:

```
from zmxtools import zar

zar.extract('mylens.zar')
zar.repack('mylens.zar')
zar.read('mylens.zar')
```

Python `pathlib.Path` objects can be used instead of strings.

4.1 Online documentation

The latest version of the API Documentation is published on https://zmxtools.readthedocs.io/. The documentation is generated automatically in the docs/ directory from the source code.

4.2 Contributing to the source code

The complete source code can be found on github: https://github.com/tttom/zmxtools. Check out Contributing for details.

4.3 License

This code is distributed under the agpl3: GNU Affero General Public License

4.4 Credits

- Wouter Vermaelen for decoding the ZAR header and finding LZW compressed contents.
- Bertrand Bordage for sharing this gist.
- This project was generated with ``*wemake-python-package*` <https://github.com/wemake-services/wemake-python-package>`_. Current template version is: cfbc9ea21c725ba5b14c33c1f52d886cfde94416. See what is updated since then.

4.5 Related Software

- Optical ToolKit reads Zemax .zmx files.
- RayTracing reads Zemax .zmx files.
- Zemax Glass reads Zemax .agf files.
- RayOptics reads Zemax .zmx and CODE-V .seq files.
- RayOpt reads Zemax .zmx as well as OSLO files.
- OpticsPy does not read Zemax .zmx files but reads CODE-V .seq files and glass information from data downloaded from https://www.refractiveindex.info/.
- OpticalGlass reads glass manufacturer Excel sheets.

4.5.1 zmxtools

zmxtools package

Submodules

zmxtools.cli module

unzar(*argv=None*)

Function that can be called as a script.

Parameters
argv (Optional[Sequence[str]]) – An optional sequence of input arguments.

Return type
int

Returns
The error code. 0: no error, 1: file skipped, 2: incorrect argument, -1: unexpected fatal error.

zmxtools.zar module

read(*input_full_file*)

Reads a zar archive file and generates a series of (unpacked file name, unpacked file contents) tuples.

The returned Generator produces tuples in the order found in the archive.

Parameters
input_full_file (Union[Path, str]) – The archive or the path to the archive.

Return type
Generator[*UnpackedData*, None, None]

Returns
A Generator of name-data tuples.

class UnpackedData(*file_name*, *unpacked_contents*)

Bases: object

A structure to represent the file blocks in a zar-archive.

Parameters:
name: A string with the name of the file contained in the archive. unpacked_contents: The unpacked (decompressed) bytes of this file.

file_name: *str*

unpacked_contents: *bytes*

__annotations__ = {'file_name': <class 'str'>, 'unpacked_contents': <class 'bytes'>}

__dataclass_fields__ = {'file_name': Field(name='file_name',type=<class 'str'>,default=<dataclasses._MISSING_TYPE object>,default_factory=<dataclasses._MISSING_TYPE object>,init=True,repr=True, hash=None,compare=True,metadata=mappingproxy({}),kw_only=False,_field_type=_FIELD), 'unpacked_contents': Field(name='unpacked_contents',type=<class 'bytes'>,default=<dataclasses._MISSING_TYPE object>,default_factory=<dataclasses._MISSING_TYPE object>,init=True,repr=True, hash=None,compare=True,metadata=mappingproxy({}),kw_only=False,_field_type=_FIELD)}

__hash__ = None

__init__(*file_name*, *unpacked_contents*)

__match_args__ = ('file_name', 'unpacked_contents')

__repr__()
Return repr(self).

extract(*input_full_file*, *output_path=None*)
Imports the data from a zar archive file and writes it as a regular directory.

Parameters

- **input_full_file** (Union[Path, str]) – The path to zar-file.
- **output_path** (Union[Path, str, None]) – The path where the files should be saved. Default: the same as the input_full_file but without the extension.

Return type
None

repack(*input_full_file*, *output_full_file=None*)
Imports the data from a zar archive file and writes it as a regular zip file.

Parameters

- **input_full_file** (Union[Path, str]) – The file path, including the file name, of the zar-file.
- **output_full_file** (Union[Path, str, None]) – TThe file path, including the file name, of the destination zip-file. Default: the same as *input_full_file* but with the extension changed to 'zip'.

Return type
None

4.5.2 Version history

We follow Semantic Versions.

Releases 0.1

The first release series provides basic decompression and conversion tools, both as command line tool and as a Python3 library.

Version 0.1.5

- Made all non-standard dependencies optional.

Version 0.1.4

- Security update of dependencies.
- Automated API-documentation generation.

Version 0.1.3

- Refactored command-line interface code and the unit tests.

Version 0.1.2

- Proper API docs, big fixes, and automated testing improved.

Version 0.1.1

- Bug and documentation fixes.

Version 0.1.0

- Initial release

4.5.3 How to contribute

Dependencies

We use poetry to manage the dependencies.

To install them you would need to run `install` command:

```
poetry install
```

To activate your `virtualenv` run `poetry shell`.

One magic command

Run `make test` to run everything we have!

Tests

We use `pytest` and `flake8` for quality control. We also use wemake_python_styleguide to enforce the code quality.

To run all tests:

```
pytest
```

To run linting:

```
flake8 .
```

Keep in mind: default virtual environment folder excluded by flake8 style checking is `.venv`. If you want to customize this parameter, you should do this in `setup.cfg`. These steps are mandatory during the CI.

Type checks

We use `mypy` to run type checks on our code. To use it:

```
mypy zmxtools tests/**/*.py
```

This step is mandatory during the CI.

Submitting your code

We use trunk based development (we also sometimes call it `wemake-git-flow`).

What the point of this method?

1. We use protected `main` branch, so the only way to push your code is via pull request
2. We use issue branches: to implement a new feature or to fix a bug create a new branch named `issue-$TASKNUMBER`
3. Then create a pull request to `main` branch
4. We use `git tag`s to make releases, so we can track what has changed since the latest release

So, this way we achieve an easy and scalable development process which frees us from merging hell and long-living branches.

In this method, the latest version of the app is always in the `main` branch.

Before submitting

Before submitting your code:

1. Add tests for code changes
2. Include or update the inline documentation.
3. Update README.md as necessary.
4. Increase the version number in `pyproject.toml` under the `[tool.poetry]` header
5. Update CHANGELOG.md with a quick summary of your changes
6. Run `make test` to:
    1. check code behaviour with `pytest`
    2. ensure that types are correct with `mypy`
    3. enforce code style using `flake8`
    4. verify the documentation with `doc8`
7. Run `make clean html` in the `docs` folder and check for documentation errors.
8. Commit the changes with a descriptive message.
9. Tag the release with the version number in `pyproject.toml` using `git tag 0.0.0`.
10. Raise a pull-request.

Other help

You can contribute by spreading a word about this library. It would also be a huge contribution to write a short article on how you are using this project. You can also share your best practices with us.

PYTHON MODULE INDEX

Z

INDEX

Symbols

E

F

M

R

U

Z