O3as: Ozone assessment service

B. Esteban, M. Hardt, T. Kerzenmacher, V. Kozlov (KIT)

Jan 29, 2021

CONTENTS

O3as is a service within the framework of the European Open Science Cloud - Synergy (EOSC-Synergy) project, mainly for scientists working on the Chemistry-Climate Model Initiative (CCMI) for the quadrennial global assessment of ozone depletion due in 2022.

The O3as service shall provide an invaluable tool to extract ozone trends from large climate prediction model data to produce figures of stratospheric ozone trends in publication quality, in a coherent way.

CHAPTER

ONE

USER DOCUMENTATION

1.1 User documentation

CHAPTER

TWO

COMPONENT DOCUMENTATION

2.1 O3as REST API (o3api)

2.1.1 O3as REST API

O3as REST API provides access to O3as (Ozone assessment) data for plotting and high-level analysis. The API leverages Swagger, Flask, and Connexion.

o3api reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

api

O3as REST API methods:

o3api.api.**get_metadata**(**args*, ***kwargs*)
Return information about the package

Returns The o3api package info

Return type dict

o3api.api.**get_model_info**(**args*, ***kwargs*)
Return information about the Ozone model

Returns Info about the Ozone model

Return type dict

o3api.api.**list_models**(**args*, ***kwargs*)
Return the list of available Ozone models

Returns The list of available models

Return type dict

o3api.api.**plot**(**args*, ***kwargs*)
Main plotting routine

Parameters **kwargs** – The provided in the API call parameters

Returns Either PDF plot or JSON document

plots

O3as helper classes to extact data for plotting:

class o3api.plots.**DataSelection**(*plot_type*, ***kwargs*)
Class to perform data selection, based on *Dataset*.

Parameters

- **begin** – Year to start data scanning from
- **end** – Year to finish data scanning
- **month** – Month(s) to select, if not a whole year
- **lat_min** – Minimum latitude to define the range (-90..90)
- **lat_max** – Maximum latitude to define the range (-90..90)

get_1980slice(*model*)
Function to select the slice for 1980 (reference year)

Parameters **model** – The model to process

Returns xarray dataset selected according to the time and latitude

Return type xarray

get_dataslice(*model*)
Function to select the slice of data according to the time and latitude requested

Parameters **model** – The model to process

Returns xarray dataset selected according to the time and latitude

Return type xarray

class o3api.plots.**Dataset**(*plot_type*, ***kwargs*)
Base Class to initialize the dataset

Parameters **plot_type** – The plot type (e.g. tco3_zm, vmro3_zm, . . .)

get_dataset(*model*)
Load data from the datafile list

Parameters **model** – The model to process

Returns xarray dataset

Return type xarray

class o3api.plots.**ProcessForTCO3**(***kwargs*)
Subclass of *DataSelection* to calculate tco3_zm

get_plot_data(*model*)
Plot tco3_zm data applying a smoothing function (boxcar) :param model: The model to process for tco3_zm :return: ready for plotting data :rtype: pandas series (pd.Series)

get_raw_data(*model*)
Process the model to get tco3_zm raw data

Parameters **model** – The model to process for tco3_zm

Returns raw data points in preparation for plotting

Return type pandas series (pd.Series)

get_ref1980(*model*)
Process the model to get tco3_zm reference for 1980

Parameters **model** – The model to process for tco3_zm

Returns xarray dataset for 1980

Return type xarray

class o3api.plots.**ProcessForTCO3Return**(***kwargs*)
Subclass of *DataSelection* to calculate tco3_return

get_plot_data(*model*)
Process the model to get tco3_return data for plotting

Parameters **model** – The model to process for tco3_return

Returns xarray dataset for plotting

Return type xarray

class o3api.plots.**ProcessForVMRO3**(***kwargs*)
Subclass of *DataSelection* to calculate vmro3_zm

get_plot_data(*model*)
Process the model to get vmro3_zm data for plotting

Parameters **model** – The model to process for vmro3_zm

Returns xarray dataset for plotting

Return type xarray

o3api.plots.**set_data_processing**(*plot_type*, ***kwargs*)
Function to inizialize proper class for data processing

Parameters **plot_type** – The plot type (e.g. tco3_zm, vmro3_zm, . . .)

Returns object corresponding to the plot type

plothelpers

O3as help functions to create figures:

o3api.plothelpers.**clean_models**(***kwargs*)
Clean models from empty entries, spaces, and quotes

Parameters **kwargs** – The provided in the API call parameters

Returns models cleaned from empty entries, spaces, quotes

Return type list

o3api.plothelpers.**get_date_range**(*ds*)
Return the range of dates in the provided dataset

Parameters **ds** – xarray dataset to check

Returns date_min, date_max

o3api.plothelpers.**get_periodicity**(*pd_time*)
Calculate periodicity in the provided data

Parameters **pd_time** (*pandas DatetimeIndex*) – The time period

Returns Calculated periodicity as the number of points per year

Return type int

o3api.plothelpers.**set_figure_attr**(*fig*, ***kwargs*)
Configure the figure attributes

Parameters

- **fig** – Figure instance
- **kwargs** – The provided in the API call parameters

Returns none

o3api.plothelpers.**set_filename**(***kwargs*)
Set file name

Parameters **kwargs** – The provided in the API call parameters

Returns file_name with added input parameters (no extension given!)

Return type string

o3api.plothelpers.**set_plot_title**(***kwargs*)
Set plot title

Parameters **kwargs** – The provided in the API call parameters

Returns plot_title with added input parameters

Return type string

2.1.2 See also

Indices and tables

- genindex
- modindex
- search
- *O3as REST API*: O3as REST API
- *o3api reference*: o3api references
- <TBD>

2.2 O3as Skimming (o3skim)

Data skimming for ozone assessment.

o3skim is an open source project and Python package that provides the tools to pre-process, standardize and reduce ozone data from netCDF models to simplify and speed up ozone data transfer and plot.

2.2.1 Getting started

- Which *Prerequisites* you need to start.
- How to *Build* your o3skim container.
- How to *Deployment* your o3skim container.
- How to use the o3skim *Command Line Interface*
- Create your *Source file* to point what to skim.

Prerequisites

To run the project as container, you need the following systems and container technologies:

- **Build machine** with docker in case you want/need to build your own image.
- **Runtime machine** with udocker and access to the data to skim.

In case you do not want to create your image, last images are uploaded in dockerhub at synergyimk.

Note udocker cannot build containers but run them.

Build

Download the code from the o3skim repository at the **Build machine**.

For example, using git:

```
$ git clone git@git.scc.kit.edu:synergy.o3as/o3skim.git
Cloning into 'o3skim'...
...
```

Build the container image at the **Build machine**.

For example, using docker:

```
$ docker build --tag o3skim .
...
Successfully built 69587025a70a
Successfully tagged o3skim:latest
```

If the build process succeeded, then you should see the image name on the container images list:

```
$ docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED
↪           SIZE
o3skim                              latest              69587025a70a        xx seconds␣
↪ago       557MB
...
```

To use your new generated image on the **Runtime machine**, the easiest way is to push to a dockerhub repository. For example, with docker:

```
$ docker push <repository>/o3skim:<tag>
The push refers to repository [docker.io/........./o3skim]
...
```

(continues on next page)

(continued from previous page)

```
7e84795fccac: Preparing
7e84795fccac: Layer already exists
ffaeb20d9e23: Layer already exists
4cdd6a90e552: Layer already exists
3e0762bebc71: Layer already exists
1e441fe06d90: Layer already exists
98ff2784e9f5: Layer already exists
2b99e2403063: Layer already exists
d0f104dc0a1f: Layer already exists
...: digest: sha256:..... size: 2004
```

If you do not have internet access from the **Build machine** or **Runtime machine** it is also possible to use docker save to export your images.

Deployment

To deploy the the application using **udocker** at the **Runtime machine** you need the o3skim container image.

The easiest way to deploy in your **Runtime machine** is by pulling the image from a remote registry. You can use the official registry at synergyimk or use the instructions at *Build* to create your image and uploaded at your own registry.

Once you decide from which registry download, pull the image that image registry. For example, to pull it from the synergy-imk official registry use:

```
$ udocker pull synergyimk/o3skim
...
Downloading layer: sha256:......
...
```

Note it is also possible to use udocker load to import images generated by docker save.

Once the image is downloaded or imported, create the local container. For example, if it was downloaded from synergyimk registry you can use:

```
$ udocker create --name=o3skim synergyimk/o3skim
fa42a912-b0d4-3bfb-987f-1c243863802d
```

Check the containers available at the **Runtime machine**:

```
$ udocker ps
CONTAINER ID                          P M NAMES              IMAGE
...
fa42a912-b0d4-3bfb-987f-1c243863802d . W ['o3skim']         synergyimk/o3skim:latest
```

Now you are ready to start using the container as *o3skim*. Read how to use the *Command Line Interface* as first steps to skim your data.

Command Line Interface

Usage:

```
usage: main [-h] [-f SOURCES_FILE] [-s {year,decade}]
            [-v {DEBUG,INFO,WARNING,ERROR,CRITICAL}]
```

To run the the application using **udocker** at the **Runtime machine** you need to provide the following volumes to the container:

- –volume, mount */app/data*: Input path with data to skim.
- –volume, mount */app/output*: Output path for skimmed results.
- –volume, mount */app/sources.yaml*: Configuration file with a data structure description at the input path in YAML format. See ../user_guide/source-file for a configuration example.

Also, in the specific case of udocker, it is needed to specify that the user *application* should run inside the container:

For example,to run the container using udocker use the following:

```
$ udocker run \
  --user=application \
  --volume=${PWD}/sources.yaml:/app/sources.yaml \
  --volume=${PWD}/data:/app/data \
  --volume=${PWD}/output:/app/output \
  o3skim --verbosity INFO
...
executing: main
...
2020-08-25 12:42:34,151 - INFO     - Configuration found at: './sources.yaml'
2020-08-25 12:42:34,152 - INFO     - Loading data from './data'
2020-08-25 12:42:34,261 - INFO     - Skimming data to './output'
```

For the main function description and commands help you can call:

```
$ udocker run --user=application o3skim --help
```

As optional arguments, it is possible to indicate:

- -h, –help: show this help message and exit
- -f, –sources_file SOURCES_FILE: Custom sources YAML configuration. (default: ./sources.yaml)
- -s, –split_by {year,decade}: Period time to split output (default: None)
- -v, –verbosity {DEBUG,INFO,WARNING,ERROR,CRITICAL}: Sets the logging level (default: ERROR)

Note that SOURCES_FILE is only modified for development purposes as usually any file from host can be mounted using the container directive '–volume'.

Source file

This is an example configuration file for sources to be skimmed

Note the following **metadata_1** for keys and values (always following YAML standards):

- **CUSTOMIZABLE_KEY**: Indicates the key can be any value
- **FIXED_KEY**: The must match the example string
- **CORRECT_VALUE**: The value must be in line with the source data

Note the following **metadata_2** for keys and values (always following YAML standards):

- **MANDATORY**: The key/value is mandatory to exist inside the section
- **OPTIONAL**: The key/value is optional to exist inside the section

For example: At the 3rd level, the variables are specified. The configuration specification for variables are "[**FIXED_KEY – OPTIONAL**]"; If the variable key *tco3_zm* is specified, the application searches for tco3 data on the source. When it is not specified, variable data are not searched so the output folder [x-]_[y-] does not contain tco3 dataset files.

First example - CCMI-1

In this example, the data source has only one model, therefore it is expected to have only one folder output named "CCMI-1_IPSL".

This model has 2 variables (*tco3_zm* and *vmro3_zm*) which datasets are located in different directories. Therefore the key *path* is the different in both of them. Therefore, the output expected at "CCMI-1_IPSL" is 2 type of files:

- tco3_zm_[YEAR]-[YEAR].nc: With tco3 skimmed data
- vmro3_zm_[YEAR]-[YEAR].nc: With vmro3 skimmed data

Where [YEAR] are optional text output depending on how the *–split_by* argument is configured at the *Command Line Interface* call.

```
# This is the preceded -x1- string at the output folder: '[x1]_[y-]'
# [CUSTOMIZABLE_KEY -- MANDATORY]
CCMI-1:

    # Source metadata; common to all models in this source
    # [FIXED_KEY -- OPTIONAL]
    metadata:

        # A metadata information example related to the source
        # [CUSTOMIZABLE_KEY -- OPTIONAL]
        meta_0: Source metadata string example

        # A metadata information example related to the source
        # [CUSTOMIZABLE_KEY -- OPTIONAL]
        meta_1: Source metadata example replaced later by model

    # This is the preceded -y1- string at the output folder: '[x1]_[y1]'
    # [CUSTOMIZABLE_KEY -- MANDATORY]
    IPSL:

        # Model metadata; Unique key values for this model
        # [FIXED_KEY -- OPTIONAL]
```

(continues on next page)

(continued from previous page)

```
    metadata:

        # A metadata information example related to the source
        # [CUSTOMIZABLE_KEY -- OPTIONAL]
        meta_1: Replaces the metadata from the source

        # A metadata information example related to the source
        # [CUSTOMIZABLE_KEY -- OPTIONAL]
        meta_2: Model metadata string example

    # Represents the information related to tco3 data
    # [FIXED_KEY -- OPTIONAL]
    tco3_zm:

        # TCO3 metadata; metadata for variable tco3
        # [FIXED_KEY -- OPTIONAL]
        metadata:

            # A tco3 metadata attribute example
            # [CUSTOMIZABLE_KEY -- OPTIONAL]
            meta_0: Structured as tco3_zm: -> meta_0:

        # Variable name for tco3 array inside the dataset
        # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
        name: toz

        # Reg expression, how to load the netCDF files
        # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
        paths: Ccmi/mon/toz/*.nc

        # Coordinates description for tco3 data.
        # [FIXED_KEY -- MANDATORY]:
        coordinates:

            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            time: time

            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            lat: lat

            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            lon: lon

    # Represents the information related to vmro3 data
    # [FIXED_KEY -- OPTIONAL]
    vmro3_zm:

        # VMRO3 metadata; metadata for variable vmro3
        # [FIXED_KEY -- OPTIONAL]
        metadata:

            # A vmro3 metadata attribute example
            # [CUSTOMIZABLE_KEY -- OPTIONAL]
            meta_0: Structured as vmro3_zm: -> meta_0:

        # Variable name for vmro3 array inside the dataset
        # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
```

(continues on next page)

(continued from previous page)

```
            name: vmro3

            # Reg expression, how to load the netCDF files
            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            paths: Ccmi/mon/vmro3

            # Coordinates description for vmro3 data.
            # [FIXED_KEY -- MANDATORY]:
            coordinates:

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                time: time

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                plev: plev

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                lat: lat

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                lon: lon
```

Second example - ECMWF

In this example, the data source has two models, therefore it is expected to have two folder outputs ["ECMWF_ERA-5", "ECMWF_ERA-i"].

The model ERA-5 has only information tco3 data, there is no vmro3 data. Therefore, only one type of files is expected at "ECMWF_ERA-5":

- tco3_zm_[YEAR].nc: With tco3 skimmed data

This case of ERA-i indeed has 2 variables (*tco3_zm* and *vmro3_zm*) but in this case, are located inside the same dataset files, therefore the key *path* should be the same for both variables. The output expected at "ECMWF_ERA-5" are 2 type of files:

- tco3_zm_[YEAR].nc: With tco3 skimmed data
- vmro3_zm_[YEAR].nc: With vmro3 skimmed data

```
# This is the preceded -x2- string at the output folder: '[x2]_[y-]'
# [CUSTOMIZABLE_KEY -- MANDATORY]
ECMWF:

    # Source metadata; common to all models in this source
    # [FIXED_KEY -- OPTIONAL]
    metadata:

        # A metadata information example related to the source
        # [CUSTOMIZABLE_KEY -- OPTIONAL]
        meta_0: Source metadata string example

        # A metadata information example related to the source
        # [CUSTOMIZABLE_KEY -- OPTIONAL]
        meta_1: Source metadata example replaced later by model
```

(continues on next page)

(continued from previous page)

```
    # This is the preceded -y1- string at the output folder: '[x2]_[y1]'
    # [CUSTOMIZABLE_KEY -- MANDATORY]
    ERA-5:

        # Model metadata; Unique key values for this model
        # [FIXED_KEY -- OPTIONAL]
        metadata:

            # A metadata information example related to the source
            # [CUSTOMIZABLE_KEY -- OPTIONAL]
            meta_1: Replaces the metadata from the source

            # A metadata information example related to the source
            # [CUSTOMIZABLE_KEY -- OPTIONAL]
            meta_2: Model metadata string example

        # Represents the information related to tco3 data
        # [FIXED_KEY -- OPTIONAL]
        tco3_zm:

            # TCO3 metadata; metadata for variable tco3
            # [FIXED_KEY -- OPTIONAL]
            metadata:

                # A tco3 metadata attribute example
                # [CUSTOMIZABLE_KEY -- OPTIONAL]
                meta_0: Structured as tco3_zm: -> meta_0:

            # Variable name for tco3 array inside the dataset
            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            name: tco3

            # Reg expression, how to load the netCDF files
            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            paths: Ecmwf/Era5

            # Coordinates description for tco3 data.
            # [FIXED_KEY -- MANDATORY]:
            coordinates:

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                lat: latitude

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                lon: longitude

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                time: time

    # This is the preceded -y2- string at the output folder: '[x2]_[y2]'
    # [CUSTOMIZABLE_KEY -- MANDATORY]
    ERA-i:

        # Model metadata; Unique key values for this model
        # [FIXED_KEY -- OPTIONAL]
        metadata:
```

(continues on next page)

(continued from previous page)

```
        # A metadata information example related to the source
        # [CUSTOMIZABLE_KEY -- OPTIONAL]
        meta_1: Replaces the metadata from the source

        # A metadata information example related to the source
        # [CUSTOMIZABLE_KEY -- OPTIONAL]
        meta_2: Model metadata string example

    # Represents the information related to tco3 data
    # [FIXED_KEY -- OPTIONAL]
    tco3_zm:

        # TCO3 metadata; metadata for variable tco3
        # [FIXED_KEY -- OPTIONAL]
        metadata:

            # A tco3 metadata attribute example
            # [CUSTOMIZABLE_KEY -- OPTIONAL]
            meta_0: Structured as tco3_zm: -> meta_0:

        # Variable name for tco3 array inside the dataset
        # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
        name: toz

        # Reg expression, how to load the netCDF files
        # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
        paths: Ecmwf/Erai

        # Coordinates description for tco3 data.
        # [FIXED_KEY -- MANDATORY]:
        coordinates:

            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            time: time

            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            lat: latitude

            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            lon: longitude

    # Represents the information related to vmro3 data
    # [FIXED_KEY -- OPTIONAL]
    vmro3_zm:

        # VMRO3 metadata; metadata for variable vmro3
        # [FIXED_KEY -- OPTIONAL]
        metadata:

            # A vmro3 metadata attribute example
            # [CUSTOMIZABLE_KEY -- OPTIONAL]
            meta_0: Structured as vmro3_zm: -> meta_0:

        # Variable name for vmro3 array inside the dataset
        # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
        name: vmro3
```

(continues on next page)

(continued from previous page)

```
            # Reg expression, how to load the netCDF files
            # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
            paths: Ecmwf/Erai

            # Coordinates description for vmro3 data.
            # [FIXED_KEY -- MANDATORY]:

            coordinates:
                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                time: time

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                plev: level

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                lat: latitude

                # [FIXED_KEY -- MANDATORY]: [CORRECT_VALUE -- MANDATORY]
                lon: longitude
```

One or two files?

Note this two examples should be located into the same file when you want the module to skim both examples with only one call to the command.

If need to skim the data in two different steps, you can place each example into a different file and call each one of them separately by the module using the input argument:

```
-f, --sources_file SOURCES_FILE
```

2.2.2 Developer guide

- How to do a *Local installation*
- How to use o3skim package *o3skim package*
- How to run and create new *Tests*

Local installation

To run tests, you need to install the tool in your system without docker.

As first step ensure you have the following dependencies:

- python > 3.6
- pip > 20.0.2
- gcc
- g++_

After downloading the repository and installing dependencies check, install with pip:

```
$ pip install -e .
```

Tests

Testing is based on sqa-baseline criteria, tox automation is used to simplify the testing process.

To install it with pip use:

```
$ pip install tox
```

To run unit and functional tests together with reports use:

```
$ tox
...
clean: commands succeeded
stylecheck: commands succeeded
bandit: commands succeeded
docs: commands succeeded
py36: commands succeeded
report: commands succeeded
...
```

The last coverage report output is produced at **htmlcov** which can be displayed in html format accessing to **index.html**.

The last Pep8 report produced by flake8 at the output file **flake8.log**.

However, you can also run different test configurations using different tox environments:

Code Style [QC.Sty]

Test pep8 maintenance style conventions based on pylint format. To run stylecheck use:

```
$ tox -e stylecheck
...
stylecheck: commands succeeded
...
```

Unit Testing [QC.Uni]

All unit tests are placed inside the package (./o3skim/test). This helps to test easily functions at low level and ensure the functions have the expected behavior. To run unit tests use:

```
$ tox -e unittesting
...
unittesting: commands succeeded
...
```

This environment also provide a coverage term report for the tests. The design of Unit Tests is based on the python unittest framework, a simple and extended test framework which ships by default together with python.

The usage is very simple and straight forward for simple tests, but the difficulty to parametrize and combine multiple test fixtures makes it not suitable for Black-Box testing without a very complex customization.

Functional Testing [QC.Fun]

Located inside tests package folder (./tests). Functional testing is used to test the system from a general overview of the application. To run functional tests use:

```
$ tox -e functional
...
functional: commands succeeded
...
```

This environment also provide a coverage term report for the tests. The framework used is pytest to provide a simple syntax to test all possible combinations from the user point of view.

Pytest detects directly all tests following the test_discovery naming conventions. Therefore all functional tests should be located on the **tests** folder at the package root and start with **test**. For example *test_sources.py*.

More than 500 test combinations are generated using which otherwise might not be feasible using other python test frameworks.

Security [QC.Sec]

Security checks are performed by bandit, a tool designed to find common security issues in Python code. To run security checks use:

```
$ tox -e functional
...
functional: commands succeeded
...
```

Documentation [QC.Doc]

Documentation is build using sphinx, a tool designed to create documentation based on code. To run documentation build checks use:

```
$ tox -e docs
...
docs: commands succeeded
...
```

The HTML pages are build inside in docs/_build.

o3skim package

O3as package with classes and utilities to handle ozone data skimming.

The main usage of the package resides on the generation of "Source" class instances from the loading of netCDF files though collection descriptions generated by source files.

However, as the generation of collection descriptions in a form of dict type might be difficult, a yaml file can be converted into the required variable by using the util "load" with the path to the 'sources.yaml' file as input.

Once the desired collection is formated in the shape of dict variable, a source object can be created from using the class "Source". During this step the model data is loaded into the source instance involving on the background process the data sorting and standardization. Note that those models with errors on the collection specification are not loaded

neither interrupt the loading process. However, notice of those event is logged on strerr together with the stack execution info.

After the data has been loaded the object intstance correctly created, it is possible to use the internal methods to generate a reduced output on the current folder.

To simplify the management of the data and output directories, the package offest a "cd" utility which can be used together with a "with" statement to change temporary the location of the execution folder.

class o3skim.**Source**(*name*, *metadata={}*, ***collections*)

Conceptual class for a data source. It is produced by the loading and standardization of multiple data models.

The current supported model variables are "tco3_zm" and "vmro3_zm", which should contain the information on how to retrieve the data from the netCDF collection.

Parameters

- **name** (*str*) – Name to provide to the source.
- **metadata** (*dict, optional*) – Source metadata, defaults to {}.
- ****collections** – kwarg where each 'key' is the model name and its 'value' another dictionary with the variable loading statements for that model. {name:str, paths: str, coordinates: dict, metadata: dict}

skim(*groupby=None*)

Request to skim all source data into the current folder.

The output is generated into multiple folder where each model output is generated in a forder with the source name defined at the source initialization followed by '_' and the model name: "<source_name>_<model_name>". If there was metadata added when creating the source, it is delivered into a "metadata.yaml" file on the directory.

Parameters **groupby** (*str, optional*) – How to group output (None, 'year', 'decade').

o3skim.**cd**(*dir*)

Changes the directory inside a 'with' context. When the code reaches the end of the 'with' block or the code fails, the previous folder is restored.

Parameters **dir** (*str*) – Path folder where to change the working directory.

o3skim.**load**(*yaml_file*)

Loads the .yaml file and returns a python dictionary with all the yaml keys and values. Note a yaml file can have key:values sotored inside values, therefore the returned dictionary might have dictionaries stored inside values.

Parameters **yaml_file** (*str*) – Path pointing to the yaml configuration file.

Returns Dictionary with the yaml structure key:value.

Return type dict

2.2.3 Authors

- @T.Kerzenmacher - Data scientist at KIT-IMK
- @V.Kozlov - Code developer at KIT-SCC
- @B.Esteban - Code developer at KIT-SCC
- *Getting started*: Start skimming your ozone data.
- *Developer guide*: Customize and support learning the details.
- *Authors*: Authors and acknowledgements.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX